A Deep Learning Framework for Character Motion Synthesis and Editing

Citation for published version:
Holden, D, Saito, J & Komura, T 2016, 'A Deep Learning Framework for Character Motion Synthesis and Editing' ACM Transactions on Graphics, vol. 35, no. 4, 138. DOI: 10.1145/2897824.2925975

Digital Object Identifier (DOI):
10.1145/2897824.2925975

Link:
Link to publication record in Edinburgh Research Explorer

Document Version:
Peer reviewed version

Published In:
ACM Transactions on Graphics

A Deep Learning Framework for Character Motion Synthesis and Editing

Daniel Holden[*]
University of Edinburgh

Jun Saito[†]
Marza Animation Planet

Taku Komura[‡]
University of Edinburgh

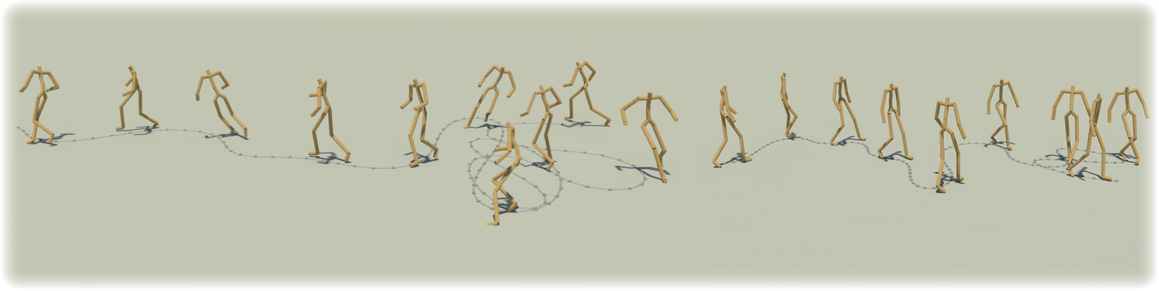

Figure 1: *Our framework allows the animator to synthesize character movements automatically from given trajectories.*

Abstract

We present a framework to synthesize character movements based on high level parameters, such that the produced movements respect the manifold of human motion, trained on a large motion capture dataset. The learned motion manifold, which is represented by the hidden units of a convolutional autoencoder, represents motion data in sparse components which can be combined to produce a wide range of complex movements. To map from high level parameters to the motion manifold, we stack a deep feedforward neural network on top of the trained autoencoder. This network is trained to produce realistic motion sequences from parameters such as a curve over the terrain that the character should follow, or a target location for punching and kicking. The feedforward control network and the motion manifold are trained independently, allowing the user to easily switch between feedforward networks according to the desired interface, without re-training the motion manifold. Once motion is generated it can be edited by performing optimization in the space of the motion manifold. This allows for imposing kinematic constraints, or transforming the style of the motion, while ensuring the edited motion remains natural. As a result, the system can produce smooth, high quality motion sequences without any manual pre-processing of the training data.

Keywords: deep learning, convolutional neural networks, autoencoder, human motion, character animation, manifold learning

Concepts: •**Computing methodologies** → **Motion capture;**

1 Introduction

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

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

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

We also propose techniques to edit the motion data in the space of the motion manifold. The hidden units of the convolutional autoencoder represent the motion in a sparse and continuous fashion, such that adjusting the data in this space preserves the naturalness and smoothness of the motion, while still allowing complex movements of the body to be reproduced. One demonstrative example of this editing is to combine the style of one motion with the timing of another by minimizing the difference in the Gram matrices of the hidden units of the synthesized motion and that of the reference

[*]email:s0822954@sms.ed.ac.uk
[†]email:saito@marza.com
[‡]email:tkomura@ed.ac.uk

SIGGRAPH '16 Technical Paper, July 24 - 28, 2016, Anaheim, CA,
ISBN: 978-1-4503-4279-7/16/07
DOI: http://dx.doi.org/10.1145/2897824.2925975

style motion [Gatys et al. 2015].

In summary, our contribution is the following:

- A fast, parallel deep learning framework for synthesizing character animation from high level parameters.
- A method of motion editing on the motion manifold for satisfying user constraints and transforming motion style.

2 Related Work

We first review kernel-based methods for motion synthesis, which have been the main technique for synthesizing motions via blending motion capture data. We next review methods of interactive character control, where user instructions are applied for synthesizing novel motion using a motion database. Finally, we review methods in deep learning and how they have been applied to character animation.

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

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

Interactive Character Control For interactive character control, a mechanism to produce a continuous motion based on the high level commands is required. Motion graphs [Arikan and Forsyth 2002; Lee et al. 2002; Kovar et al. 2002] are an effective data structure for such a purpose. Motion graphs are automatically computed from a large set of motion capture data. As motion graphs only replay captured motion data, techniques to blend motions in the same class are proposed to enrich the dataset [Min and Chai 2012; Heck and Gleicher 2007; Safonova and Hodgins 2007; Shin and Oh 2006; Levine et al. 2012].

Many of the methods that apply motion blending for motion synthesis during interactive character control require the motions to be classified, segmented and aligned to produce a rich model of each motion class. Poses from different times or different classes must not be mixed. Although Kovar and Gleicher [2004] try to automate this process, the choice of the distance metric between motions and segmentation criteria of the motion sequence can significantly affect the performance and the accuracy. The requirement of explicitly conducting these steps can be a bottleneck of their performance. On the contrary, our unsupervised framework automatically blends nearby motions for synthesizing realistic movements, without any requirement of motion segmentation and classification.

For finding the best action at each time step following user instruction, techniques based on reinforcement learning are applied [Lee and Lee 2004; Safonova and Hodgins 2007; Lee et al. 2010; Levine et al. 2012]. Reinforcement learning requires a huge amount of precomputation which increases exponentially with respect to the number of actions available to the character. For this reason, methods such as motion fields [Lee et al. 2010] quickly become intractable as the amount of data and the number of control parameters grow, and various additional techniques need to be used to reduce the number of states. For example, Levine et al. [2012] classify the motion dataset into different classes in advance and reduce the dimensionality within each motion class. We avoid using reinforcement learning and directly regress the high level commands given by the user to the low level motion features. Such a mapping is computed by stochastic gradient descent and does not grow with respect to the number of control parameters such as in reinforcement learning.

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

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

There have been a few approaches to apply deep learning to human motion capture data. Du et al. [2015] construct a hierarchical RNN using a large motion dataset from various sources and show their method achieves a state-of-the-art recognition rate. Holden et al. [2015] apply a convolutional autoencoder to the CMU motion capture database and show the learned representation achieves good performance in tasks such as motion retrieval.

In terms of motion synthesis, Taylor et al. [2009; 2011] apply conditional Restricted Boltzmann Machines (cRBM) for synthesizing gait animation. Mittelman et al. [2014] use the spike-and-slab version of the recurrent temporal RBM to improve the reconstruction. Fragkiadaki et al. [2015] introduce Encoder-Recurrent-Decoder (ERD) networks, a type of RNN that combines representation learning with learning temporal dynamics, and produce smooth

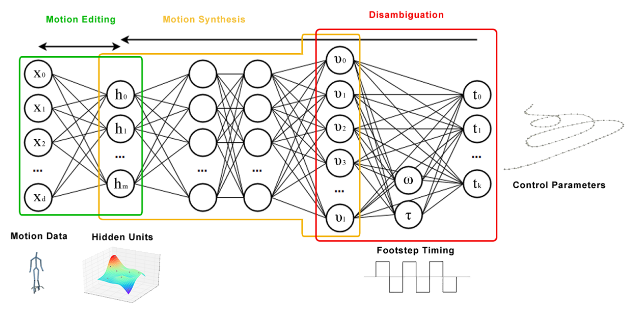

Figure 2: *The outline of our method. High level parameterizations are disambiguated and used as input to feed forward neural networks. These networks produce motion in the space of the hidden units of a convolutional autoencoder which can be further used to edit the generated motion.*

interpolated movements while reducing drifting. These are time-series approaches, where computing the entire motion requires integration from the first frame. We find this framework is not very suitable for the purpose of animation production as animators wish to see the entire motion at once and then sequentially apply minor edits while revising the motion. They do not wish to see edits happening at early frames being propagated into the future, which will be the case when time-series approaches are adopted. For this reason, in this paper we adopt and improve on the convolutional autoencoder representation [Holden et al. 2015] which can produce motion at once, in parallel, without performing any integration process.

3 System Overview

The outline of the system is shown in Fig. 2. Using data from a large human motion database (see Section 4), a convolutional autoencoder is trained and thus a general motion manifold is found (green box in Fig. 2, see Section 5). After this training, motion can be represented by the hidden units of the network. Given this representation, a mapping is produced between high level control parameters and the hidden units via a feedforward neural network stacked on top of the convolutional autoencoder (orange box in Fig. 2, see Section 6). The high level control parameters shown in this work are the trajectory of the character over the terrain and the movement of the end effectors. As these parameterizations can contain ambiguities, another small network is used to compute parameters which disambiguate the input (red box in Fig. 2, see Section 6.3). Just the subset of the motion capture data relevant to the task is used to train these feedforward networks. Using this framework, the user can produce an animation of the character walking and running by drawing a curve on the terrain, or the user can let the character punch and kick by specifying the trajectories of the hands and feet. Once a motion has been generated, it can be edited in the space of hidden units, such that the resulting motion satisfies constraints such as positional constraints for foot-skate cleanup (see Section 7.1). Using this technique we describe a method to transform the style of the character motion using a short stylized clip as a reference (Section 7.2).

4 Data Acquisition

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

4.1 The Motion Dataset for Deep Learning

We construct a motion dataset for deep learning by collecting many freely available large online databases of motion capture [CMU ; Müller et al. 2007; Ofli et al. 2013; Xia et al. 2015], as well as adding data from our internal captures, and retargeting them to a uniform skeleton structure with a single scale and the same bone lengths. The retargeting is done by first copying any corresponding joint angles in the source skeleton structure to the target skeleton structure, then scaling the source skeleton to the same size as the target skeleton, and finally performing a full-body inverse kinematics scheme [Yamane and Nakamura 2003] to move the joints of the target skeleton to match any corresponding joint positions in the source skeleton. Once constructed, the final dataset is about twice the size of the CMU motion capture database and contains around six million frames of high quality motion capture data for a single character sampled at 120 frames per second.

4.2 Data Format for Training

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

We find the joint position representation effective for several reasons: the Euclidean distance between two poses in this representation closely matches visual dissimilarity of the poses, multiple poses can be interpolated with simple linear operators, and many constraints are naturally formulated in this representation.

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

5 Building the Motion Manifold

To construct a manifold over the space of human motion we build an autoencoding convolutional neural network and train it on the complete motion database. We follow the approach by Holden et al. [2015], but adopt a different setup for optimizing the network for motion synthesis. First, we only use a single layer for encoding the motion, as multiple layers of pooling/de-pooling can result in blurred motion after reconstruction due to the pooling layers of the network reducing the temporal resolution of the data. Using only one layer lacks the power to further abstract the low level features so this task is performed by the deep feedforward network described in Section 6. We also change several components to improve the performance of the network training and bases quality. The details are described below.

5.1 Network Structure

In our framework, the convolutional autoencoder performs a one-dimensional convolution over the temporal domain, independently for each filter. The network provides a *forward operation* $\mathbf{\Phi}$ (encoding) and a *backward operation* $\mathbf{\Phi}^\dagger$ (decoding). The forward operation receives the input vector $\mathbf{X}$ in the visible unit space and outputs the encoded values $\mathbf{H}$ in the hidden unit space.

The *forward operation*:

$$\mathbf{\Phi}(\mathbf{X}) = ReLU(\mathbf{\Psi}(\mathbf{X} * \mathbf{W}_0 + \mathbf{b}_0)), \tag{1}$$

consists of a convolution (denoted $*$) using weights matrix $\mathbf{W}_0 \in \mathbb{R}^{m \times d \times w_0}$, addition of a bias $\mathbf{b}_0 \in \mathbb{R}^m$, a max pooling operation $\mathbf{\Psi}$, and the nonlinear operation $ReLU(\mathbf{x}) = \max(\mathbf{x}, 0)$, where w_0 is the temporal filter width and m is the number of hidden units in the autoencoding layer, each set to 25 and 256 in this work; a temporal filter width of 25 ensures each filter roughly corresponds to half a second of motion, while 256 hidden units is experimentally found to produce good reconstruction.

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

The *backward operation*:

$$\mathbf{\Phi}^\dagger(\mathbf{H}) = (\mathbf{\Psi}^\dagger(\mathbf{H}) - \mathbf{b}_0) * \tilde{\mathbf{W}}_0. \tag{2}$$

takes hidden units $\mathbf{H} \in \mathbb{R}^{\frac{n}{2} \times m}$ as input, and consists of an inverse-pooling operation $\mathbf{\Psi}^\dagger$, a subtraction of a bias $\mathbf{b}_0$ and convolution using the weights matrix $\tilde{\mathbf{W}}_0$. $\tilde{\mathbf{W}}_0 \in \mathbb{R}^{d \times m \times w_0}$ is simply the weights matrix $\mathbf{W}_0$ reflected on the temporal axis, and transposed on the first two axes, used to invert the convolution operation.

When performing the inverse pooling operation, each unit in the hidden layer must produce two units in the visible layer (those which were pooled during the forward operation). This operation is therefore non-invertible and an approximation must be used. During training $\mathbf{\Psi}^\dagger$ randomly picks between the two corresponding visible units and assigns the complete value to one of those units, leaving the other unit set to zero. This represents a good approximation of the inverse of the maximum operation but introduces noise into the result. Therefore when performing synthesis $\mathbf{\Psi}^\dagger$ acts like an average pooling operation and spreads the hidden unit value evenly across both visible units.

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

5.2 Training the Autoencoder

We train the network to reproduce some input $\mathbf{X}$ following both the *forward* and *backward* operations. Training is therefore given as

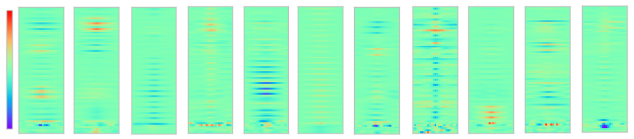

Figure 3: *A selection of convolutional filters from the autoencoder. Here the horizontal axis represents time, while the vertical axis represents the degrees of freedom. Filters are sparse (mainly zero), and show strong temporal and inter-joint correlations.*

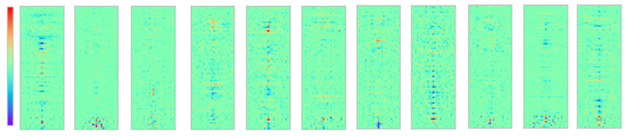

Figure 4: *Convolutional filters learned by a network without dropout or max pooling. Compared to the filters in Fig. 3 these filters are noisy and have not learned any strong temporal correlations as most of their activations are located around the central frame of the filter.*

an optimization problem. We minimize the following cost function with respect to the parameters of the network, $\theta = \{\mathbf{W}_0, \mathbf{b}_0\}$:

$$Cost(\mathbf{X}, \theta) = \|\mathbf{X} - \mathbf{\Phi}^\dagger(\mathbf{\Phi}(\mathbf{X}))\|_2^2 + \alpha \, \|\theta\|_1 \tag{3}$$

In this equation, the first term measures the squared reproduction error and the second term represents an additional sparsity term that ensures the minimum number of network parameters are used to reproduce the input. This is scaled by some small constant α which in our case is set to 0.1.

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

Once training is complete the filters express strong temporal and inter-joint correspondences. A visualisation of the resulting weights can be seen in Fig. 3. Here it is shown that each filter expresses the movement of several joints over a period of time.

6 Mapping High Level Parameters to Human Motions

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

In the rest of this section, we first describe about the structure of the feedforward network and how it can be trained, and then about the details of the high level parameters for the locomotion task.

6.1 Structure of the Feedforward Network

We now describe the feedforward convolutional network which maps the high level parameters $\mathbf{T}$ to the hidden layer of the autoencoding network constructed in the previous section, such that eventually the system outputs a motion of the character $\mathbf{X} \in \mathbb{R}^{n \times d}$.

The feedforward convolutional network uses a similar *forward operation* as defined in Eq. (1) but contains three layers, and an additional operation $\boldsymbol{\Upsilon}$, which is a task-specific operation to resolve the ambiguity problem. We will discuss more about it in Section 6.3. The feedforward operation is given by the following:

$$\begin{aligned}\boldsymbol{\Pi}(\mathbf{T}) = ReLU(\boldsymbol{\Psi}(ReLU(ReLU(\boldsymbol{\Upsilon}(\mathbf{T}) \\ * \mathbf{W}_1 + \mathbf{b}_1) * \mathbf{W}_2 + \mathbf{b}_2) * \mathbf{W}_3 + \mathbf{b}_3)),\end{aligned} \quad (4)$$

where $\mathbf{W}_1 \in \mathbb{R}^{h_1 \times l \times w_1}$, $\mathbf{b}_1 \in \mathbb{R}^{h_1}$, $\mathbf{W}_2 \in \mathbb{R}^{h_2 \times h_1 \times w_2}$, $\mathbf{b}_2 \in \mathbb{R}^{h_2}$, $\mathbf{W}_3 \in \mathbb{R}^{m \times h_2 \times w_3}$, $\mathbf{b}_3 \in \mathbb{R}^{m}$, h_1, h_2 are the number of hidden units in the two hidden layers of the feedforward network, w_1, w_2, w_3 are the filter widths of the three convolutional operators and l is the DOF of the high level parameters, which are set to 64, 128, 45, 25, 15 and 7, respectively. These filter widths ensure each frame of motion is generated using roughly one second of trajectory information. The parameters of this feedforward network used for regression are therefore given by $\phi = \{\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3, \mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$.

6.2 Training the Feedforward Network

To train the regression between the high level parameters and the output motion, we minimize a cost function using stochastic gradient descent in the same way as explained in Section 5.2, but this time with respect to the parameters of the feedforward network, keeping the parameters of the autoencoding network fixed. The cost function is defined as the following and consists of two terms:

$$Cost(\mathbf{T}, \mathbf{X}, \phi) = \|\mathbf{X} - \boldsymbol{\Phi}^{\dagger}(\boldsymbol{\Pi}(\mathbf{T}))\|_2^2 + \alpha \, \|\phi\|_1 \quad (5)$$

The first term computes the mean squared error of the regression and the second term is a sparsity term to ensure the minimum number of hidden units are used to perform the regression. As before, α is set to 0.1.

When training this network, we only use data relevant to the task. For example, during the locomotion task we only use the locomotion-specific data. Training therefore takes significantly less time than the autoencoder. For the locomotion task, training is performed for 200 epochs and takes approximately 1 hour.

6.3 Disambiguation for Locomotion

In this section, we describe about our solution to disambiguate the locomotion, given a curve drawn on the terrain. A curve on a terrain alone does not give enough information to fully describe the motion

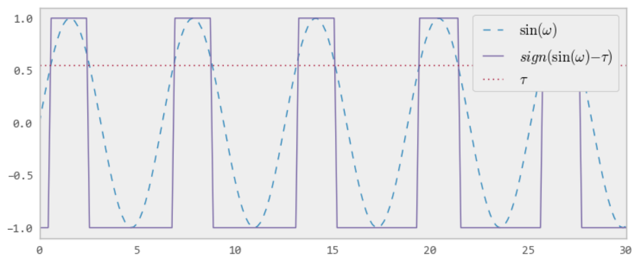

Figure 5: *A square wave representing the foot contact computed from a sin wave. The foot is in contact with the ground when the value is 1.*

that should be produced due to the ambiguity problem mentioned above. We examined various types of inputs, and discovered that providing the timing when the feet is in contact with the ground can greatly disambiguate the locomotion. Indeed, the contact timing even distinguishes walking and running as there is always a double support phase in walking, and there is a flying phase in running. We therefore train a model which can be used to automatically compute foot contacts from a given trajectory. We include this in the input to the feedforward network to resolve the ambiguity.

The input to this network is the trajectory in the form of translational velocities on the XZ plane and rotational velocity around the Y axis, given for each timestep and relative to the forward facing direction of the path $\mathbf{T} \in \mathbb{R}^{n \times k}$, where n is the number of frames in the trajectory and k is the dimensionality of the trajectory input, which is 3. The character height is considered constant. This input is passed to the function $\boldsymbol{\Upsilon}$ in Eq. (4), which adds foot contact information to the trajectory input:

$$\boldsymbol{\Upsilon}(\mathbf{T}) = [\mathbf{T} \;\; \mathbf{F}], \quad (6)$$

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

Modeling Contact States by Square Waves: We model the states of the four contacts using four square waves and learn the parameters of these waves from the data in a way that allows us to compute them from the trajectory $\mathbf{T}$. The four square waves that model the four contacts, and produce $\mathbf{F}$, are defined as follows:

$$\mathbf{F}(\omega, \tau) = \begin{bmatrix} sign(\sin(c\,\omega + a^h) - b^h - \tau^{lh}) \\ sign(\sin(c\,\omega + a^t) - b^t - \tau^{lt}) \\ sign(\sin(c\,\omega + a^h + \pi) - b^h - \tau^{rh}) \\ sign(\sin(c\,\omega + a^t + \pi) - b^t - \tau^{rt}) \end{bmatrix}^{\mathsf{T}}, \quad (7)$$

where ω and τ control the *frequency* and *step duration* at each frame of motion (see Fig. 5), and are the parameters we are interested in computing from the trajectory.

These parameters alone are enough to produce contact information, but to allow for more artistic control some user parameters are also provided. a^h, a^t are constants that adjust the phases of the heels and toes, b^h, b^t are constants that can be used to adjust the contact duration of the heels and toes (therefore forcing the character to walk or run), and c is a constant that can scale the frequency of the stepping. In our experiments, we set a^h, a^t to -0.1 and zero, b^h, b^t to zero, and c to one. Below, we describe how to extract ω and τ from the locomotion data.

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

$$\tau_i = \cos \frac{\pi d_i}{u_i + d_i}. \tag{8}$$

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

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

$$\mathbf{\Gamma}(\mathbf{T}) = ReLU(\mathbf{T} * \mathbf{W}_4 + \mathbf{b}_4) * \mathbf{W}_5 + \mathbf{b}_5 \tag{9}$$

Here $\mathbf{W}_4 \in \mathbb{R}^{h_4 \times k \times w_4}$, $\mathbf{b}_4 \in \mathbb{R}^{h_4}$, $\mathbf{W}_5 \in \mathbb{R}^{l \times h_4 \times w_5}$, $\mathbf{b}_5 \in \mathbb{R}^{l}$ are the parameters of this network, where w_4, w_5 are the filter widths, h_4 is the number of hidden units, and k, l are the DOF of $\mathbf{T}$, $\mathbf{\Gamma}$ at each frame, respectively, which are 3 and 5. This network is trained using stochastic gradient descent as explained in Section 5.2.

Once this network is trained, given some trajectory $\mathbf{T}$, it computes the values for τ, $\mathbf{\Delta}\omega$, which can be used to calculate $\mathbf{F}$ using Eq. (7), therefore producing foot contact information for the trajectory.

7 Motion Editing in Hidden Unit Space

In this section, we describe how to edit or transform the style of the motion in the space of hidden units, which is the abstract representation of the motion data learned by the autoencoder. Because the motion is edited in the space of the hidden units, which parameterize the manifold over valid motion, it ensures that even after editing, the motion remains smooth and natural. We represent constraints as costs with respect to the values of hidden units. This formulation of motion editing as a minimization problem is often convenient and powerful as it specifies the desired result of the edit without inferring any technique of achieving it. We first describe an approach to apply kinematic constraints (see Section 7.1) and then about adjusting the style of the motion in the space of hidden units (see Section 7.2).

7.1 Applying Constraints in Hidden Unit Space

Because the scope of motion editing is very large we start by describing how to apply constraints in the hidden space using three common constraints often found in character animation as examples: positional constraints, bone length constraints, and trajectory constraints, but our approach is applicable to other types of constraints providing the constraint can be described using a cost function. Note that all these costs compute the error for all frames simultaneously, summed over the temporal domain.

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

$$Pos(\mathbf{H}) = \sum_j \|\mathbf{v}_r^{\mathbf{H}} + \omega^{\mathbf{H}} \times \mathbf{p}_j^{\mathbf{H}} + \mathbf{v}_j^{\mathbf{H}} - \mathbf{v}'_j\|_2^2. \tag{10}$$

where $\mathbf{v}'_j \in \mathbb{R}^{n \times 3}$ is the target velocity of joint j in the body coordinate system, and $\mathbf{v}_r^{\mathbf{H}}, \mathbf{p}_j^{\mathbf{H}}, \mathbf{v}_j^{\mathbf{H}} \in \mathbb{R}^{n \times 3}, \omega^{\mathbf{H}} \in \mathbb{R}^n$ are the root velocity, joint j's local position and velocity, and the body's angular velocity around the Y axis, respectively, computed from the hidden unit values $\mathbf{H}$ by the decoding operation $\mathbf{\Phi}^\dagger(\mathbf{H})$ in Eq. (2), for all the frames. For example, in order to avoid foot sliding, the heel and toe velocity must be zero when they are in contact with the ground.

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

$$Bone(\mathbf{H}) \quad = \quad \sum_i \sum_b |\|\mathbf{p}_{b_{j_1}}^{\mathbf{H}i} - \mathbf{p}_{b_{j_2}}^{\mathbf{H}i}\| - l_b|^2 \tag{11}$$

where b is the index for the set of bones in the body, $\mathbf{p}_{b_{j_1}}^{\mathbf{H}i}, \mathbf{p}_{b_{j_2}}^{\mathbf{H}i}$ are the reconstructed 3D positions of the two end joints of b at frame i, computed by the decoding operation $\mathbf{\Phi}^\dagger(\mathbf{H})$ in Eq. (2) and l_b is the length of bone b.

Trajectory Constraints: As a result of motion editing or error in the synthesis motion may not exactly follow the desired trajectory. We may therefore need to additionally constrain the motion to some trajectory precisely. The cost for such a constraint can be written as follows:

$$Traj(\mathbf{H}) \quad = \quad \|\omega^{\mathbf{H}} - \omega'\|_2^2 + \|\mathbf{v}_r^{\mathbf{H}} - \mathbf{v}'_r\|_2^2 \tag{12}$$

Projection to Null Space of Constraints: The motion generated by the autoencoder is adjusted in the space of hidden units via gradient descent until the total cost converges within a threshold:

$$\mathbf{H}' = arg \min_{\mathbf{H}} \; Pos(\mathbf{H}) + Bone(\mathbf{H}) + Traj(\mathbf{H}). \tag{13}$$

By minimizing Eq. (13) and projecting the found $\mathbf{H}'$ back into the visible unit space using Eq. (2), we can constrain the joints to the desired position while keeping the rigidity of each bone.

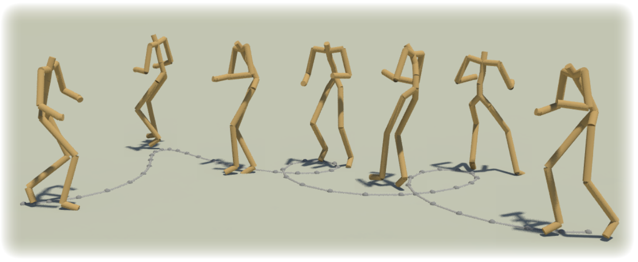

Figure 6: *A side stepping motion produced from a velocity profile from some test data and an angular velocity profile drawn by Maya.*

7.2 Motion Stylization in Hidden Unit Space

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

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

$$Style(\mathbf{H}) = s\|G(\mathbf{\Phi}(\mathbf{S})) - G(\mathbf{H})\|_2^2 + c\|\mathbf{\Phi}(\mathbf{C}) - \mathbf{H}\|_2^2 \quad (14)$$

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

$$G(\mathbf{H}) = \frac{\sum_i^n \mathbf{H}_i \mathbf{H}_i^\intercal}{n}. \quad (15)$$

Unlike in Section 7.1, where $\mathbf{H}$ is found via motion synthesis or the *forward operation* of the autoencoder, to avoid a bias toward either *content* or *style*, $\mathbf{H}$ is initialized from white noise and a stylized motion is found by optimizing Eq. (14) until convergence using adaptive gradient descent with automatic derivatives calculated via Theano. The computed motion is *then* edited using Eq. (13) to satisfy kinematic constraints.

8 Experimental Results

We now show some experimental results of training and synthesizing character movements. We first show examples of animating character movements using high level parameters and the framework described in Section 6, with projection to the null space of constraints as described in Section 7.1. We next show examples of applying stylization using the framework described in Section 7.2. As our system has a fast execution at runtime it is suitable for creating animation of large crowds. We therefore show such an example of this. We then evaluate the autoencoder representation by comparing its performance with comparable network structures. Finally

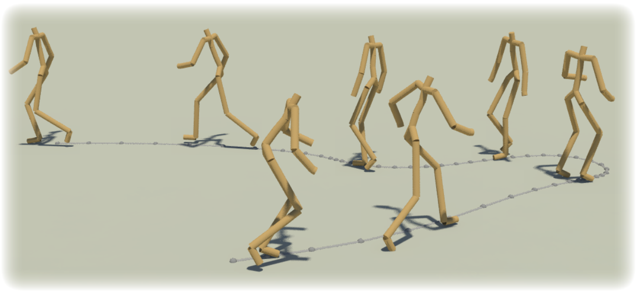

Figure 7: *A locomotion including transition from walk to stop and run.*

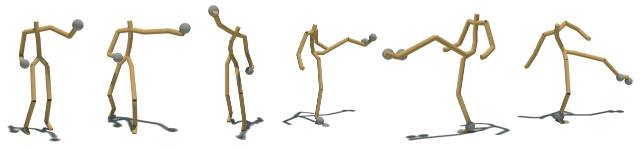

Figure 8: *The character performs punching and kicking to attack the given targets.*

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

Locomotion on the Terrain The feedforward network is trained such that a curve drawn on the terrain is used to generate the actual locomotion of the character. Among the data in the database, various types of locomotion data with different speed and stepping patterns are used to train the system. Using the training data, the trajectory of the root is projected onto the ground to produce a terrain curve to be used as an input.

During runtime, curves drawn by Maya are first used to produce walking and running animation. In the first two examples, the speed of the character is constant. The timing that the heels and toes are in contact with the ground is automatically generated from the curve and used as the input to the feedforward network. The character walks when the velocity is low and runs when the velocity is high (see Fig. 1). In the next example, we take the body velocity profile from some test set not used in the training. We also add some turning motion by drawing the angular velocity profile by Maya (see Fig. 6). In the final example, we use a speed profile from a test data item where the character accelerates and decelerates. This is applied to a terrain curve drawn by Maya. A transition from walking to stopping and running appears as a result (see Fig. 7).

Punching and Kicking Control We show another example where the feedforward network is set up such that the character punches and kicks to follow end effector trajectories provided by the user. We tested the system using test data not included in the training set. The character generates full body movements that follow the trajectories of the end effectors. Some snapshots of the animation are shown in Fig. 8.

Motion Editing in the Hidden Unit Space Here we show the motions before and after applying the constraints to the those generated by the feedforward network. We also compare the motion edited in the hidden unit space with those edited in the Cartesian space by inverse kinematics. The former produces much smoother results as the motion is edited on the leanred manifold. The results

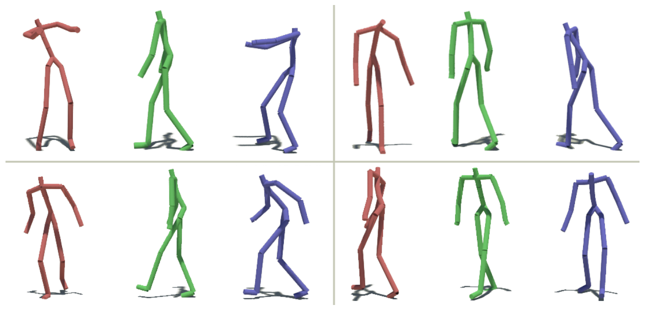
Figure 9: *Several animations are generated with the timing from one clip and the style of. Red: input style motions. Green: input timing motions. Blue: output motion. In a clockwise order the styles used are* zombie, depressed, old man, injured.

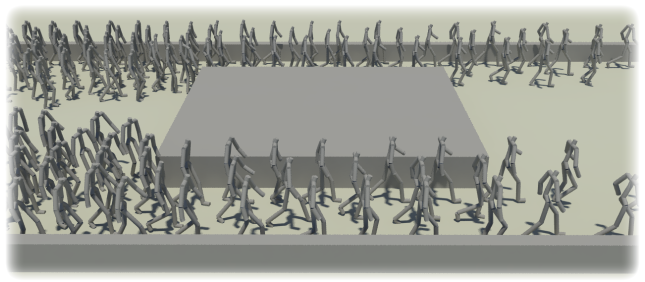
Figure 10: *Crowd motion for 200 characters is generated in parallel using the GPU.*

are presented in the video.

Transforming the Style of the Motion We next show an example of transforming the style of the character's locomotion using a separate motion clip. Style data where the character walks in a (1) zombie style (2) depressed style (3) old man style and (4) injured style are given, and they are passed through the autoencoder to compute the Gram matrices. These Gram matrices are used for converting the style of the given locomotion data using the optimization method described in Section 7.2. Locomotion data is taken from the dataset, while style data is a combination of internal captures, and captures from [Xia et al. 2015]. The snapshots of the animation are shown in Fig. 9.

Crowd Animation Our approach allows for parallel computation across the timeline using the GPU. This allows us to create motion for many characters at once. We make use of this feature to apply our system for synthesizing an animation of a large set of characters using the terrain curve framework described in Section 6. Results of this are shown in Fig. 10.

Comparing the Autoencoder with Other Network Structures Here we evaluate the representation found by the autoencoder. We compare its performance to a naive construction of a neural network without max pooling or dropout. If trained without dropout or max pooling the network does not learn strong temporal coherence, as can be seen in Fig. 4. This motion manifold without temporal coherence is fairly similar to a per-pose PCA - when applied to the style transfer task, because there is no temporal smoothness encoded in this model, the output is extremely noisy as shown in Fig. 11. In the motion synthesis task the neural network without max pooling or dropout actually has a slightly lower mean error, but similar noise was present in results. This can be seen in Fig. 12.

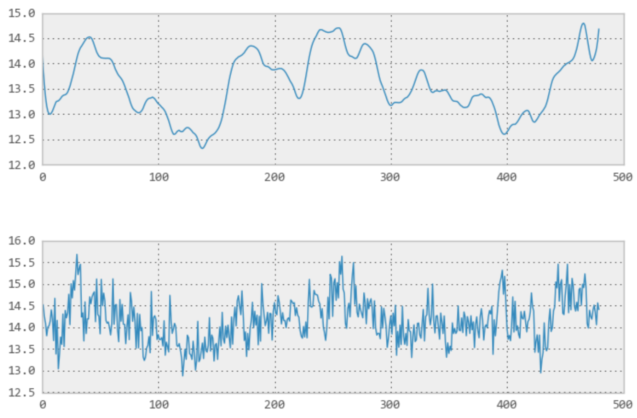

Figure 11: *Two graphs of the vertical movement of the hand joint in motion generated in the style transfer task. Top: movement when the neural network uses dropout and max pooling. Bottom: movement when the neural network does not use these operations - the movement is very noisy due to the lack of temporal correlation encoded in the motion manifold.*

Breakdown of the Computation In this section we give a breakdown of the various timings of the computation for each result presented. This is shown in Table 1. All examples of animation are generated at a sample rate of 60fps. For the crowd scene the frame rate is given by finding the total number of frames generated across all 200 characters. All results are produced using a NVIDIA GeForce GTX 660 and *Theano*. Our technique clearly scales well as the total time required to generate results remains similar, even with long sequences of animation or many characters. All times are given in seconds. In particular our technique works well on the crowd scene due to the fact it can run in parallel both across characters, and across the timeline.

9 Discussions

Many other approaches to motion synthesis are time-series approaches [Taylor and Hinton 2009; Taylor et al. 2011; Mittelman et al. 2014; Xia et al. 2015], but our approach to motion synthesis is a *procedural* approach as it does not require step-by-step calculation, and individual frames at arbitrary times can be generated on demand. This makes it a good fit for animation production software such as *Maya* which allows animators to jump to arbitrary points in the timeline. Animators also do not need to worry about changes affecting the generated animation outside of the local convolution window. This also makes the system highly parallelizable, as motion for all frames can be computed independently. As a result the trajectories of many characters can be generated during runtime on the GPU at fast rates. Generating motion in parallel across the timeline requires continuity between frames. This is handled by our framework in two ways. The high level continuity (such as the timing) is provided by the generation of foot contact information, while the low level continuity (smoothness etc.) is ensured by the manifold.

Although procedural approaches are not new in character animation [Lee and Shin 1999; Kim et al. 2009; Min et al. 2009], previous methods require the motion to be segmented, aligned and labeled before the data can be included into the model. On the contrary, our model automatically learns the model from a large set of mo-

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

Table 1: *Performance breakdown.*

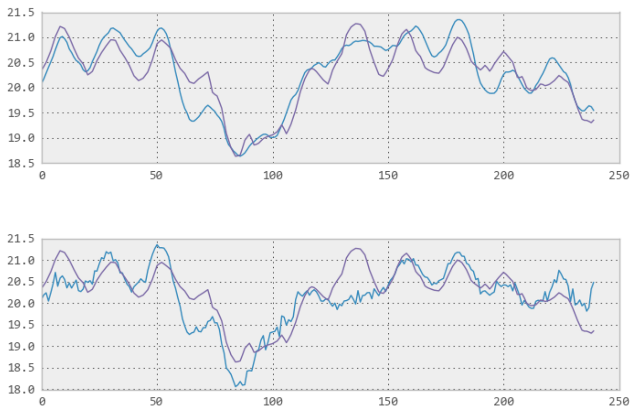

Figure 12: *Two graphs of the vertical movement of the spine joint in motion generated in the motion synthesis task. Purple Line: Ground Truth. Blue Line: Generated Movement. Top: movement when the neural network uses dropout and max pooling - although the movement does not follow exactly, the signal is smooth. Bottom: movement when the neural network does not use these operations - the movement fits more closely to the ground truth but has visible high frequency noise.*

tion data without manual labeling or segmentation. This makes the system highly practical as the users can easily add the new motion data into the training set to enrich the model.

Our convolutional filters only shift along the temporal axis; a natural question to ask is if this convolution can also be used spatially, for example, over the graph structure of the character. The idea of using a temporal convolutional model is to ensure the learned bases of the autoencoder are local and invariant - that their influence is limited to a few frames, and that they can appear anywhere in the timeline. These assumptions of locality and invariance do not generalize well in the spatial domain. There are strong correlations between separate parts of the body according to the motion (for example, arms and legs synchronized during walking), and it is difficult to confine the influence along the graph structure. Also, the bases such as those for the arm are in general not applicable to other parts of the body, which shows the structure is not invariant. It is to be noted that our filters do capture the correlation of different joints; the signals of different DOFs are summed in the convolution operation in Eq. (1), and thus the filters are optimized to discover correlated movements.

There are some important parameters of the system, which are determined carefully taking into account the nature of human movements and through experimental results. These include the filter widths of the human motion (w_0 in Eq. (1)) and those of the trajectories for the feedforward network (w_1, w_2, w_3 in Eq. (4)). For the filter width of the human motion ($w_0 = 25$), this covers about half-a-second. Setting this value too long results in the motions to be smoothed out excessively or even fail to train. Setting it too short will make the system work like per-pose training, where the smoothness of motion cannot be learned from the data. For the feedforward network, the filter width is set a little longer ($w_1 = 45, w_2 = 25, w_3 = 15$), such that the character prepares early enough for future events. Setting this too short will result in lack of variations, and too long will result in overfitting, where odd movements are produced for novel trajectories. These values are initially set through intuition and fine-tuned through visual analysis.

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

10 Conclusion

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

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

Acknowledgements

We thank the reviewers for the fruitful suggestions. This research is supported by Marza Animation Planet.

References

ALLEN, B. F., AND FALOUTSOS, P. 2009. Evolved controllers for simulated locomotion. In *Motion in games*. Springer, 219–230.

ARIKAN, O., AND FORSYTH, D. A. 2002. Interactive motion generation from examples. *ACM Trans on Graph 21*, 3, 483–490.

BERGSTRA, J., BREULEUX, O., BASTIEN, F., LAMBLIN, P., PASCANU, R., DESJARDINS, G., TURIAN, J., WARDE-FARLEY, D., AND BENGIO, Y. 2010. Theano: a CPU and GPU math expression compiler. In *Proc. of the Python for Scientific Computing Conference (SciPy)*. Oral Presentation.

CMU. Carnegie-Mellon Mocap Database. http://mocap.cs.cmu.edu/.

DU, Y., WANG, W., AND WANG, L. 2015. Hierarchical recurrent neural network for skeleton based action recognition. In *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*.

FRAGKIADAKI, K., LEVINE, S., FELSEN, P., AND MALIK, J. 2015. Recurrent network models for human dynamics. In *Proc. of the IEEE International Conference on Computer Vision*, 4346–4354.

GATYS, L. A., ECKER, A. S., AND BETHGE, M. 2015. A neural algorithm of artistic style. *CoRR abs/1508.06576*.

GOODFELLOW, I., POUGET-ABADIE, J., MIRZA, M., XU, B., WARDE-FARLEY, D., OZAIR, S., COURVILLE, A., AND BENGIO, Y. 2014. Generative adversarial nets. In *Proc. of Advances in Neural Information Processing Systems*. 2672–2680.

GRAVES, A., MOHAMED, A.-R., AND HINTON, G. 2013. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, IEEE, 6645–6649.

GROCHOW, K., MARTIN, S. L., HERTZMANN, A., AND POPOVIĆ, Z. 2004. Style-based inverse kinematics. *ACM Trans on Graph 23*, 3, 522–531.

HARIHARAN, B., ARBELÁEZ, P. A., GIRSHICK, R. B., AND MALIK, J. 2014. Hypercolumns for object segmentation and fine-grained localization. *CoRR abs/1411.5752*.

HECK, R., AND GLEICHER, M. 2007. Parametric motion graphs. In *Proc. of the 2007 Symposium on Interactive 3D Graphics and Games*, ACM, 129–136.

HINTON, G. 2012. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, G. Montavon, G. Orr, and K.-R. Mller, Eds., vol. 7700 of *Lecture Notes in Computer Science*. 599–619.

HOLDEN, D., SAITO, J., KOMURA, T., AND JOYCE, T. 2015. Learning motion manifolds with convolutional autoencoders. In *SIGGRAPH Asia 2015 Technical Briefs*, ACM, 18:1–18:4.

KIM, M., HYUN, K., KIM, J., AND LEE, J. 2009. Synchronized multi-character motion editing. *ACM Trans on Graph 28*, 3, 79.

KINGMA, D. P., AND BA, J. 2014. Adam: A method for stochastic optimization. *CoRR abs/1412.6980*.

KOVAR, L., AND GLEICHER, M. 2004. Automated extraction and parameterization of motions in large data sets. *ACM Trans on Graph 23*, 3, 559–568.

KOVAR, L., GLEICHER, M., AND PIGHIN, F. 2002. Motion graphs. *ACM Trans on Graph 21*, 3, 473–482.

KRIZHEVSKY, A., SUTSKEVER, I., AND HINTON, G. E. 2012. Imagenet classification with deep convolutional neural networks. In *Proc. of Advances in Neural Information Processing Systems*, 1097–1105.

LEE, J., AND LEE, K. H. 2004. Precomputing avatar behavior from human motion data. In *Proc. of 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 79–87.

LEE, J., AND SHIN, S. Y. 1999. A hierarchical approach to interactive motion editing for human-like figures. *SIGGRAPH'99*, 39–48.

LEE, J., CHAI, J., REITSMA, P. S., HODGINS, J. K., AND POLLARD, N. S. 2002. Interactive control of avatars animated with human motion data. *ACM Trans on Graph 21*, 3, 491–500.

LEE, Y., WAMPLER, K., BERNSTEIN, G., POPOVIĆ, J., AND POPOVIĆ, Z. 2010. Motion fields for interactive character locomotion. *ACM Trans on Graph 29*, 6, 138.

LEVINE, S., AND KOLTUN, V. 2014. Learning complex neural network policies with trajectory optimization. In *Proc. of the 31st International Conference on Machine Learning (ICML-14)*, 829–837.

LEVINE, S., WANG, J. M., HARAUX, A., POPOVIĆ, Z., AND KOLTUN, V. 2012. Continuous character control with low-dimensional embeddings. *ACM Trans on Graph 31*, 4, 28.

MIN, J., AND CHAI, J. 2012. Motion graphs++: a compact generative model for semantic motion analysis and synthesis. *ACM Trans on Graph 31*, 6, 153.

MIN, J., CHEN, Y.-L., AND CHAI, J. 2009. Interactive generation of human animation with deformable motion models. *ACM Trans on Graph 29*, 1, 9.

MITTELMAN, R., KUIPERS, B., SAVARESE, S., AND LEE, H. 2014. Structured recurrent temporal restricted boltzmann machines. In *Proc. of the 31st International Conference on Machine Learning (ICML-14)*, 1647–1655.

MORDATCH, I., LOWREY, K., ANDREW, G., POPOVIC, Z., AND TODOROV, E. 2015. Interactive control of diverse complex characters with neural networks. In *Proc. of Advances in Neural Information Processing Systems*.

MUKAI, T., AND KURIYAMA, S. 2005. Geostatistical motion interpolation. *ACM Trans on Graph 24*, 3, 1062–1070.

MÜLLER, M., RÖDER, T., CLAUSEN, M., EBERHARDT, B., KRÜGER, B., AND WEBER, A. 2007. Documentation mocap database hdm05. Tech. Rep. CG-2007-2, Universität Bonn, June.

NAIR, V., AND HINTON, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *Proc. of the 27th International Conference on Machine Learning (ICML-10)*, 807–814.

OFLI, F., CHAUDHRY, R., KURILLO, G., VIDAL, R., AND BAJCSY, R. 2013. Berkeley mhad: A comprehensive multimodal human action database. In *Applications of Computer Vision (WACV), 2013 IEEE Workshop on*, 53–60.

ROSE, C., COHEN, M. F., AND BODENHEIMER, B. 1998. Verbs and adverbs: Multidimensional motion interpolation. *IEEE Comput. Graph. Appl. 18*, 5, 32–40.

ROSE III, C. F., SLOAN, P.-P. J., AND COHEN, M. F. 2001. Artist-directed inverse-kinematics using radial basis function interpolation. *Computer Graphics Forum 20*, 3, 239–250.

SAFONOVA, A., AND HODGINS, J. K. 2007. Construction and optimal search of interpolated motion graphs. *ACM Trans on Graph 26*, 3, 106.

SHIN, H. J., AND OH, H. S. 2006. Fat graphs: constructing an interactive character with continuous controls. In *Proc. of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, Eurographics Association, 291–298.

SRIVASTAVA, N., HINTON, G., KRIZHEVSKY, A., SUTSKEVER, I., AND SALAKHUTDINOV, R. 2014. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research 15*, 1, 1929–1958.

TAN, J., GU, Y., LIU, C. K., AND TURK, G. 2014. Learning bicycle stunts. *ACM Trans on Graph 33*, 4, 50.

TAYLOR, G. W., AND HINTON, G. E. 2009. Factored conditional restricted boltzmann machines for modeling motion style. In *Proc. of the 26th International Conference on Machine Learning*, ACM, 1025–1032.

TAYLOR, G. W., HINTON, G. E., AND ROWEIS, S. T. 2011. Two distributed-state models for generating high-dimensional time series. *The Journal of Machine Learning Research 12*, 1025–1068.

VINCENT, P., LAROCHELLE, H., LAJOIE, I., BENGIO, Y., AND MANZAGOL, P.-A. 2010. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *The Journal of Machine Learning Research 11*, 3371–3408.

WANG, J., HERTZMANN, A., AND BLEI, D. M. 2005. Gaussian process dynamical models. In *Proc. of Advances in Neural Information Processing Systems*, 1441–1448.

XIA, S., WANG, C., CHAI, J., AND HODGINS, J. 2015. Realtime style transfer for unlabeled heterogeneous human motion. *ACM Trans on Graph 34*, 4, 119:1–119:10.

YAMANE, K., AND NAKAMURA, Y. 2003. Natural motion animation through constraining and deconstraining at will. *Visualization and Computer Graphics, IEEE Transactions on 9*, 3, 352–360.

ZEILER, M. D., AND FERGUS, R. 2014. Visualizing and understanding convolutional networks. In *Computer Vision–ECCV 2014*. Springer, 818–833.