A study of Motion-Data Compression In Motion-Matching Systems

Yineng Wang

March 2022

Abstract

In animation systems, data-drive methods have been studied to generate character motions. The learned motion matching algorithms is a variant of motion matching algorithms to generate character motion on the fly, exploiting autoencoder's ability to extract latent representation of motion data. In this report, we study one of the key stage of the learend motion matching algorithm, *Decompression*. We implement a *Compressor* network that compresses the pose data into a latent code, and a *Decompressor* network that reconstructs the pose data. We investigate the effect of using forward kinematics in training. We also study how the number of layers influence the compression ratio and reconstruction loss.

1 Introduction

In video games, there are demands for realistic character animations. Dynamically-generated animation clips, as an alternative to pre-recorded and replayed animations clips, provides more natural character motions. Motion matching is an animation-generation technique first proposed by Clavet and Büttner [4]. The technique searches a database of animations and find the one that best fits the context. It is able to produce animation on the fly, with a level of control, i.e., allows animators to specify the properties of animations. The primary limitation of this method is that it requires large amount of data. Therefore, the memory usage and run-time performance increases linearly with the amount of data [3].

To address the limitation, Holden et al. propose a learned motion matching model which retains the benefits of motion matching, but has scalability of neural-network-based models [3]. Learned motion matching presents state-of-the-art runtime performance and memory usage in animation generation.

The goal of this project is to replicate and evaluate one of the three key stages of the learned motion matching algorithm, *Decompression*. With a *Compressor*, large amount of character motion data represented in frame-by-frame joint data is compressed into a latent vector via an encoder network, and can be reconstructed via *Decompressor*, a decoder network, thus decreasing the amount of data stored in the matching database and the memory usage.

2 Background

This section introduces autoencoders that works as an compressor for motion data, as well as the pipeline of the basic motion matching algorithm, and each stage's correspondence in the learned motion matching algorithm.

2.1 Autoencoder

An autoencoder is a type of neural network used to extract representation (latent code) of data without any labels. Traditionally, the latent representations are in low-dimensional space, so autoencoders can be used for dimensionality reduction.

The basic architecture of an autoencoder consists of an encoder that maps the high dimensional latent code, and a decoder that produce a reconstruction of the input. A basic autoencoder can be formulated by:

$$\mathbf{h} = \text{encoder}(\mathbf{x}) \tag{1}$$

$$\mathbf{x}' = \text{decoder}(\mathbf{h}), \tag{2}$$

where $\mathbf{x}$ is the input vector, $\mathbf{h}$ is the latent code, and $\mathbf{x}'$ is the reconstructed input. The loss function is usually defined by the distance between the input vector and the reconstructed input vector, i.e.,

$$\mathcal{L} = \text{distance}(\mathbf{x}, \mathbf{x}'). \tag{3}$$

2.2 Pipeline of Basic Motion Matching

The learned motion matching model searches a matching database of animation for a best matched frame every N frames. In the matching database, for each frame, a key $\mathbf{x}$ and a pose vector $\mathbf{y}$ are defined. $\mathbf{x}$ is a feature vector appropriate for locomotion controller. The pose vector $\mathbf{y}$ contains all the pose information, like the local translations and rotations of all joints [3].

There are three key stages in the algorithm. In the the *Projection* stage, a query vector $\hat{\mathbf{x}}$ is computed for every N frames at runtime. It can be constructed by estimating future trajectory of the character at some frames ahead, and computing the current positions as well as velocities of key joints. The algorithm then searches the matching database for the feature vector $\mathbf{x}_{k^*}$ that has the minimal squared euclidean distance to the query vector:

$$k^* = \arg\min_k \|\hat{\mathbf{x}} - \mathbf{x}_k\|^2 \tag{4}$$

In the *Decompression* stage, the pose vector $\mathbf{y}_{k^*}$ associated with $\mathbf{x}_{k^*}$ is retrieved from the matching database.

Then the algorithms comes to the *Stepping* stage. For the next few frames, the animation index advances at each frame, and the associated pose vectors $\mathbf{y}_{k^*+1}$, $\mathbf{y}_{k^*+2}$, ..., are retrieved.

A transition of animation from current frame to the best matched frame is applied using inertilization blending [1].

2.3 Pipeline of Learned Motion Matching

Instead of storing the whole feature vector $\mathbf{x}$ and pose vector $\mathbf{y}$ in the matching database, the learned motion matching algorithm takes advantage of the scalability of neural-networks, and compute the vectors on the fly.

In the *Decompression* stage, the *Decompressor*, an decoder network is used. It takes the feature vector $\mathbf{x}$, combined with an additional latent feature vector $\mathbf{z}$, and produces the corresponding pose vector $\mathbf{y}$. To find the latent feature vector $\mathbf{z}$, an encoder network, *Compressor* is used. It takes $\mathbf{y}$ as input, and produces the latent code $\mathbf{z}$. Combining the *Compressor* and *Decompressor*, the algorithm pre-trains an autoencoder-like network. The *Decompressor* avoid storing $\mathbf{y}$ in the matching database, which accounts for the primary memory usage. When training the *Decompressor*, forward kinematics is used in the loss function, which will be discussed in section 3 and 4.1.

In the *Stepping* stage, another autoencoder-like network called *Stepper* is used. It takes $\mathbf{x}_i\mathbf{z}_i$ at the current frame, and produces $\mathbf{x}_{i+1}\mathbf{z}_{i+1}$. By using the *Stepper*, the algorithm produces a stream of feature and latent vectors, which significantly decreases the number of $\mathbf{x}$ and $\mathbf{z}$ stored in the matching database.

In the *Projection* stage, a *Projector* network is used to map the query vector $\hat{\mathbf{x}}$ to the nearest feature vector $\mathbf{x}^*$ and corresponding latent vector $\mathbf{z}^*$. It removes the need to store $\mathbf{x}$ and $\mathbf{z}$, and hence the need to search for the nearest neighbor in the matching database.

Note that the *Decompressor* must be trained first so the latent vector $\mathbf{z}$ can be computed and used by the *Stepper* and the *Projector*.

3 Data Acquisition and Pre-processing

This section discusses how the dataset used for training is acquired and pre-processed.

3.1 Dataset

This project uses the Ubisoft La Forge Animation dataset[1] [2]. The dataset contains 5 subjects, 77 sequences, and 496,672 motion frames at 30fps. The animation sequences are in the BVH file format, which stores the positions and rotations of joints for each frame, and provides information on skeleton hierarchy. Therefore, we can use forward kinematics to compute more pose information besides global positions and local joint rotations. In this project, 3 sequences of total 53,500 frames, containing idle, walking and running motions are used for training.

[1] `https://github.com/ubisoft/ubisoft-laforge-animation-dataset`

3.2 Data Pre-processing

After the position and rotation values are extracted from the BVH file, feature vectors and pose vectors are computed by using forward kinematics for each frame. The feature vectors and pose vectors are defined in a similar way to Holden [3]. In this project, the feature vectors are defined as $\mathbf{x} = \{\mathbf{t}^t\ \mathbf{t}^d\ \mathbf{f}^t\ \dot{\mathbf{f}}^t\}$, where $\mathbf{t}^t$ are the 2D future trajectory positions on the ground, looking 20, 40, and 60 frames ahead, $\mathbf{t}^d$ are the future trajectory facing directions in 20, 40, and 60 future frames, local to the character, $\mathbf{f}^t$ are the left and right foot local positions, $\dot{\mathbf{f}}^t$ are the left and right foot local velocities. The pose vectors are defined as $\mathbf{y} = \{\mathbf{y}^t\ \mathbf{y}^r\ \dot{\mathbf{y}}^t\ \dot{\mathbf{y}}^r\ \dot{\mathbf{r}}^t\ \dot{\mathbf{r}}^r\}$, where $\mathbf{y}^t$ is the local joint positions, $\mathbf{y}^r$ is the local joint rotations, $\dot{\mathbf{y}}^t$ is the local joint velocities, $\dot{\mathbf{y}}^r$ is the local joint angular velocities, $\dot{\mathbf{r}}^t$ is the character root velocities, and $\dot{\mathbf{r}}^r$ is the character root angular velocities. Each feature is normalized by its mean value and standard deviation in the dataset.

For the same reason as in QuaterNet by Pavllo [5] [6], the rotations are represented by quaternions instead of Euler angles, because quaternions are free of discontinuities and singularities. An extra step removing discontinuities of data is taken, because a quaternion $\mathbf{q}$ and its antipodal representation $-\mathbf{q}$ represent the same orientation [5].

4 Network Architecture and Training

4.1 Network Architecture

This project aims to implement the *Compressor* and *Decompressor* network. The network architecture is presented in Figure 1.

Each frame is associated with a feature vector $\mathbf{x}$ and a pose vector $\mathbf{y}$. We first compute the forward kinematics of $\mathbf{y}$, which yields the pose vector $\mathbf{q}$. The *Compressor* takes both $\mathbf{y}$ and $\mathbf{q}$, and produces the latent vector $\mathbf{z}$. Then $\mathbf{z}$ is feed into the *Decompressor* together with the $\mathbf{x}$, and reconstructs the pose vector $\mathbf{y}'$. From $\mathbf{y}'$, $\mathbf{q}'$ is computed using forward kinematics. The introduction of the $\mathbf{q}$ vector allow the loss function to take locomotion information into consideration, which is discussed in the next section.

4.2 Loss Function and Regularization Term

To avoid jittery and low quality motions, Holden proposed a loss function which takes kinematics into consideration [3]. It is defined as

$$\mathcal{L}_{loss} = \|\mathbf{y} \ominus \mathbf{y}'\| + \|\mathbf{q} \ominus \mathbf{q}'\| + \left\| \frac{\mathbf{y}_0 - \mathbf{y}_1}{\delta t} - \frac{\mathbf{y}'_0 - \mathbf{y}'_1}{\delta t} \right\| + \left\| \frac{\mathbf{q}_0 - \mathbf{q}_1}{\delta t} - \frac{\mathbf{q}'_0 - \mathbf{q}'_1}{\delta t} \right\| \quad (5)$$

where the $\ominus$ operator is defined as basic element-wise subtractions. the subscriptions 0 and 1 also denotes vectors of two consecutive frames in the mini-batch, and δt is the reciprocal of the FPS.

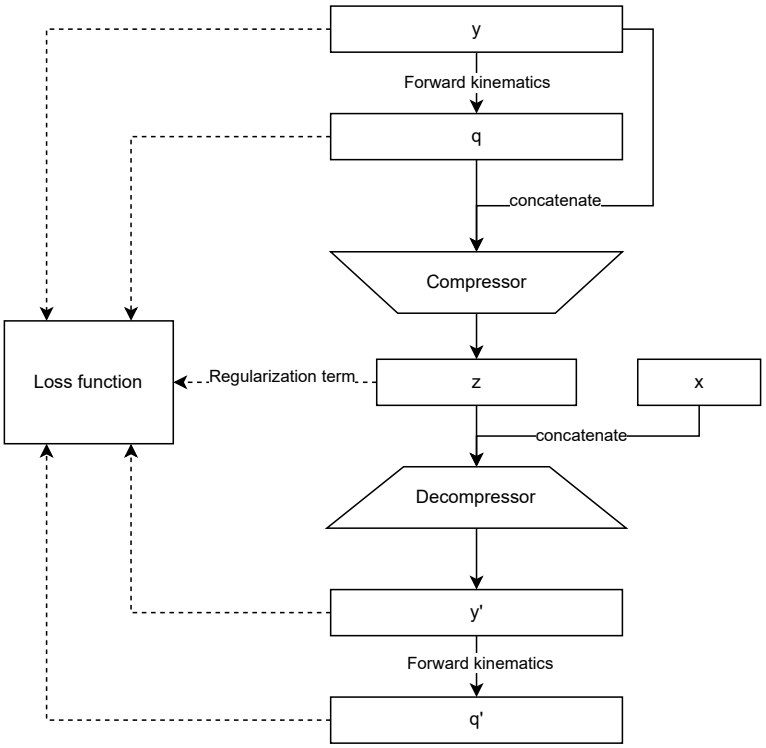


Figure 1: The network architecture of *Decompressor* and *Compressor*. They together form an autoencoder-like network.

This loss function measures the error in positions as well as inter-frame velocities, ensuring the output changes smoothly frame by frame.

The regularization term is defined as

$$\mathcal{L}_{reg} = \|\mathbf{z}\|_2^2 + \|\mathbf{z}\|_1 + \left\| \frac{\mathbf{z}_1 - \mathbf{z}_0}{\delta t} \right\| \tag{6}$$

where $\mathbf{z}_1$ and $\mathbf{z}_0$ represents latent vectors of two consecutive frames in the mini-batch.

5 Experiments

The section shows the key hyperparameter settings in training and the details of experiments.

5.1 Training

The *Compressor* and *Decompressor* are trained in PyTorch using the AdamW optimizer, with a learning rate of 1e-3 and weight decay of 1e-3. The learning

rate is decayed by 0.99 each epoch. The size of mini-batches is set to 32. The network is trained for 8,000 epochs.

5.2 Enabling/Disabling Forward Kinematics

The introduction of forward kinematics is one of the key factors that enables the learned motion matching algorithm to extract latent motion representations [3]. To investigate the impact of forward kinematics, an experiment is conducted with forward kinematics disabled. This also removes the $\mathbf{q}$ terms in the loss function, thus decreasing the overall loss, but could produce worse reconstruction.

5.3 Number of Layers in the Network

In Holden's implementation, they use 5 layers with 512 hidden units in each of the hidden layers for *Compressor*, and 3 layers with 512 hidden units for *Decompressor* (the input layer and the output layer are included). Increasing the number of layers would possibly decrease the loss, meanwhile increasing the model size (number of parameters). An experiment is conducted to investigate how the number of layers impact the performance and memory usage, in which a combination of (5, 3), (5, 4), (5, 5), (4, 3), (4, 4), (4, 5), (3, 3), (3, 4), (3, 5) layers in the *Compressor* and *Decompressor* are evaluated.

6 Results

Figure 2 shows an example of visualization of the *Decompressor*'s reconstructions of motion data with forward kinematics enabled/disabled, in comparison with their original pose.

Table 1 shows the effect of the number of layers in the *Compressor* and *Decompressor* on networks' reconstruction loss, as well as compression ratio. The compression ratio is computed as

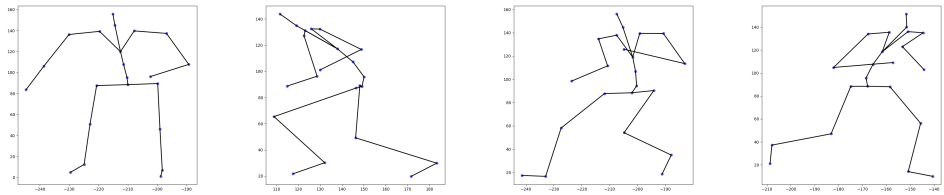
$$\frac{(\text{size}(\mathbf{x}) + \text{size}(\mathbf{y})) \cdot \#\text{frames}}{\text{ModelSize} + (\text{size}(\mathbf{x}) + \text{size}(\mathbf{z})) \cdot \#\text{frames}}$$

where the model size is computed as

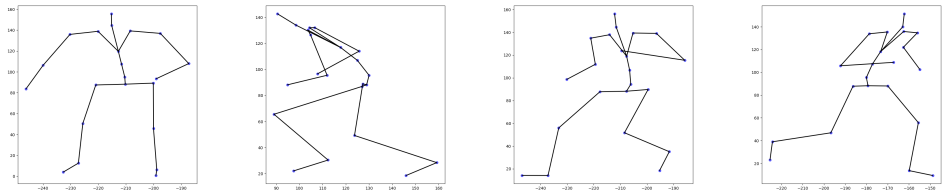
$$\text{size}(\textit{Compressor}.\text{parameters}) + \text{size}(\textit{Decompressor}.\text{parameters})$$

7 Conclusion and Future Work

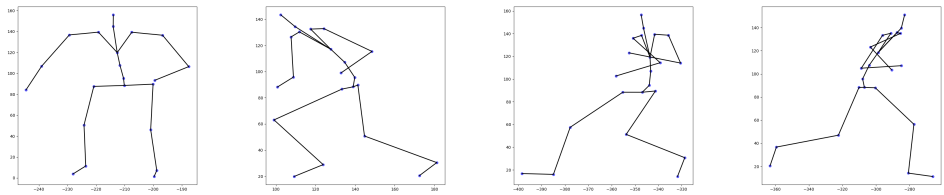
As Figure 2 shows, enabling or disabling forward kinematics does not have significant influence on the reconstruction of the original motion data. This indicates y is probably sufficient for the training of the *Decompressor* network, especially in terms of loss function.



(a) original (b) original (c) original (d) original



(a) reconstructed (FK enabled) (b) reconstructed (FK enabled) (c) reconstructed (FK enabled) (c) reconstructed (FK enabled)



(a) reconstructed (FK disabled) (b) reconstructed (FK disabled) (c) reconstructed (FK disabled) (c) reconstructed (FK disabled)

Figure 2: Original motion compared with corresponding reconstructed motion using the *Decompressor*, with forward kinematics enabled/disabled

As Table 1 shows, increasing the number of layers in the network does not necessarily improve the network's performance in reconstruction. By using fewer layers, we can increase the compression ratio, but the improvement is marginal, considering the fact that in real-life applications, it is the motion data that accounts for the vast majority of storage.

This project only replicates and evaluates one stage of the learned motion matching algorithm, the *Compressor* and *Decompressor* network. To fully utilize the ability of autoencoder networks to represent data in low-dimensional space, the other two networks, Stepper and Projector, also need to be implemented.

Moreover, the algorithm has not been integrated into a real-life animation system. At first I tried using Unreal Engine 4, but its built-in animation system has a preference for high-level animation representations, rather than operating

Table 1: The effect of the number of layers in the *Compressor* and *Decompressor* on networks' reconstruction loss, as well as compression ratio

# layers (Compressor)	# layers (Deompressor)	Loss at the end of training	Model size (MB)	Compression ratio
5	3	2.698	4.353	4.261
5	4	2.734	5.404	4.011
5	5	3.201	6.454	3.791
4	3	2.696	3.303	4.541
4	4	2.695	4.353	4.260
4	5	3.114	5.404	4.181
3	3	2.727	2.252	**4.862**
3	4	**2.667**	3.303	4.541
3	5	2.963	4.355	4.261

bones directly. The algorithm needs to be further tested in terms of memory usage and runtime performance on a real-life animation system.

References

[1] Kevin Bergamin et al. "DReCon: Data-Driven Responsive Control of Physics-Based Characters". In: *ACM Trans. Graph.* 38.6 (Nov. 2019). ISSN: 0730-0301. DOI: 10.1145/3355089.3356536. URL: https://doi.org/10.1145/3355089.3356536.

[2] Félix G. Harvey et al. "Robust Motion In-Betweening". In: 39.4 (2020).

[3] Daniel Holden et al. "Learned motion matching". In: *ACM Trans. Graph.* 39.4 (2020), p. 53. DOI: 10.1145/3386569.3392440. URL: https://doi.org/10.1145/3386569.3392440.

[4] *Motion Matching - The Road to Next Gen Animation*. https://www.youtube.com/watch?v=z_wpgHFSWss&t=658s. Accessed: 2022-01-12.

[5] Dario Pavllo, David Grangier, and Michael Auli. *QuaterNet: A Quaternion-based Recurrent Model for Human Motion*. 2018. arXiv: 1805.06485 [cs.CV].

[6] Dario Pavllo et al. "Modeling Human Motion with Quaternion-Based Neural Networks". In: *International Journal of Computer Vision* 128.4 (Oct. 2019), pp. 855–872. ISSN: 1573-1405. DOI: 10.1007/s11263-019-01245-6. URL: http://dx.doi.org/10.1007/s11263-019-01245-6.