Computers & Graphics 120 (2024) 103911

Contents lists available at ScienceDirect

Computers & Graphics

journal homepage: www.elsevier.com/locate/cag

Special Section on MIG 2023

# Making motion matching stable and fast with Lipschitz-continuous neural networks and Sparse Mixture of Experts

Tobias Kleanthous [a,*], Antonio Martini [b]

[a] Production R&D, Tencent, Edinburgh, UK
[b] Production R&D, Tencent, London, UK

ARTICLE INFO

*Keywords:*
Character animation
Animation
Neural networks
Motion matching
Regularization

ABSTRACT

Motion matching has become a widely adopted technique for generating high-quality interactive animation systems in video games. However, its current implementations suffer from significant computational and memory resource overheads, limiting its scalability in the context of modern video game performance profiles.

Our method significantly reduces the computational complexity of approaches to motion synthesis, such as "Learned Motion Matching", while simultaneously improving the compactness of the data that can be stored and the robustness of pose output. As a result, our method enables the efficient execution of motion matching that significantly outperforms other implementations, by 8.5× times in CPU execution cost and at 80% of the memory requirements of "Learned Motion Matching", on contemporary video game hardware, thereby enhancing its practical applicability and scalability in the gaming industry and unlocking the ability to apply on large numbers of animated in-game characters.

In this paper, we expand upon our published paper "Learning Robust and Scalable Motion with Lipschitz Continuity and Sparse Mixture of Experts", where we successfully proposed a novel method for learning motion matching that combines a Sparse Mixture of Experts model architecture and a Lipschitz-continuous latent space for representation of poses. We present further details on our method, with extensions to our approach to expert utilization within our neural networks.

## 1. Introduction

Large-scale *"AAA"* games regularly push limits in seeking to create ever richer, more characterful and naturalistic performances to entertain and immerse their players. The ongoing increases in complexity of character animation systems in modern video games has been well documented in an escalating battle for hardware resources, such as CPU and memory. In recent years, video games developers have embraced a wide-range of techniques to deliver hundreds of styles across thousands of animations when striving to capture the imaginations of players at hitherto unprecedented scales, but not without substantial costs in terms of authoring complexity and production [1].

Research in the area of interactive character animation has sought to manage the complexity of implementations, whilst extending the expressiveness available to games developers, from parametric animation graphs, to *motion fields* [2] and the increasingly popular approach of *motion matching* [3,4], before the now growing body of research in applying machine learning techniques to animated character control [5,6].

In this paper, we build upon the successful foundations of both motion matching and machine learning approaches to motion synthesis, to improve scalability and robustness of these techniques. Our method innovatively employs the Sparse Mixture of Experts method and Lipschitz continuous neural networks in real-time animation systems for the first time. Furthermore, we demonstrate significant improvements in runtime performance metrics and robustness of output. As a result, our method enables high-quality animation systems executed by neural networks, that outperform traditional methods used in the video games industry.

## 2. Related work

In this section, we highlight related works covering animation control, limitations concerning compute complexity of machine learning models and their execution, and approaches to smooth representations of data in *neural networks (NNs)*.

### 2.1. Motion matching and learned motion matching

Motion matching has gained popularity for its expressive control, whilst enabling users to move away from the laborious manual creation

* Corresponding author.
*E-mail addresses:* tobiask@global.tencent.com (T. Kleanthous), antonio@global.tencent.com (A. Martini).

https://doi.org/10.1016/j.cag.2024.103911
Received 19 January 2024; Received in revised form 19 March 2024; Accepted 22 March 2024
Available online 26 March 2024

and maintenance of stately graphs to control the flow and blend of many animations simultaneously [3,4]. The core principles of a system require analysing some mixture of user driven control parameters and internally derived parameters stored in a database that can uniquely identify desirable transitions at runtime to assist in striking a balance between continuity and satisfying goals for a given action. Commonly, features such as future and historic trajectories, alongside pose driven features, such as model space velocities of joints, are used.

A simplified example of executing motion matching for a locomotion system may be implemented as follows. Prior to runtime, features are extracted from each individual frame of the source animation data, such as those in Table 10. These features are then stored as an N-dimensional vector in some form of accelerated nearest neighbour structure. Each frame of execution within a game, user input is provided to modify the requirements of some of these features, whilst preserving others that represent continuity from an existing example frame actively playing back. An example could be modify the desired trajectory for some time in the future in the features, but preserving those features derived from the pose, such as joint velocities, to ensure some pressure on preserving continuity in posing. The accelerated nearest neighbour structure is then queried for the nearest n-samples, if these produce a lower cost than currently playing sample, for example using a weighted $L2$ distance metric, a transition is requested and the systems begins executing from the nearly selected animation data.

Whilst the qualitative output, simplification of expression complex state-switching and control of motion achievable is highly desirable, a substantial requirement for additional data storage and CPU cost is incurred when deploying motion matching. *Learned Motion Matching (LMM)* [7] addressed issues concerning memory requirements by reducing the database and animations instead to a collection of smaller *feedforward networks (FFNs)*, however, applicability was limited by the needs to balance a reduction of memory with an increase in CPU costs, an already heavily contested resource in optimizing large-scale video games. Other methods to improve runtime performance such as GPU based execution [8] have proven effective, but make trade offs with latency of control, that prevent responsive player control of characters, and again, are in contention with a resource heavily used in accelerating high-quality rendering.

In this work, we show that the promise of a robust, and comparatively cost-effective execution of such techniques via *NNs* is possible to achieve in a manner that excels beyond prior methods.

### 2.2. Compute complexity of neural network inference on local hardware

The main performance limitation of using feedforward networks *(FFN)* for *LMM* [7], stems from the fact that, as the motion database increases in size, it is necessary to increase *FFN* depth and width in order to preserve motion reproduction accuracy, resulting in a drastic increase in the number of model weights which leads to runtime evaluation times beyond production budget (See also Section 8 in [7]). This problem was also recognized in [9], which attempted to generate a sparsified Mixture of Experts*(MoE)* by post-processing the model weights with a pruning scheme. This weights sparsification approach however suffers from severe limitations as increases in model performance are directly correlated to decreased motion reproduction accuracy, resulting either in visual artifacts or in limited performance gains. An additional problem noted in the same work, is that as sparse matrix–vector operations are not supported in hardware, the theoretical gains do not necessarily translate to faster execution times.

Inspired by recent advances in the field of large language models and conditional computation [10–12], in this work we show for the first time that, with some small modifications and additional training considerations, *SMoE* type layers applied to motion synthesis result in highly accurate and performant architectures, without requiring any post training simplifications.

Further to our work in [13], we present modifications to our method that improve upon the utilization of experts within our models in Section 5.1.

### 2.3. Smooth latent manifolds via Lipschitz continuous neural networks

We note, that components of animations retain temporospatial relationships that express smoothness in their raw form, which other methods seek to exploit [14]. By maximizing the preservation of this relationship in a latent manifold, we seek to induce an intuitive mapping that can be further exploited to improve utilization and quality. Lipschitz continuity in neural networks provides an interesting avenue of research when seeking to enforce this relationship.

A function $f$ is defined as Lipschitz continuous when there is some constant $c$, where $c$ is the absolute sum of gradients across the range of values expressed by $x$ and $y$ (see Section 7 for further detail), satisfies the following:

$$\|f(x) - f(y)\| \leq c \cdot \|x - y\| \tag{1}$$

Applied to *NNs*, enforcing Lipschitz continuity allows us to ensure that the similarity of a given pair of inputs can lead to a direct relationship with the clear similarity of the pair's respective *NN* outputs. In our method, this means in terms of maximizing expressable continuity of consecutive frames of animation between the source pose space and their latent form. This can be enforced by learning minimized weights of layers within some learned bounds [15,16]. The benefits are twofold, a more robust and better performing model can be learned with such an approach, that also more readily generalizes across unseen data [17]. This is an intriguing proposition in the context of animation in machine learning, where common approaches have relied on a more compact encoding of the animation data as a working representation. Expressing a close relative relationship between animation data and their latent representation presents a benefit for the predictability of their usage.

Various techniques, such as *variational autoencoders (VAEs)* [18], have been applied to address smoothness and continuity when modelling animation tasks, with mixed results, often suffering from over smoothing. *Periodic autoencoders (PAEs)* [14] that learn approximations of temporospatial relationships which can be used to align animation representations, have shown great success, but do not fully imbue that ability within a single model, instead solving a more easily learned relationship that can be subsequently used in other models such as *phase functioned neural networks (PFNNs)* [5].

Influenced by [16] applying Lipschitz continuous *NNs* for geometric interpolation, we show in this work, that a Lipschitz continuous encoder, applied to animation data and techniques seen in *LMM* [7], can improve the quality of results, not only within the decoded latent pose representation, but the predictive components that leverage the encoder's latent output, by better representing the original relationships of animation frames in the latent manifold.

We further detail the reasoning behind our decisions from [13] to leverage Lipschitz continuity in Section 7.

## 3. Core concepts

In this section, we discuss the core details of our method and our application of *SMoE* and Lipschitz continuity to *NNs* for animation control.

### 3.1. Sparse mixture of experts (SMoE)

We employ a SMoE layer architecture with top-2 routing [10]. As we are not dealing with discrete tokens like in NLP applications, and in order to avoid any form of motion jitter, we avoid the probabilistic dropping of the second expert as typically done in existing work [10, 12]. As illustrated in Fig. 2, a gating layer takes $x$ as input and routes it to the best matched top-2 experts selected out of a set $\{E_i(x)\}_{i=1}^{N}$ of $N$ experts. The gating logits $h(x) = LINEAR(x)$ are calculated by passing the input through a linear layer, the final gate values are then obtained

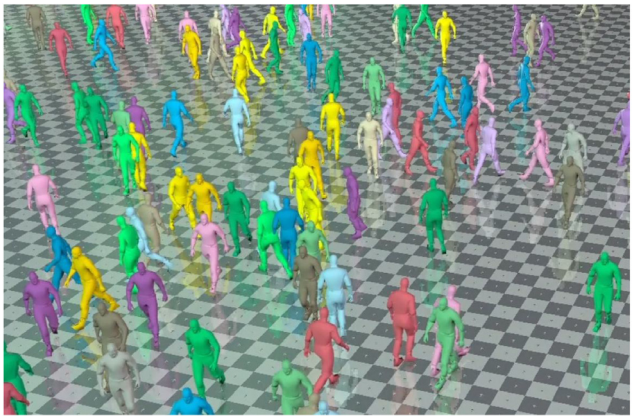


**Fig. 1.** Our method enables evaluating 100 s of characters concurrently within typical video game animation CPU budgets.

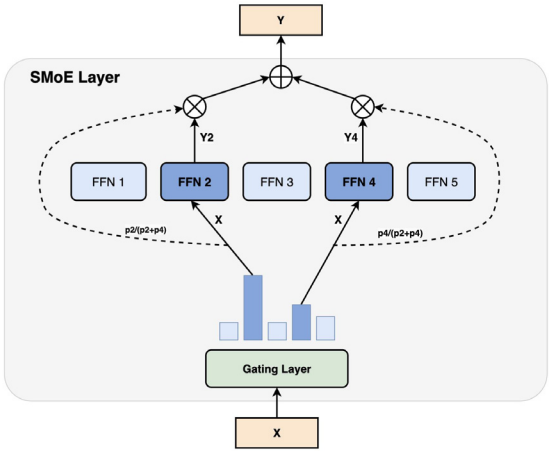



**Fig. 2.** Example of activations in a SMoE layer with 5 experts and top-2 gating.

via a softmax operation which results in a probability distribution over the $N$ experts

$$p_i(x) = \frac{e^{h(x)_i}}{\sum_j^N e^{h(x)_j}} \tag{2}$$

The input $x$ is then routed to the experts with the top-2 gate values. The final results is computed as the weighted sum of the top-2 active experts output weighted by their associated normalized gate values:

$$y = \frac{p_{1st}(x)}{p_{1st}(x) + p_{2nd}(x)} E_{1st}(x) + \frac{p_{2nd}(x)}{p_{1st}(x) + p_{2nd}(x)} E_{2nd}(x) \tag{3}$$

Where $1st$ and $2nd$ refer to the indices of respectively the highest and second highest gate values in Eq. (2). Experts $E_i(x)$ are feed-forward networks(FFN) sharing the same architecture and employing *GELU* [19] activation functions, which we have found to provide superior performance to ELU, ReLU and LeakyReLU activations. A *SMoE* architecture can be implemented as a single layer which can then be integrated as a part of a larger model. Further, it is worth noting that single experts have no predefined interpretable meaning, how data is routed to experts and what each single expert learns is automatically determined during training.

### 3.2. Lipschitz layers for smooth continuity in latent space

We implement a Lipschitz linear layer as in section 4.1.1 of [16]. A learnable Lipschitz bound $c$ is used to normalize the weight matrix $W$ with rows $R$, using scaling vector created by the function $Scale$.

$$Scale(c, W) = Min(\frac{softplus(c)}{Max(\sum_{r=1}^{R} |W_r|, \epsilon)}, 1.0) \tag{4}$$

$$\epsilon = 1 \times 10^{-7} \tag{5}$$

$$WeightNormalization(W) = W \cdot Scale(c, W) \tag{6}$$

This weight normalization is then applied when treating the Lipschitz layer interchangeably with a linear layer in a model, with bias $b$.

$$LipschitzLinear(X) = X * WeightNormalization(W) + b \tag{7}$$

For initialization, for $W$ we use the same Kaiming initialization [20] used in *PyTorch's* [21] linear layer implementation and then initialize $c$ by taking the $L\infty$-norm of $W$. At runtime, $c$ remains constant and the normalization of the weight matrices within a model can thus be evaluated as constant when preparing the model for inference.

## 4. Architecture

In this section, we highlight the architecture of our method and models.

### 4.1. Pose representation

To represent a single pose, we combine a hierarchy of $J$ joints and a pair of linear and angular root motion velocities.

Individual joints are formed of a position and rotation. We use positions and root motion velocities of the form $P \in \mathbb{R}^3$. For rotations, after experimentation across Euler angles and quaternions, as [22], we decided on 6D representations due to their superior ability to better express continuity in a neural network [23,24], we denote as $R \in \mathbb{R}^6$.

Poses are assessed during training both in local and model (or character) spaces, where model space is the result of a forward kinematics operator $FK(P, R)$ on the pose hierarchy.

### 4.2. Model architecture

We structure our method similarly as seen in *LMM* [7], in this section, we detail differences according to the unique approaches in our method.

#### 4.2.1. Pose autoencoder

Our pose autoencoder structure follows similar patterns in related work [7,18], we train the complete encoder and decoder pairing, with the ability to discard the encoder component at runtime. The inputs to our model is the concatenation of the pose in local space $X$, the model space evaluation after applying forward kinematics $FK(X)$ and any applicable tagging, such as foot contacts $C \in 0.0, 1.0$. The output is our pose prediction in local space $\hat{X}$, including root motion velocities, and any probabilities for any associated tagging per pose $\hat{C}$.

We form an encoder with several linear layers, each with a learnable Lipschitz constant (see Section 3.2). As per [7], the resultant latent encoding $z$ is then concatenated with standardized motion matching features $M$, extracted from the training data. Finally, we feed the components into our decoder, where we use the *SMoE* architecture to distribute across 128 significantly smaller experts prior to reforming the original pose dimensions in 2 following linear layers (see Fig. 3 and Encoder, Decoder and *SMoE* structures in Tables 1–3 respectively). Lastly, we perform sigmoid activation on the final 2 components of the pose output to recover probabilities of respective foot contact.

In experimentation, we found applying the Lipschitz layers in our decoder to produce a degradation in the quality of our final results, with little impact on the regularization observed in the pose latent. We believe this is, in part, due to the lack of enforcement of smoothness on all components of the concatenated motion matching features, and as

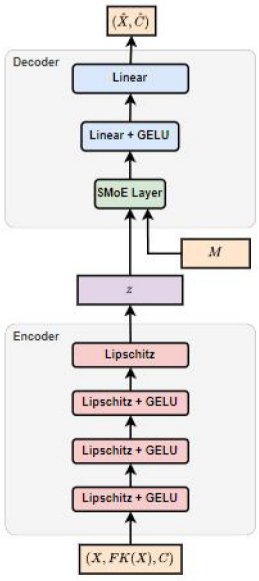



**Fig. 3.** Example flow of our pose autoencoder.

**Table 1**
Layers in our Pose Encoder.

| Layer | Units |
|---|---|
| Lipschitz + GELU | 512 |
| Lipschitz + GELU | 512 |
| Lipschitz + GELU | 512 |
| Lipschitz | 32 |

**Table 2**
Layers in our Pose Decoder.

| Layer | Units/Count |
|---|---|
| Linear + Softmax | 128 |
| Experts | 128 |
| Linear + GELU | 256 |
| Linear | 872 |

**Table 3**
Layers in a single Pose Decoder Expert.

| Layer | Units |
|---|---|
| Linear + GELU | 32 |
| Linear + GELU | 32 |
| Linear + GELU | 32 |
| Linear | 32 |

a result, applying a Lipschitz bound on the decoder negatively impacts the ability for the weights to learn usefully from this additional input.

Whilst we experimented with a larger latent, eschewing the provision of the motion matching features as input, we did not sufficiently solve the need for recovery of those features for runtime evaluation elsewhere in the system. We propose further investigation to improve on this in the future to ensure full Lipschitz continuity of all components of the model.

#### 4.2.2. Stepper

As in [7], we produce a stepper model that is autoregressively trained to predict output deltas of the latent $z$ and motion matching features $M$, $\delta z$ and $\delta M$ respectively. The stepper model replaces the point in a traditional animation system or motion matching implementation where samples of an animation on a future frame are taken, instead, the model predicts what the values will be based on a prior frame's output latent and matching feature values.

**Table 4**
Layers in SMoE Stepper.

| Layer | Units/Count |
|---|---|
| Linear + Softmax | 64 |
| Experts | 64 |

**Table 5**
Layers in a single stepper expert.

| Layer | Units/Count |
|---|---|
| Linear + GELU | 32 |
| Linear + GELU | 32 |
| Linear | 73 |

**Table 6**
Layers in SMoE Projector.

| Layer | Units/Count |
|---|---|
| Linear + Softmax | 64 |
| Experts | 64 |

Each step is equivalent to a timestep of

$$\delta t = \frac{1}{fps} \tag{8}$$

where fps, or frames per second, is 60 Hz in our data. Thus, training starts at sampled frame of the concatenated inputs $(z_0, M_0)$ and is then predicted autoregressively on some number of $N$ predictions up to $(z_n, M_n)$, in our case, for $N = 30$ further updates.

$$\delta\hat{z}_n, \delta\hat{M}_n = Stepper(z_n, M_n) \tag{9}$$

$$\hat{z}_{n+1} = \hat{z}_n + \delta\hat{z}_n \tag{10}$$

$$\hat{M}_{n+1} = \hat{M}_n + \delta\hat{M}_n \tag{11}$$

Consequently, we are motivated as in [16], to ensure our latent representation of a pose is smooth such that we may sub-sample on intervals where

$$\delta t < \frac{1}{fps} \tag{12}$$

thereby, being able to smoothly sample in-between frames, as we do when sampling the original source animation. We cumulatively gather losses for the predictions and velocity similarities over $N$ and then take the mean. In all cases, we use a mean squared error $MSE$, where the predicted latent and predicted latent delta are weighted twice that of the equivalent for motion matching features.

Our implementation is entirely based on a *SMoE* layer, detailed in Tables 4 and 5. Furthermore, we induce a robustness to runtime error by inducing noise at training time, as detailed in Section 5.9.

#### 4.2.3. Projector

We implement a *projector* model as in [7], whose purpose is to provide a replacement for the KD-tree (or other spatial partitioning system) as seen in *motion matching* [4]. Given a weighted motion matching feature query $M_{weighted}$, the nearest corresponding latent and motion matching features from the source, $z$ and $\hat{M}$ are returned.

$$z, \hat{M} = Projector(M_{weighted}) \tag{13}$$

These results can then be feed into the *stepper* and *decoder* models for their respective operations at runtime.

As with our *stepper* implementation, we form our projector entirely as a *SMoE*, as detailed in Tables 6 and 7. Further information on training this model is detailed in Section 5.6.

## 5. Training

We provide further implementations of our training in this section. Our models and training are implemented using *PyTorch* [21]. All

**Table 7**
Layers in Single Projector Expert.

| Layer | Units/Count |
|---|---|
| Linear + GELU | 64 |
| Linear + GELU | 64 |
| Linear + GELU | 64 |
| Linear + GELU | 64 |
| Linear | 73 |

**Table 8**
Example of gating layer activations with three inputs and two experts.

| Input | Expert1 | Expert2 |
|---|---|---|
| x1 | 0.49 | 0.51 |
| x2 | 0.32 | 0.68 |
| x3 | 0.9 | 0.1 |

training was performed across internal compute servers utilizing *Nvidia V100* GPUs.

Owing to the nature of the implementation using multiple models, we are able to train the stepper and projector models concurrently after training has completed on the pose autoencoder.

For all models, we train using the Adan optimizer [25], which we found to have a significant accuracy and performance advantage over commonly used alternatives. We set a learning rate $lr = 0.02$ with no weight-decay, and used a simple learning rate scheduler that decreased by factor $\gamma = 0.99$ for every 3 virtual epochs where the weighted moving average of overall reported losses do not reduce.

In practice, our models completed training to a desired level on assessed metrics before fully converging, thus, given an open-ended number of steps and compute time, we would expect to see further improvements in the quality of results.

At the time of writing, a full set of models can be trained with sub-centimetre accuracy and be used in-game in 9 h.

### 5.1. SMoE expert utilization

In order to ensure efficient experts utilization, we add an auxiliary load balancing loss term during training. We have experimented with two different losses. For the first loss [13], we followed the approach of [11] which is summarized below for convenience.

Given $N$ experts indexed by $i = 1$ to $N$ and a batch $\mathcal{B}$ with $T$ batch entries, the auxiliary loss is computed as the scaled dot-product between vectors $f$ and $P$,

$$\text{LoadBalancingLoss} = \alpha \cdot N \cdot \sum_{i=1}^{N} f_i \cdot P_i \tag{14}$$

where $\alpha$ is the loss relative weight within the overall model cost function and it is set to 0.01 for all models, $f_i$ is the fraction of batch entries dispatched to expert $i$,

$$f_i = \frac{1}{T} \sum_{x \in \mathcal{B}} \mathbb{1}\{\text{argmax}\, p(x) = i\} \tag{15}$$

and $P_i$ is the fraction of the gating layer probability allocated for expert $i$,

$$P_i = \frac{1}{T} \sum_{x \in \mathcal{B}} p_i(x). \tag{16}$$

It is claimed in [11] that the auxiliary loss of Eq. (14) encourages uniform routing of the batch entries since it is minimized under a uniform distribution, this would occur when both vectors $f_i$ and $P_i$ have values of $1/N$. See also [11] for additional details. While this loss performed satisfactorily in practice [13], we note that loss values smaller than the claimed ones can be achieved under non uniform distributions. For example, if we consider the configuration of 3 inputs and 2 experts detailed in Table 8, we obtain $f = (1/3, 2/3)$, $P = (0.57, 0.43)$ and $< f, P >= 0.476$, which is smaller than the statement minimum of $1/2$.

We have instead opted for the soft constraint loss described in [26], which produced better overall experts utilization while being theoretically sound. This loss consists of the Kullback–Leibler divergence (*KL-divergence*) between the estimated probability of experts activation and a uniform distribution $Q = 1/N$:

$$\begin{aligned} \mathcal{L}_{KL} &= w_{KL} \cdot D_{KL}(P \parallel Q) \\ &= w_{KL} \cdot \sum_{i=1}^{N} P_i \cdot \ln(P_i * N) \end{aligned} \tag{17}$$

Where $P_i$ is estimated by averaging over batch activations as detailed in Eq. (16).

Similarly to what suggested in [10], we have found top-2 routing to be a good compromise between model accuracy and run-time performance. However, contrarily to what observed in past work on large language models [10], where larger batch sizes result in model deterioration, we have instead found that for motion data, larger batch sizes consistently results in more efficient and accurate models.

### 5.2. Pose losses in autoencoder training

When evaluating poses during training of the autoencoder, we consider the weighting per joint and evaluate the results both in local and model spaces. As noted in Section 5.3, the impact of error in local space produces a larger proportional error on child joints further up the pose hierarchy, so local space accuracy is of some importance to guiding our models towards producing more accurate results. We create a pose representation for a number of joints $J$ output from our autoencoder $\hat{P}$ and compare to ground truth poses in our training set $P$. For positional and root motion velocity losses, we evaluate across batch $B$, with per-joint weights $W$ using an $L1$ loss per joint. We found this discouraged an averaging effect over reconstruction of positions compared to $L2$

$$PositionalLoss(P, \hat{P}, W) = \frac{1}{\mathcal{J}} \sum_{j=1}^{\mathcal{J}} W_j \cdot \|P_j - \hat{P}_j\|_1 \tag{18}$$

For rotational losses, we take the two normalized orthogonal vector components of the 6D rotational representation, $Rx$ and $Ry$, and minimize the predicted rotation from ground truth. We take the same approach as in [27], forming rotation matrices from our ground truth and predicted 6D rotations, $M$ and $\hat{M}$, respectively. In the interest of making compatible with balancing with our normalization and positional losses, we normalize the result by $2\pi$.

$$GeodesicDelta(M, \hat{M}) = \arccos\left(\frac{tr(M\hat{M}) - 1}{2}\right) \tag{19}$$

$$AngleLoss(R, \hat{R}) = \frac{GeodesicDelta(Matrix(R), Matrix(\hat{R}))}{2\pi} \tag{20}$$

We found this to produce better continuity of smooth rotations over time when assessing our results compared to using an $L1$ or $L2$ distance metric on the components that make up our rotation representation. We see further improvements by integrating with a normalization loss on the components in local space.

$$NormLoss(\hat{R}) = |1.0 - \|\hat{R}x\|_2| + |1.0 - \|\hat{R}y\|_2| \tag{21}$$

$$6DLoss(R, \hat{R}) = AngleLoss(R, \hat{R}) + NormLoss(\hat{R}) \tag{22}$$

$$RotationalLoss(R, \hat{R}, W) = \frac{1}{\mathcal{J}} \sum_{j=1}^{\mathcal{J}} W_j \cdot 6DLoss(R_j, \hat{R}_j) \tag{23}$$

When assessing results in training, we split our weight evaluation equally between rotational and positional losses.

### 5.3. Pose weighting

In developing our method, we experimented with many different means of weighting individual joints that comprise the pose when assessing losses. There are many potential factors that influence the decision, but we sought to find the means to enable our method to be more rapidly applied across different skeleton topologies, with minimal tuning by the end user.

Our weighting method works on the basis of two assumptions, in local space error on joints that are further down the hierarchy has an oversized impact on accuracy of those descendants further along; in model space, perceptual error becomes more noticeable the further from the root of the hierarchy a joint is. We thus produce two sets of weightings, per joint, one in local space, the other the weights to apply on the model space poses. Our mechanism begins with a simple voting scheme, each time we find a joint with a parent we recursively apply a vote to its parent and so on, until we reach the bottom of the hierarchy that forms the skeleton. At this point, we now have our local space joint weights, we then simply invert the relationship to produce their model space equivalents, those joints with the fewest children, becoming the highest weighted.

In practice, this very simple approach reduced the need for us to continue manually tuning joint weights, we see increased overall accuracy in our a results and we more easily enable support for other skeletons without explicit changes to the method.

### 5.4. Consecutive poses in autoencoder training

In training our autoencoder, we train in pairs of temporally consecutive poses, as in [7]. We also similarly implement an additional loss for minimizing the difference between local and model space velocities in the training data and predictions.

$$VelocityLoss(P, \hat{P}) = \frac{1}{J}\sum_{j=1}^{J}\left\|\frac{(P_1 - P_0)}{\delta t} - \frac{(\hat{P}_1 - \hat{P}_0)}{\delta t}\right\|_1 \tag{24}$$

### 5.5. Contact label prediction and velocity losses in autoencoder training

We note that our method supports a user defined number of arbitrary labels, or *tags*, in particular, commonly used tagging for foot or hand contacts. As we mention in Section 4.2.1, these tags $T$ are provided as input during training, and a sigmoid activation is applied to produce predicted probabilities $\hat{T}$, we then assess using a binary cross entropy loss $BCE(T, \hat{T})$.

As we highlight in Section 5.4, we train our autoencoder in pairs of consecutive poses to evaluate an overall velocity consistency of our predictions. We further conditionally apply a loss on the predicted velocity $\hat{V}$, after forward kinematics, of joints deemed "in-contact" during phases where manual or automated tagging $C \in 0, 1$ indicates a given set of $J$ joints should be in contact and thus, the difference minimized in relative global velocity.

$$ContactLoss(V, \hat{V}, C) = \sum_{j=1}^{J} C_j \cdot \|V - \hat{V}\|_2 \tag{25}$$

### 5.6. Query distribution sampling and losses

For training the *Projector* model for sample retrieval based on motion matching queries, we use a similar process as seen in [7]. We sample a scale $n^\sigma$ from the uniform distribution, a value *noise* from the unit Gaussian to generate noise on the motion matching features $x$. We then apply feature weights $w$ and search for a result in our nearest neighbour structure of ground truth, this is then used create an index to select a motion matching feature vector and latent from our training data $X$ and $Z$. We then invoke a prediction from the *projector* using the weighted noisy motion matching features.

$$n^\sigma = \mathcal{U}(0, 1) \tag{26}$$

$$noise = \mathcal{N}(0, 1) \tag{27}$$

$$y = x + n^\sigma \cdot noise \tag{28}$$

$$y_{weighted} = y * \sqrt{w} \tag{29}$$

$$index = NearestNeighbour(y_{weighted}) \tag{30}$$

$$y_{actual}, z_{actual} = X[index], Z[index] \tag{31}$$

$$\hat{y}, \hat{z} = Projector(y_{weighted}) \tag{32}$$

Our losses are also similar, with the weights $\lambda_1 = 1.0$, $\lambda_2 = 10.0$ and $\lambda_3 = 0.3$. We encourage accuracy of the predictions and maintaining the distance of the query and result observed when sampling the source nearest neighbour acceleration structure.

$$YLoss = \lambda_1 \cdot \|y_{actual} - \hat{y}\|_1 \tag{33}$$

$$ZLoss = \lambda_2 \cdot \|z_{actual} - \hat{z}\|_1 \tag{34}$$

$$QueryDistLoss = \lambda_3 \cdot \left\|\|y_{actual} - y\|_2^2 - \|\hat{y} - y\|_2^2\right\|_1 \tag{35}$$

For $NearestNeighbour(y_{weighted})$, we use FAISS [28] to search a structure of pre-weighted ground truth to accelerate training times.

### 5.7. Smoother latent regularization

Where $N$ is the number of Lipschitz layers in the encoder, we calculate the cumulative product of each layer's Lipschitz bounds $softplus(c_i)$, as per [16]. We then combine with $L1$ and $L2$ metrics, and velocity minimization losses (weighted $\lambda_1$, $\lambda_2$ and $\lambda_3$, respectively), as per [7] (see also Section 5.4), on the latent values $z$ produced per batch $\mathcal{B}$ to produce a final regularization loss.

$$L1Loss(z) = \lambda_1 \cdot \|z\|_1 \tag{36}$$

$$L2Loss(z) = \lambda_2 \cdot \|z\|_2^2 \tag{37}$$

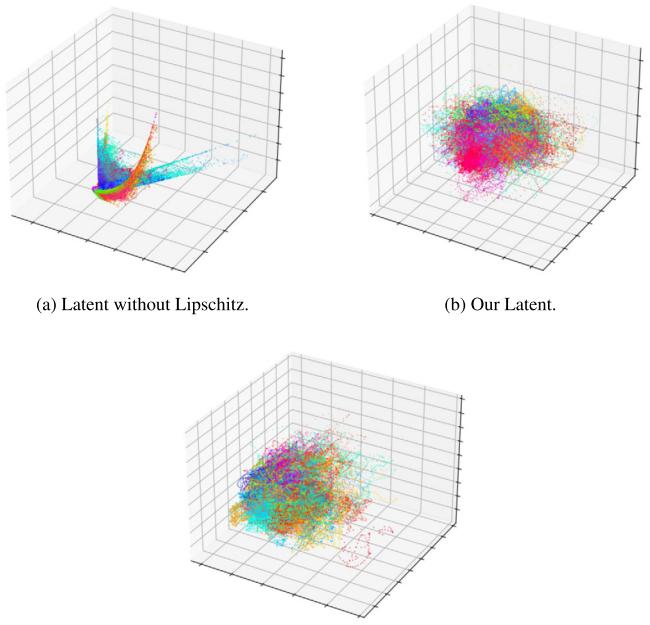
$$LatentVelocityLoss(z, \delta t) = \lambda_3 \cdot \left\|\frac{z_1 - z_0}{\delta t}\right\|_1 \tag{38}$$

$$LipschitzLoss = \alpha \cdot \prod_{i=1}^{N} softplus(c_i) \tag{39}$$

We found beyond a certain value, where minimizing the weights of the *NN* is a primary goal, there was little perceptible impact on our results by further optimization of the weighting $\alpha$ and settled on a value of $\alpha = 0.001$ across all experiments. For our $L1$, $L2$ and latent velocity losses, we used weights $\lambda_1 = 0.1$, $\lambda_2 = 0.1$ and $\lambda_3 = 0.01$. We observe a better overall distribution of samples in the latent for a given training set (see Fig. 4), which is conducive to improved results in our *stepper* implementation, when complemented with our usage of the *SMoE* architecture. Without the addition of the Lipschitz term, we see substantial banding and regular unpredictable jumps in the progression of a poses relative to another in temporal terms in the latent manifold. Intuitively, we see a relationship between samples of the motion matching features correlate well. By making this addition, we find a substantial benefit in the ability for our method to predict integration along the latent, and further, to correct drift in predictions over time.

### 5.8. SMoE and large batch sizes

As noted in Section 5.1, across all our models, we observed behaviour contrary to that seen in applying *SMoE* to large language models [10], whereby, liberally increasing batch sizes substantially improved the quality of our results. Consequently, for all models, we trained batch sizes of 12288, with 100 batches for each of 300 virtual epochs. We decided on these numbers as we observed the limiting factor in returns being a trade off of time available to train and memory available on GPU for such large batch sizes. We also observed a small increase in training time from our usage of *SMoE* compared to less accurate implementations comprised of *FFNs*.



(a) Latent without Lipschitz.

(b) Our Latent.

(c) Motion matching features.

**Fig. 4.** Latent and motion matching feature PCA projections in 3D.

### 5.9. Robustness to noise

As noted in section 7.1 of [7], one of the primary functions of the *projector* model, is to correct *drift* of predicted latents from the *stepper* back on to the valid latent manifold. Whilst the *stepper* provides great utility over a shorter window of time, its overall tendency to gradually move out of validity limits the ability to further stagger *projector* calls or to enable concurrent playback of decoded predictions.

In principle, this addition is made to ensure that we compensate for the out of sample results from the *projector*, and never train the *stepper* on ground truth that it has no guarantee of encountering at inference. To account for noise in input, we propose a simple method of applying noise during training. This induces an ability, combined with our Lipschitz latent, for our method to better align the *stepper* output for a greater period of time. We follow the same autoregressive process of training with a simple addition. To simulate initially noisy data for the first step with samples $z_0, M_0$ of each batch $\mathcal{B}$, we apply gaussian noise using the standard deviation of the latent and motion matching features, $\sigma(z)$ and $\sigma(M)$. We use the standard deviation to give some meaningful reference unit without introducing a dependency on calculating the variance of error returned post-*projector* training, which would necessitate removing the ability to train the *stepper* and *projector* concurrently.

$$z^n = \mathcal{N}(0,1) \cdot \sigma(z) \cdot \lambda_{scale} \tag{40}$$

$$M^n = \mathcal{N}(0,1) \cdot \sigma(M) \cdot \lambda_{scale} \tag{41}$$

$$z_{initial} = z_0 + z^n \tag{42}$$

$$M_{initial} = M_0 + M^n \tag{43}$$

$$\delta\hat{z}_1, \delta\hat{M}_1 = Stepper(z_{initial}, M_{inital}) \tag{44}$$

We then continue to train our *stepper* without further alterations, after making the first predictions using $z_{initial}$ and $M_{initial}$. After some experimentation, we found a larger value of $\lambda_{scale} = 0.15$ provided us with robust results, see Section 6.3.1.

## 6. Evaluation

In this section, we present evaluation of the performance of our method in multiple areas. All model inference is handled via our internal technology that enables deploying pre-trained models within video games on consumer hardware and executes experts correctly.

In the development of our method, we compared exporting models from *PyTorch* for execution via *ONNX Runtime* [29], however, the current implementation across platforms is unable to successfully conditionally reduce the work of evaluating zero weighted experts, instead forming several add-multiply operations for each unused expert. As a consequence, our method requires some degree of specialized implementation for on-device inference to see the theoretical gains in compute complexity reduction.

**Table 9**
Details of the animation database used for metrics in our evaluation.

| Frames | Sample Rate | Joints Per Pose | Memory |
|---|---|---|---|
| 154 130 | 60 hz | 96 | 398.63 Mb |

**Table 10**
Locomotion motion matching features.

| Feature | Length |
|---|---|
| Left Foot Linear XYZ Velocities | 3 |
| Right Foot Linear XYZ Velocities | 3 |
| Pelvis Linear XYZ Velocities | 3 |
| Left Foot Position | 3 |
| Right Foot Position | 3 |
| Root Position XZ at Time −0.1 s | 2 |
| Root Direction XZ at Time −0.1 s | 2 |
| Root Position XZ at Time 0.3 s | 2 |
| Root Direction XZ at Time 0.3 s | 2 |
| Root Position XZ at Time 0.6 s | 2 |
| Root Direction XZ at Time 0.6 s | 2 |
| Root Position XZ at Time 1.0 s | 2 |
| Root Direction XZ at Time 1.0 s | 2 |
| Phase Embedding | 10 |
| Total | 41 |

### 6.1. Animation data and motion matching features in evaluation

For examples referenced in this section, we compared performance on a common locomotion task, with animation data sampled from motion captured performed by our colleagues, covering a range of gaits, speeds and orientations. We also further augmented the data by mirroring suitable assets to increase coverage to produce in the region of 42 min of sample data, total data noted in Table 9.

We defer to [4] for fuller details on the base functionality and implementation details of motion matching, but state here the features that were extracted from animation data to train the locomotion example in our paper. We note that the method was also validated on quadrupeds and other styles of bipedal locomotion and actions with varying sizes of motion matching features.

We provided a motion matching feature vector per-frame of animation, with the components listed in Table 10.

When implementing our reference motion matching database, we store these values as half-precision floating point, but when used for training, we take the full single-precision values.

### 6.2. Runtime performance

We evaluated single threaded performance with multiple runs on the same CPU, an Intel i9-11900K 3.5 GHz. We trained comparable *FFNs*, as detailed in [13], with the same training data, for all components. In Table 11, we compare both execution cost per inference step, in microseconds ($\mu s$), and memory required for weights, in megabytes (Mb). The lowest execution or memory cost is highlighted in **bold**.

In expressing our models using *SMoE*, as stated, we are able to substantially reduce the theoretical complexity count of operations for inference of each model compared to equivalent *FFNs*. When testing smaller expert sizes, we saw diminishing returns for performance

**Table 11**
Relative CPU performance and memory usage of components between our method and *FFNs* as in *LMM*.

| Model | FFN (μs) | Ours (μs) | FFN (Mb) | Ours (Mb) |
|---|---|---|---|---|
| Decoder | 60.94 | **19.94** | 3.86 | **3.63** |
| Stepper | 54.02 | **1.85** | 3.30 | **1.45** |
| Projector | 98.57 | **3.58** | 5.24 | **4.87** |
| Total | 213.53 | **25.37** | 12.40 | **9.95** |

**Table 12**
Components of a motion matching implementation of our data and their memory sizes.

| Component | Size (Mb) |
|---|---|
| Compressed animation Data | 36.64 |
| KD-Tree | 3.68 |
| Motion matching features DB | 12.05 |
| Total | 52.37 |

**Table 13**
Comparing CPU execution time and memory usage of each method.

| Method | CPU time (μs) | Memory (Mb) |
|---|---|---|
| FFN | 213.53 | 12.40 |
| Our method | **25.37** | **9.95** |
| Motion matching | 78.20 | 52.37 |

on CPU, which leads us to believe that our testing environment is potentially bound by memory access for any further performance improvements. Further reductions in execution and memory costs can be introduced by quantizing from single-precision floating-point to half, where full half-precision math support is available on a given device.

As seen in *LMM* [7], even an *FFN* implementation brings about a substantial reduction in memory usage compared to even compressed animation. Without taking into account additional components used by a motion matching implementation, we see a significantly higher compression ratio using our models versus in-engine compression formats, 9.95Mb versus 36.64Mb. When we factor in the omission of spatial representations of the motion matching features and their database, that advantage only grows further.

We implement an optimized reference motion matching system, using *nanoflann* [30,31] to accelerate our spatial searches for samples, with our compressed animation data and storing our motion matching features database as half-precision floats. This allows us to generate new databases and test prior to training models for final usage, but also gives us an important point of comparison. Several components are included and require additional storage, as specified in Table 12.

We compare all three methods in Table 13 and, again, highlight the lowest execution or memory cost in **bold**.

As we demonstrate, our method provides a solution that does not require the user compromise on performance nor storage, and even performs favourably compared to the time decompressing a single frame in high-quality compression libraries for animation of around 20–30$\mu s$ [32] alone.

#### 6.2.1. Further performance notes

When testing in game-like environments, using our method, it is possible to execute hundreds of characters within typical execution budgets for animation, typically between 1–2*ms* across multiple CPU threads. For example, in Fig. 1, we animate around 200 characters concurrently, with no reduction in detail of sampling across 4 CPU threads concurrently, in under 2*ms*.

We give total system costs, but we highlight a number of considerations when evaluating. By default, for all methods, we only trigger the projector or sample search every 100 ms of game updates, so this cost can be further amortized over a number of frames. Though we note, unlike both motion matching and *LMM* our projector does not represent the highest cost of our system at runtime.

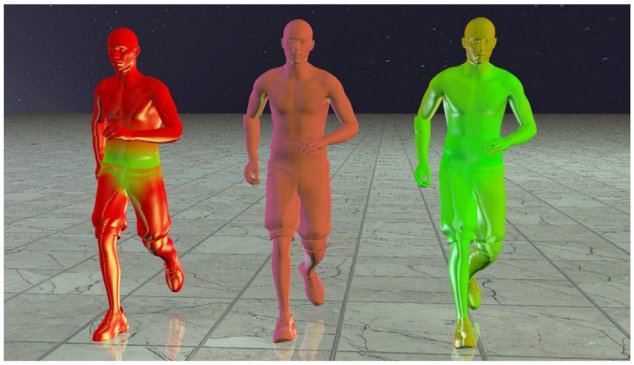


**Fig. 5.** Comparing pose outputs from left to right: *FFN*, ground truth, our method.

Furthermore, depending on the strategy employed for decoding poses, one can further optimize characters using our method as per Section 7.1. Equally, many games employ strategies to stagger the frequency of individual pose updates according to an in-game level-of-detail or *"LODing"*, with pose updates as low as 5–10*Hz* not uncommon for distant characters in *"AAA"* open-world video games. Prior work to ours does not sufficiently stabilize predictive behaviour over such large timescales, but due to the increased robustness of our method, such substantial times between updates becomes feasible to employ to make such a decision for overall performance.

### 6.3. Pose accuracy

Comparing *FFNs* trained with our losses using arrangements as in *LMM*, our method sees notable improvements in pose reconstruction, see Fig. 5. We colour vertices in predictions based on the error of the bones they are skinned to, with a linear green-to-red heatmap, where green sees error of 0.004 m or below and red of 0.02 m or above.

By increasing the relative size of the *FFNs*, we can see an improvement in accuracy that matches ours, at the expense of additional compute complexity and memory size for model weights, furthermore, when tuning batch sizes and steps required to achieve this accuracy, the training time for the pose autoencoder *FFN* increased to 5 days. However, notably, increasing the size of the models does not leverage the benefit of the Lipschitz regularizing effect on the latent and subsequent predictions output from other models in the system, it also only further increases the already highlighted disadvantage in performance.

#### 6.3.1. Retrieval and predictive accuracy

Our implementation compares favourably in error when retrieving samples of the latents produced from training pose autoencoders, however, we see a significant reduction in the error when retrieving the source motion matching features. Our method provides the means to safely predict latents and motion matching features over a longer time period, this allows us to implement decision making similar to traditional motion matching, whereby, we can avoid switching samples that appear to be less desirable than those already playing in the system at runtime, thereby giving us the opportunity to more smoothly play back our resultant choices.

When we introduce noise into our queries and predict future updates via the stepper we see further improvements in the robustness of our results in comparison, with a more pronounced impact on the relative error in the latent representation.

We find the smoothing impact on the latent from the Lipschitz constrained encoder produces long and consistent predictions over an extended period of time. We also tend to see a linear increase in error over time of these predictions. Some examples are presented in Fig. 8, which entails a search of the motion matching features for the frame at time 0 for the projector and then integration via the

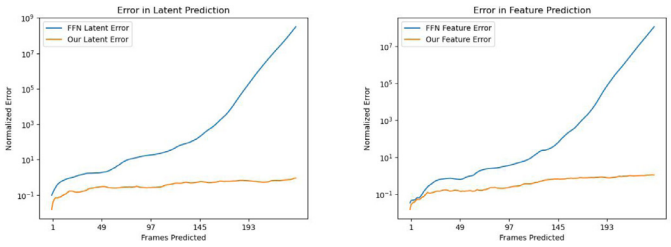



**Fig. 6.** Mean error across autoregressive predictions.

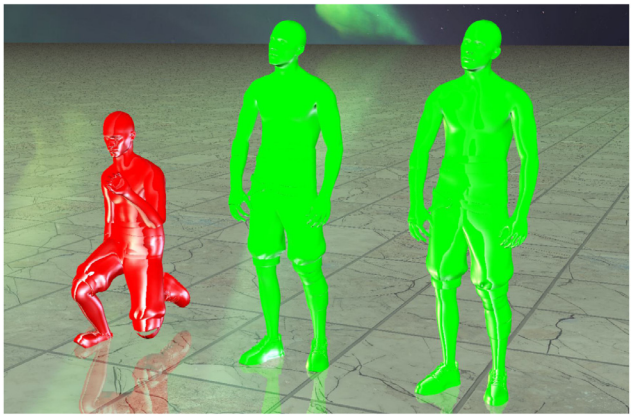


**Fig. 7.** Comparing one second of forward prediction, left, *FFN*, middle, ground truth, right, ours. We see that without the Lipschitz regularization, pose predictions rapidly drift away from the valid manifold.

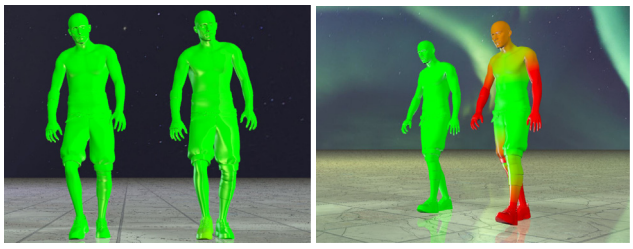
stepper a number of samples ahead, prior to decoding. We compare the results of predicting from 40000 randomly drawn samples. We randomly apply Gaussian sampled noise on the first query, then sample autoregressively, as in training. Note, our smoother latent is more predictable for a longer period of time than the results from an *FFN* with no Lipschitz bounds, see Fig. 6. There is some contrast in the duration at which our predictions remain coherent, when we compare to *FFN* output without Lipschitz regularization, the decoded pose does not correlate with any plausible mapping from the latent manifold to a real pose, as in our method, see Fig. 7.

Fig. 8 shows some examples of the typical quality of predictions we see in our method. At frame 0, we see a close relationship with ground truth, typical mean accuracy across the pose in the region of 0.003 m. At frame 120, we still see a relationship with ground truth, but it is possible to discern differences when comparing directly, typical mean accuracy across the pose in the region of 0.02 m or below. At frame 240, we see a coherent pose, but there are now significant visible differences, typical mean accuracy in the region of 0.03 m. Whilst we are at some points able to predict some 240 frames or more from an initial sample, we find sufficient number of outlier cases where we produce an unusable pose or sample that does not align with valid posing on the latent manifold. As a consequence, we choose to limit all predictions to a maximum of 120 frames, or 2 s.

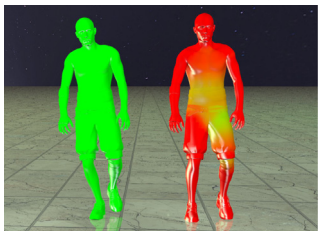
## 7. Impact of Lipschitz-continuous neural network regularization

For the sake of brevity, in [13], we limited the depth of our reasoning behind employing a Lipschitz continuous encoder model, here we give some further detail and discussion.

In Fig. 4, we demonstrate a PCA plot to illustrate the relative differences between learning a latent manifold with and without the Lipschitz constraint, as we stated, there is an observably improved density within this space that more closely maps to the linearly interpolatable manifolds within which the motion matching features and



(a) Frame 0 (b) Frame 120



(c) Frame 240

**Fig. 8.** Comparing accuracy of pose retrieval and predictions over time, character on the left, ground truth, on the right, our method.

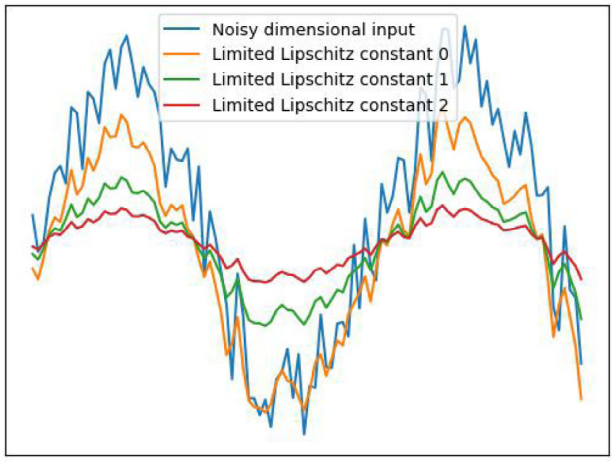



**Fig. 9.** A simple example of noisy 2D input plotted against the output after limiting the Lipschitz constant uniformly.

source animation data exist. However, to further illustrate we can see the impact of increasingly limiting the Lipschitz constant of a simple noisy 2-dimensional example in Fig. 9, the plot gradually reduces closer and closer to a simple linear relationship across the range for which the constant is constrained.

We observe the Lipschitz constant is the absolute sum of the gradients across the affected regions.

$$c = \sum \left| \frac{\delta y}{\delta x} \right| \tag{45}$$

Thus, to limit the Lipschitz constant by a prescribed constant, $L$.

$$Scale = \frac{L}{c} \tag{46}$$

$$LimitedGradient = \frac{\delta y}{\delta x} * Scale \tag{47}$$

When we extend this to the formulae detailed in Section 3.2, we can understand that by learning a single limiting Lipschitz constant for a given weight matrix, all affected dimensions are constrained to adhere to the same constrained relationship.

This provides a more holistic regularization factor than using $L1$, $L2$ and $LatentVelocity$ losses in Section 5.7, where ultimately learning our

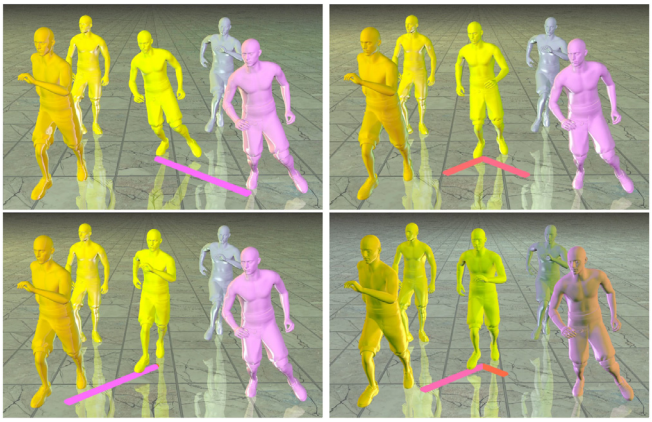


**Fig. 10.** Our method displays plausible character posing when blending between disconnected samples in the latent pose manifold. Clockwise from top-left, we see incremental blending from the ground-truth sample on the front-right over to the front-left. The character in the middle, sampling the blend latents, produces smooth, plausible, posing throughout the transition.

relationship between individual samples in the data are only influenced by very local proximity of those samples.

As a consequence, by learning the smallest feasible constant per-layer of the latent encoder, we induce an increasingly linear relationship between samples in the resultant latent manifold. We can see how this does not also enforce a required change or assumption of the distribution of the source training data, as seen in encoding methods such as *VAEs*, and further eliminates much of the undefined space in the manifold, thereby enabling a simpler learnable function to decode and predict future samples in the pose decoder and stepper that more closely aligns with the intuition of linear relationships between nearby pose configurations in the source data.

### 7.1. Smooth pose blending in latent space

In replicating the behaviour of motion matching, additional expenses are added to a runtime implementation due to the necessity of providing the means to blend poses as a character transitions between constituent animations that make the complete scene. In order to address the potential performance penalty of cross-blending or decoding multiple poses [7], *inertialization* [33] has been leveraged in other systems to reduce costs to a fixed transition.

Whilst effective, *inertialization* implementations vary from complex polynomial solutions to user-tuned spring or velocity based solutions. In practice, these can be challenging to predictably tune. In seeking to reduce the complexity and increase the robustness of our method, we experimented with the linearizing properties of our latent regularization (as in [16]) to support an intuitive alternative to traditional means of linear blending between poses, as in Fig. 10.

Our method for interpolation of pose $P$ prior to decoding becomes a simple matter of weighted linear interpolation, we take $N$ active samples $X$ and their respective weights $W$ to generate a single sample $Y$ that can then be decoded and presented in-game.

$$WeightSum = \sum_{i=1}^{N} W_i \tag{48}$$

$$Y = \frac{1}{WeightSum} \sum_{i=1}^{N} X_i W_i \tag{49}$$

$$P = DECODE(Y) \tag{50}$$

Without enforcing the Lipschitz bounds on an *NN*, blending of the latents does not become a definable task, as only learning local proximity between given pairs of samples is enforced by the other losses that shape the latent. In our experiments with our *FFN* implementation (as per [7]), we are unable to move smoothly along the latent manifold, interpolating through undefined regions, and instead produce small changes around an apparently averaged pose that do not correlate with the ground truth input poses, as can be seen in Fig. 11.

Whilst we see a great deal of success in using this extension to our method to sample a final pose for presentation in-game, such usage is not a prerequisite for a useful implementation.

### 7.2. Pose extrapolations

As in [16], we can apply extrapolation to generate new data, however, such usage rapidly produces implausible poses, see Fig. 12 for examples. With investment into introducing additional validation on the manifold of a wider range of pose configurations than just the source animation frames, we believe there is interesting potential for investigating further use in the future.

## 8. Impact of improved expert utilization

As discussed in Section 5.1, we improved upon the utilization of experts as seen in [13] with a more sound *KL-divergence* based loss. In training, we observed a more rapid optimization of this loss, whilst also seeing improvements of reported mean error across the output of

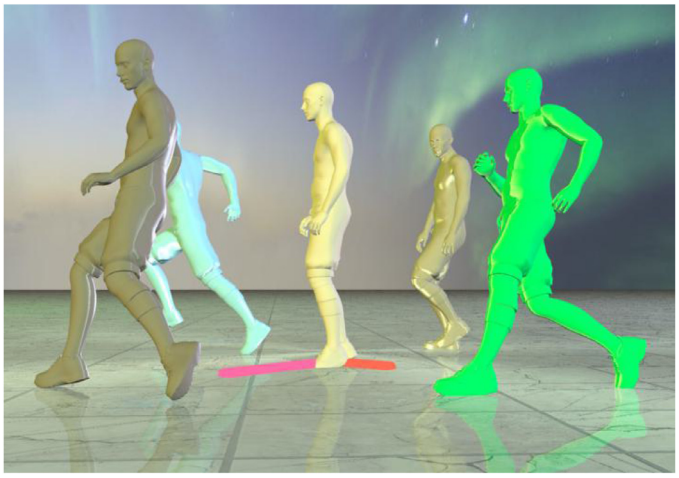


**Fig. 11.** Without enforcing Lipschitz regularization, linear blending is not possible on the latent manifold. Here we see and example blending the latent without Lipschitz regularization, an apparently averaged pose is seen, rather than the equivalent of the linear blend of the source data that our method is able to produce.

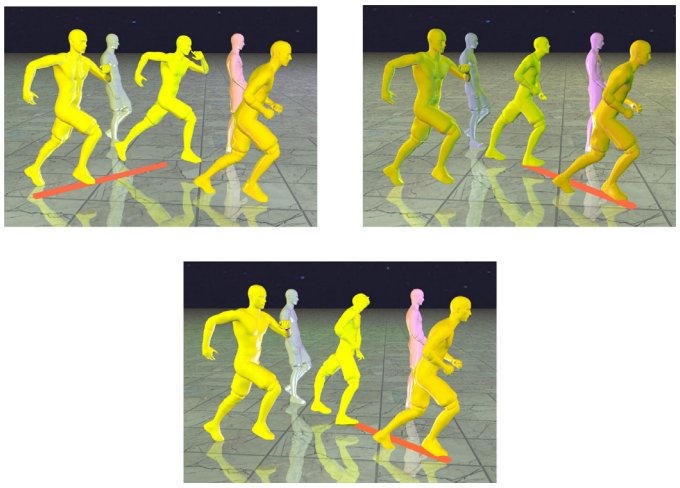


**Fig. 12.** Extrapolation succeeds in a small margin, but rapidly produces implausible and undesirable posing. In the samples we see blends where the total weight is created by overweighting two samples beyond a normalized weight of 1.0.

all our models. Most significantly, the benefits were observed where a great number of experts are created, in the decoding component of our pose autoencoder, where we have observed an overall improvement in mean pose error in the region of 0.001–0.002 m with the same training configurations. In practice, whilst this difference is relatively minimal, we see the ability to reduce the number of iterations required to reach satisfactory output, as in [13], with the potential to reduce required training times. This is an important benefit when utilizing neural network based methods within the development environment of a video game, where iteration times can be a significant consideration for the utilization of such technology.

## 9. Conclusion

We demonstrate several advantageous developments on prior state-of-the-art, to enable the desirable behaviour of motion matching like animation control at a scale previously unachievable without further compromises. We note, that the superior improvements in memory reduction as were seen in *LMM* [7] no longer need to be evaluated as a trade off for execution costs when implementing a motion matching system by using our method. Indeed, we propose that our method instead unlocks both greater potential for larger animated character counts and execution on more limited contemporary video game platforms, without necessitating reductions in quality or difficult to implement performance strategies.

We also see that introducing Lipschitz bounds in motion synthesis allows for predictive benefits and a smoother interpretable latent manifold for animation representation. We do not rely on fully developing generality in our method, however, we see interesting avenues in how this method may apply to generate new data, particularly by means of extrapolation or interpolation of encodings of samples that are not seen in the training set, as we have highlighted.

Whilst we chose to try to strike a balance between performance and reconstruction quality, we note that the complexity and memory usage reductions from our method leave a greater capacity for scaling up or down the size and number of experts within the models to achieve different desired quality goals.

Lastly, whilst motion matching provides flexible means to define the features used to define a match, we find there is still an undesirable requirement for engineering feature selection and weights, beyond the specific control values that define the task, such as event times or trajectory goals for root motion. This remains true, even with the improvements that more autonomously support alignment via phases, introduced by utilizing ideas in [14]. We are keen to see how the onerous feedback loop of this task could be improved to allow end users of such a system more time to focus on the quality of the input and output.

## CRediT authorship contribution statement

**Tobias Kleanthous:** Investigation, Methodology, Writing – original draft, Writing – review & editing. **Antonio Martini:** Writing – original draft, Methodology, Investigation, Writing – review & editing.

## Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

## Data availability

The data that has been used is confidential.

## Acknowledgements

We thank our colleagues on the Production R&D team for their valuable feedback during development and participation in hours of motion capture in the depths of summer. We also thank our colleagues at Sharkmob for the use of their time and motion capture facilities in producing data for our research, and the team at 10 Chambers for their input and additional data in the further development of our method.

## References

[1] Kleanthous T. Making the believable horses of red dead redemption II. 2021, GDC '21, URL https://www.youtube.com/watch?v=8vtCqfFAjKQ.
[2] Lee Y, Wampler K, Bernstein G, Popović J, Popović Z. Motion fields for interactive character locomotion. In: ACM SIGGRAPH Asia 2010 papers on - SIGGRAPH ASIA '10. ACM Press; 2010, http://dx.doi.org/10.1145/1882262.1866160.
[3] Büttner M, Clavet S. Motion matching - the road to next gen animation. 2015, Nucl.ai '2015.
[4] Clavet S. Motion matching and the road to next-gen animation. 2016, GDC '16, URL https://www.gdcvault.com/play/1023280/Motion-Matching-and-The-Road.
[5] Holden D, Komura T, Saito J. Phase-functioned neural networks for character control. ACM Trans Graphics 2017;36(4):1–13. http://dx.doi.org/10.1145/3072959.3073663.
[6] Starke S, Zhao Y, Komura T, Zaman K. Local motion phases for learning multi-contact character movements. ACM Trans Graphics 2020;39(4). http://dx.doi.org/10.1145/3386569.3392450.
[7] Holden D, Kanoun O, Perepichka M, Popa T. Learned motion matching. ACM Trans Graph 2020;39(4). http://dx.doi.org/10.1145/3386569.3392440.
[8] Raparthi N, Acosta E, Liu A, McLaughlin T. GPU-based motion matching for crowds in the unreal engine. In: SIGGRAPH Asia 2020 posters. ACM; 2020, http://dx.doi.org/10.1145/3415264.3425474.

[9] Maiorca A, Hubens N, Laraba S, Dutoit T. Towards lightweight neural animation: Exploration of neural network pruning in mixture of experts-based animation models. In: de Sousa AA, Debattista K, Bouatouch K, editors. Proceedings of the 17th international joint conference on computer vision, imaging and computer graphics theory and applications, VISIGRaPP 2022, volume 1: GRAPP, online streaming, February 6-8, 2022. SCITEPRESS; 2022, p. 286–93. http://dx.doi.org/10.5220/0010908700003124.

[10] Zoph B, Bello I, Kumar S, Du N, Huang Y, Dean J, et al. ST-moe: Designing stable and transferable sparse expert models. 2022, arXiv:2202.08906.

[11] Fedus W, Zoph B, Shazeer N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. J Mach Learn Res 2022;23:120:1–39, URL http://jmlr.org/papers/v23/21-0998.html.

[12] Lepikhin D, Lee H, Xu Y, Chen D, Firat O, Huang Y, et al. Gshard: Scaling giant models with conditional computation and automatic sharding. In: 9th International conference on learning representations, ICLR 2021, virtual event, Austria, May 3-7, 2021. OpenReview.net; 2021, URL https://openreview.net/forum?id=qrwe7XHTmYb.

[13] Kleanthous T, Martini A. Learning robust and scalable motion matching with Lipschitz continuity and sparse mixture of experts. In: ACM SIGGRApH conference on motion, interaction and games. ACM; 2023, http://dx.doi.org/10.1145/3623264.3624442.

[14] Starke S, Mason I, Komura T. DeepPhase: periodic autoencoders for learning motion phase manifolds. ACM Trans Graph 2022;41(4). http://dx.doi.org/10.1145/3528223.3530178.

[15] Gouk H, Frank E, Pfahringer B, Cree MJ. Regularisation of neural networks by enforcing Lipschitz continuity. Mach Learn 2020;110(2):393–416. http://dx.doi.org/10.1007/s10994-020-05929-w.

[16] Liu H-TD, Williams F, Jacobson A, Fidler S, Litany O. Learning smooth neural functions via Lipschitz regularization. In: Special interest group on computer graphics and interactive techniques conference proceedings (SIGGRAPH '22 conference proceedings). New York, NY, USA: ACM; 2022, http://dx.doi.org/10.1145/3528233.3530713.

[17] Miyato T, Kataoka T, Koyama M, Yoshida Y. Spectral normalization for generative adversarial networks. 2018, arXiv:1802.05957.

[18] Ling HY, Zinno F, Cheng G, van de Panne M. Character controllers using motion VAEs. ACM Trans Graph 2020;39(4). http://dx.doi.org/10.1145/3386569.3392422, URL https://dl.acm.org/doi/10.1145/3386569.3392422.

[19] Hendrycks D, Gimpel K. Gaussian error linear units (GELUs). 2023, arXiv:1606.08415.

[20] He K, Zhang X, Ren S, Sun J. Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. 2015, arXiv:1502.01852.

[21] Paszke A, Gross S, Massa F, Lerer A, Bradbury J, Chanan G, et al. Pytorch: An imperative style, high-performance deep learning library. In: Advances in neural information processing systems 32. Curran Associates, Inc.; 2019, p. 8024–35, URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[22] Pavllo D, Grangier D, Auli M. QuaterNet: A quaternion-based recurrent model for human motion. 2018, arXiv:1805.06485.

[23] Li P, Aberman K, Zhang Z, Hanocka R, SorkineHornung O. Ganimator: Neural motion synthesis from a single sequence. ACM Trans Graph 2022;41(4). http://dx.doi.org/10.1145/3528223.3530157.

[24] Zhou Y, Barnes C, Lu J, Yang J, Li H. On the continuity of rotation representations in neural networks. 2020, arXiv:1812.07035.

[25] Xie X, Zhou P, Li H, Lin Z, Yan S. Adan: Adaptive nesterov momentum algorithm for faster optimizing deep models. 2023, arXiv:2208.06677.

[26] Pavlitska S, Hubschneider C, Struppek L, Zöllner JM. Sparsely-gated mixture-of-expert layers for CNN interpretability. In: International joint conference on neural networks, IJCNN 2023, gold coast, Australia, June 18-23, 2023. IEEE; 2023, p. 1–10. http://dx.doi.org/10.1109/IJCNN54540.2023.10191904.

[27] Hempel T, Abdelrahman AA, Al-Hamadi A. 6D rotation representation for unconstrained head pose estimation. In: 2022 IEEE international conference on image processing. 2022, p. 2496–500. http://dx.doi.org/10.1109/ICIP46576.2022.9897219.

[28] Johnson J, Douze M, Jégou H. Billion-scale similarity search with GPUs. IEEE Trans Big Data 2019;7(3):535–47.

[29] ONNX Runtime developers. ONNX runtime. 2021, Version: x.y.z, https://onnxruntime.ai/.

[30] Suju DA, Jose H. FLANN: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas. In: 2017 fourth international conference on signal processing, communication and networking. 2017, p. 1–6. http://dx.doi.org/10.1109/ICSCN.2017.8085676.

[31] Blanco JL, Rai PK. Nanoflann: a C++ header-only fork of FLANN, a library for nearest neighbor (NN) with KD-trees. 2014, https://github.com/jlblancoc/nanoflann.

[32] Frechette N, Animation Compression Library contributors. Animation compression library. 2017, https://github.com/nfrechette/acl.

[33] Bollo D. Inertialization: High-performance animation transitions in gears of war. 2018, URL https://www.youtube.com/watch?v=BYyv4KTegJI.