Mem-MLP: Real-Time 3D Human Motion Generation from Sparse Inputs

Supplementary Material

Method	MPJPE ↓	Upper PE ↓	Lower PE ↓	Jitter ↓
Rotation Branch	3.36	1.56	5.97	7.34
Position Branch	3.07	1.24	5.72	5.45
Rotation + Position Branch	3.08	1.25	5.72	6.03

Table 1. Experiments on inverse kinematics. We run an optimized inverse kinematics solution to correct the estimated angles (rotation branch) using 3D joint locations (position branch) as target points.

#Layers (PB)	#Layers (RB)	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓
1	1	2.60	3.14	15.15	6.04
2	2	2.57	3.08	15.02	6.03
4	4	2.57	3.11	15.04	6.41

Table 2. Experiments on layer depth of multi-task predictor. PB and RB denote position and rotation branches.

Method	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓
MHP w/ MB	2.59	3.40	17.80	5.57
MHP wo/ MB	2.57	3.08	15.08	6.03

Table 3. Effect of Memory-Block (MB) integration into Multi-Head Predictor (MHP).

In the ablation study, we also conduct additional experiments on memory block, multi-task predictor, number of MLP blocks, sequence length, impact of VQ-VAE and comparison with LSTM and GRU alternatives. All methods are evaluated for Scenario-1 (S1).

1. Architectural Design

Inverse Kinematics: We observe that the estimated 3D joint positions generated by the second branch of our multi-head predictor are more accurate than those from the first branch. However, since the SMPL model cannot directly reconstruct 3D body pose representations from joint positions alone, we employ an inverse kinematics (IK) solver using the L-BFGS optimization method, achieving optimal results within 15 iterations. The IK solver optimizes joint rotations by initializing with the output of the rotation branch and targeting the more accurate 3D joint positions from the second branch. The corresponding results are presented in Table 1.

Layer Depth of Multi-Task Predictor: In Table 2 we evaluate the effect of different numbers of MLP blocks per branch. Our default configuration (2 layers) yields the best trade-off, especially reducing Jitter by 16.1% over 1-layer and 13.9% over 4-layer configurations.

Layer Depth of MLP Backbone: We conduct experiments

MLP Blocks L	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓	FLOPS ↓
8	2.57	3.08	15.02	6.03	0.25G
10	2.58	3.07	14.78	6.19	0.27G
12	2.59	3.11	14.88	6.10	0.30G

Table 4. Effect of Number of MLP blocks L on model predictive accuracy.

Seq. Length T	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓
11	2.60	3.27	20.81	15.05
21	2.61	3.19	16.82	8.22
41	2.57	3.08	15.02	6.03
60	**2.56**	**3.05**	**14.27**	**5.35**

Table 5. Impact of sequence length T on model performance.

Method	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓
Mem-MLP w/o VQ-VAE	2.60	3.16	15.35	6.61
Mem-MLP w/ VQ-VAE	2.57	3.08	15.08	6.03

Table 6. Effect of VQ-VAE in Memory Block.

with different number of MLP blocks in the backbone of our architecture. As presented in 4, our default configuration ($L = 8$) presents better MPJRE and Jitter results, while ($L = 10$), shows an improvement in MPJPE and MPJVE metrics, but increases the computational overhead in our model, which falls slightly under real-time performance.

Integration of Memory-Block to Multi-Head Predictor: It is observed that, Table 3, including MB in MHP improves temporal consistency (lower Jitter), although MPJRE and MPJPE are slightly worse. This shows that MB smooths the motion by leveraging past context.

Impact of Sequence Length: Longer sequences improve performance (especially Jitter and MPJVE), but increase computation, Table 5. This trade-off must be tuned depending on application constraints.

Impact of VQ-VAE: To better understand the impact of code-vector $\mathcal{E}_l$ in the Memory Block of our architecture we conduct another experiment where we remove the code-vector from our Memory Block, and the feature $\mathbf{Z}_m$ is given by the following equation: $\mathbf{Z}_m = \mathbf{m} * \mathbf{Z}_\theta$. Table 6 shows that the incorporation of code-vector $\mathcal{E}_l$ in our Memory Block enhances the predictive accuracy of our model in all metrics. In our experiments, we set the dimension of the code-vector to 64.

VQ-VAE Details: Below we report extra details on the VQ-VAE model. The overall architecture is illustrated in

Model	$\mathbf{L_{enc}}$	MPJPE ↓	Upper PE ↓	Lower PE ↓	Jitter ↓
Mem-MLP	2	2.59	3.11	15.13	6.08
Mem-MLP	4	2.57	3.08	15.02	6.03
Mem-MLP	6	2.58	3.15	15.34	6.21

Table 7. Experiments on different sizes of $\mathbf{L_{enc}}$ and their impact on Mem-MLP.

Figure 1. We adopt a lightweight MLP-based design: each MLP block comprises layer normalization, a single linear layer and a SiLU activation; this block is used uniformly in both the encoder and decoder. The encoder depth is denoted $\mathbf{L_{enc}}$, with baseline value $\mathbf{L_{enc}} = 4$, the decoder depth $\mathbf{L_{dec}}$, with baseline value $\mathbf{L_{dec}} = 1$ and the latent dimensionality is $d_{z_s} = 256$ of the model. The training is performed trying to minimize a loss that consists of two terms, the rotation reconstruction loss (Equation 1), and the commitment loss introduced in [40], which encourages the encoder outputs to commit to specific codebook entries. The discrete codebook is denoted $\mathcal{C} = \{\mathbf{e}_1, \dots, \mathbf{e}_K\} \subset \mathbb{R}^{d_{z_s}}$, where $K = 64$ is the number of codebook entries in our best performing Mem-MLP models. This choice yields a compact yet expressive discrete representation suitable for downstream reconstruction and prior modeling. Mem-MLP pipeline is only using the pre-trained, frozen, encoder of a VQ-VAE model during training and inference. This design reduces training time while improving accuracy. We also experimented with joint training of the VQ-VAE and Mem-MLP, but observed substantial performance degradation. We train the VQ-VAE for 100 epochs using the Adam optimizer with batch size 256. The Adam hyperparameters are set to $\beta = (0.9, 0.99)$ and a weight decay of 10^{-4}. The initial learning rate is $1 \times 10 - 4$ and it is decreased by a factor of 0.2 at scheduled milestone epochs [20, 50, 70].

VQ-VAE Configuration Settings: To quantify the sensitivity of downstream performance to the VQ-VAE configuration, we perform ablation experiments over codebook sizes $K \in 32, 64, 128$ and the encoder depths $\mathbf{L_{enc}} \in 2, 4, 6$. For each VQ-VAE variant we freeze the learned codebook and integrate the corresponding discrete priors into the final Mem-MLP model. More specifically, Table 7 presents the results for varying encoder depths $\mathbf{L_{enc}}$, while keeping all other VQ-VAE hyper-parameters ($\mathbf{L_{dec}}$, d_{z_s}, and $K = 64$) fixed. Each trained VQ-VAE is subsequently employed within Mem-MLP, where a new model is trained from scratch under the corresponding configuration. In parallel, Table 8 reports the ablation results on the VQ-VAE codebook size K, with all remaining hyperparameters held constant.

Comparison with LSTM & GRU Memory Blocks: To further assess the effectiveness of our approach, we implemented recurrent memory variants by replacing the memory block with LSTM- or GRU-based layers. The architecture is extended with a recurrent memory block that applies Layer Normalization to the input sparse features, followed by either an LSTM or GRU layer, and a SiLU activation function. During training, the recurrent units are initialized differently depending on the configuration: (a) LSTM-

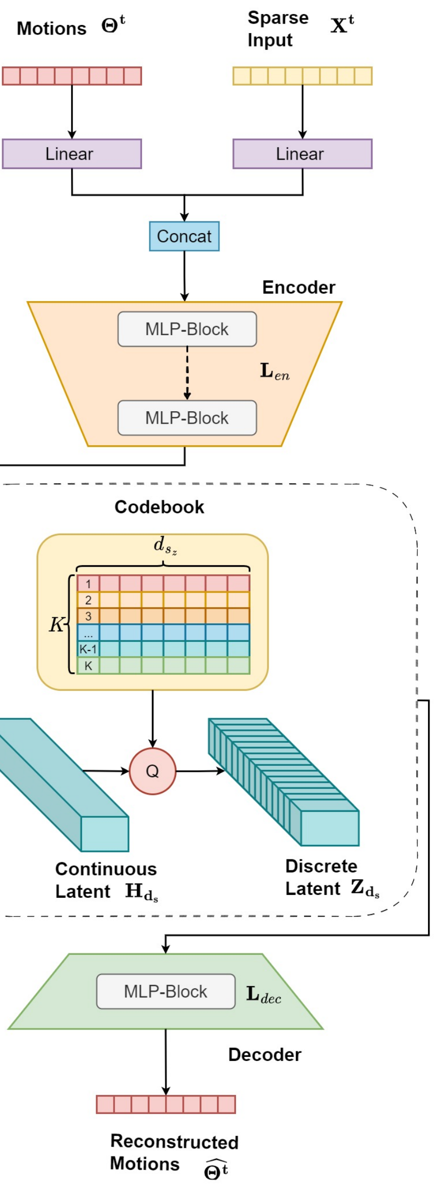
Motions Θ^t
Sparse Input $\mathbf{X}^t$
Linear
Linear
Concat
Encoder
MLP-Block
$\mathbf{L}_{en}$
MLP-Block
Codebook
d_{s_z}
K
1
2
3
...
K-1
K
Q
Continuous Latent $\mathbf{H}_{\mathbf{d_s}}$
Discrete Latent $\mathbf{Z}_{\mathbf{d_s}}$
MLP-Block
$\mathbf{L}_{dec}$
Decoder
Reconstructed Motions $\widehat{\Theta}^t$

Figure 1. VQ-VAE architecture diagram

Model	K	MPJPE ↓	Upper PE ↓	Lower PE ↓	Jitter
Mem-MLP	32	2.57	3.13	15.30	6.14
Mem-MLP	64	2.57	3.08	15.02	6.03
Mem-MLP	128	2.62	3.20	15.55	6.52

Table 8. Experiments on different sizes of K, and their impact on Mem-MLP.

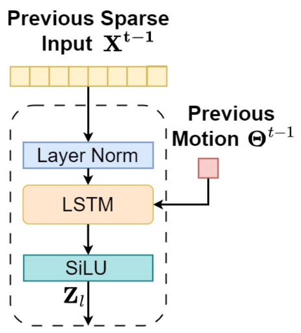
Previous Sparse Input $\mathbf{X}^{t-1}$
Previous Motion $\mathbf{\Theta}^{t-1}$
Layer Norm
LSTM
SiLU
$\mathbf{Z}_l$

Figure 2. LSTM-based memory block diagramm.

based memory block: The hidden state is initialized using the final time step of the sparse motion features, thereby incorporating prior motion context, while the cell state is zero-initialized. (b) GRU-based memory block: The hidden state alone is initialized from the final time step of the sparse motion features. Figure 2 provides visual explanation on the LSTM-based memory block structure. In evaluation mode, both LSTM and GRU blocks are run without explicit initialization, relying instead on internally propagated states. Table 10 reports the results, where our method consistently outperforms both RNN-based memory configurations. While the Memory-Block superficially resembles recurrent architectures because it reuses temporal context, its design differs in two critical aspects: (i) it avoids the sequential gating of LSTMs/GRUs, enabling higher parallelism and (ii) it fuses two complementary sources of information, current sparse inputs and VQ-VAE code-vectors (autoregressively providing full-pose priors), yielding richer context than standard recurrence.

Placement of Memory Blocks in Backbone: To further assess the impact of memory blocks on the performance of our model, we conduct an ablation study focusing on their placement within the backbone. Specifically, in our model, memory blocks are incorporated in some of the layers of backbone for computational efficiency. In particular, we introduce Memory-blocks in 2nd, 4th, 6th, and/or 8th layers of the backbone. The results presented in Table 9 illustrate the effect of progressively adding memory blocks to the backbone via different layers. From the results, we observe a constant improvement in performance as more memory block outputs are introduced into deeper layers of the backbone. Notably, incorporating memory blocks at all four layers obtains the best scores for MPJRE, MPJPE, and MPJVE metrics. However, a slightly lower Jitter score is obtained when memory block is introduced for the 8th layer.

2. Model Size and Complexity

Model Size and Model Complexity: It is crucial to consider the model size and model complexity (i.e., number of FLOPs) to better assess the computational efficiency and memory footprint. The results in Table 11 highlight the trade-offs between these factors across state-of-the-art methods. Notably, *Mem-MLP* demonstrates a strong balance across all three metrics. With a model size of 15.38 MB, it is the second most compact among the others, slightly higher than AGRoL-MLP [13]. This compactness positively affects the architecture, which minimizes storage requirements. Additionally, *Mem-MLP* achieves the lowest FLOP (0.25G), outperforming all other methods, including AvatarPoser [19], AGRoL-MLP [13], and MANIKIN-S [20], while being significantly more efficient than diffusion-based models such as AGRoL-Diffusion [13], BoDiffusion [9], and MMD [12].

Regarding the number of parameters, *Mem-MLP* (4.10M) is the second lowest after AGRoL-MLP [13] (3.74M) and is notably lighter than diffusion-based models, such as MMD [12] (101.72M) BoDiffusion [9] (21.94M)

Memory Block Layer(s) in Backbone	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓
2	2.63	3.17	15.12	6.82
$2, 4$	2.63	3.20	15.57	6.66
$2, 4, 6$	2.60	3.14	15.04	6.37
$2, 4, 6, 8$	**2.57**	**3.08**	**15.02**	**6.03**

Table 9. Impact of placement of memory blocks in backbone. The performance for all metrics improves when memory blocks are incorporated in the deeper layers of our backbone. The best results are in **bold**.

Method	MPJRE ↓	MPJPE ↓	MPJVE ↓	Jitter ↓	FLOPs(G) ↓
GRU-MB	2.63	3.50	19.11	8.77	0.24
LSTM-MB	2.59	3.29	17.40	7.72	0.25
Ours	2.57	3.08	15.02	6.03	0.25

Table 10. Assessment of LSTM and GRU as Alternatives to the Existing Memory Block.

Method	FLOPs ↓	Size (MB) ↓	#Params (M) ↓
AvatarPoser [19]	0.33G	15.73	4.12
AGRoL-MLP [13]	0.88G	**14.25**	**3.74**
AGRoL-Diffusion [13]	1.00G	28.54	7.48
AvatarJLM [46]	4.64G	243.41	63.81
BoDiffusion [9]	0.46G	83.70	21.94
SAGE-Net [15]	4.10G	458.79	120.27
EgoPoser [21]	0.33G	15.77	4.12
MMD [12]	7.98G	853.47	101.72
MANIKIN-S [20]	0.33G	–	4.12
MANIKIN-L [20]	4.64G	–	63.8
MANIKIN-LN [20]	4.64G	–	63.8
HDM-Poser [10]	0.38G	66.55	9.55
Mem-MLP-41 (Ours)	**0.25**G	<u>15.38</u>	<u>4.10</u>
Mem-MLP-60 (Ours)	0.38G	16.08	4.59

Table 11. Model size and complexity (FLOPs, parameters, and memory).

Method	CPU (ms) $\downarrow$	CPU (FPS) $\uparrow$
AvatarPoser [19]	**6.0**	**72.0**
AGRoL-MLP [13]	30.1	26.0
AGRoL-Diffusion [13]	290.5	3.5
Mem-MLP-41 (Ours)	<u>6.8</u>	**72.0**
Mem-MLP-60 (Ours)	8.5	65.0

Table 12. On-device inference time (CPU). Lower is better for time (ms), higher is better for FPS.

and SAGE-Net [15] (120.27M). Additionally, MANIKIN [20] achieves competing predictive accuracy with computational cost. Overall, these results indicate that our *Mem-MLP* model achieves the optimal trade-off between efficiency and model complexity, making it well-suited for resource-constrained environments. Its reduced FLOPs and its compact size suggest that it can operate efficiently in real-time applications with minimal computational overhead.

On-Device Inference Speed: We deployed a subset of methods (i.e., AvatarPoser [19], AGRoL-MLP [13], AGRoL-diffusion [13], and ours) on the Meta Quest 3 to evaluate their on-device performance, Table 12. This allows us to illustrate how each method can operate in real-world, resource-constrained environments. As previously noted, *Mem-MLP-41* and AvatarPoser [19] achieve real-time performance, running at $6.8ms$ and $6.0ms$ on-device, respectively. Moreover, for *Mem-MLP-60*, we observe that its execution time slightly increases to $8.5ms$, maintaining near real-time performance. Nevertheless, this model remains a viable option for the applications where slightly higher latency is acceptable, when the methods are integrated with frame-interpolation techniques or a better HMD hardware with improved processing capabilities. In the deployment stage, all methods are converted to ONNX models and executed on a single CPU.

3. Qualitative Results

We visualize the generation results of different methods for various motion types. The results are illustrated in Figs. 3, 4, 5, 6, 7. In particular, we report heat-map errors that ease the visualization errors for the different parts of human-body.

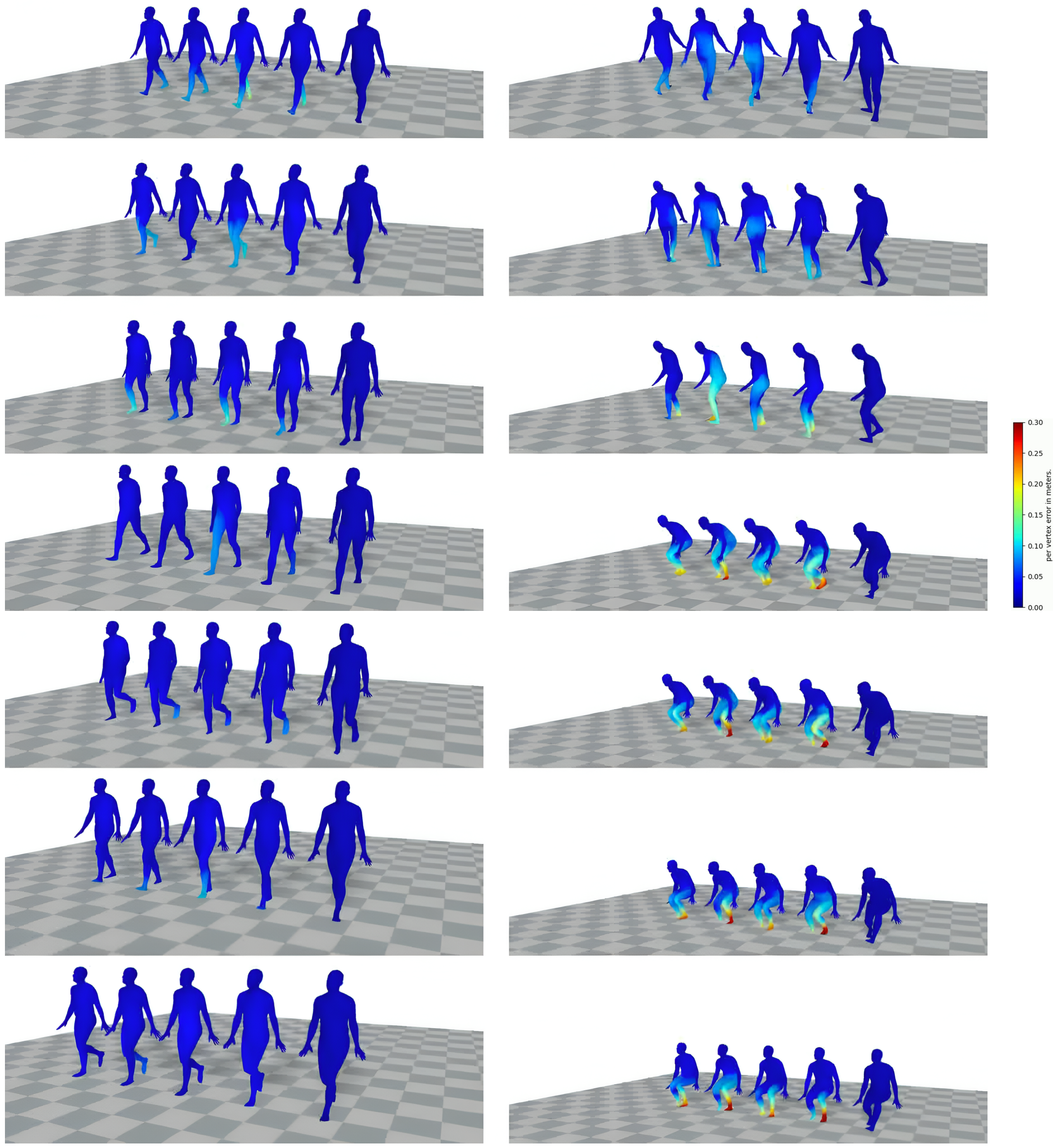

Figure 3. Results on normal walk and sitting motions. Heat-map errors are plotted for different methods. From left to right: Avatar-JLM [46], AGRoL-Diffusion [13], SAGE-Net [15], ours (*Mem-MLP-41*) and the ground-truth.

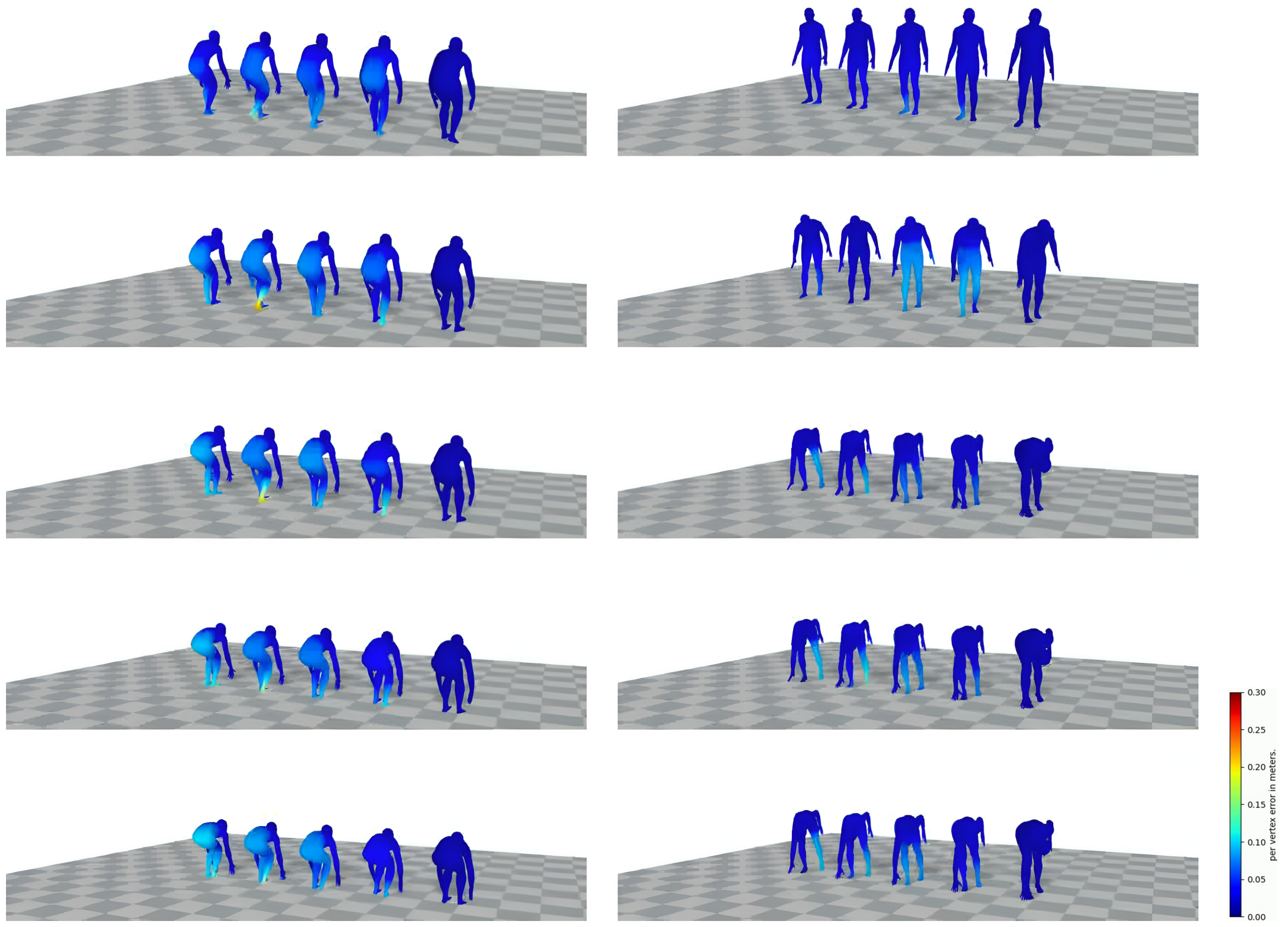

Figure 4. Results on lifting and touching ground motions. Heat-map errors are plotted for different methods. From left to right: AvatarJLM [46], AGRoL-Diffusion [13], SAGE-Net [15], ours (*Mem-MLP-41*) and the ground-truth.

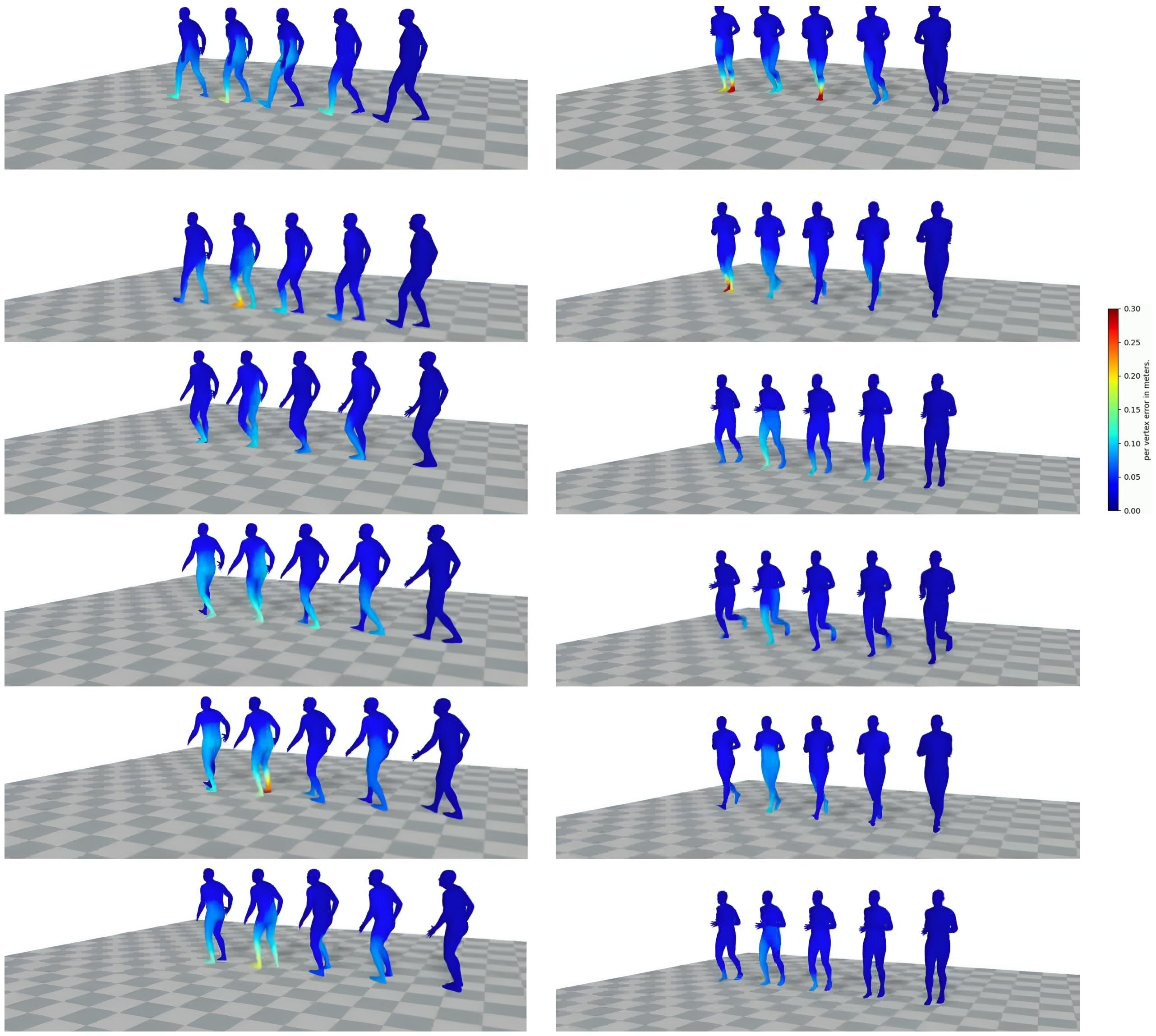

Figure 5. Results on backward walk and normal running motions. Heat-map errors are plotted for different methods. From left to right: AvatarJLM [46], AGRoL-Diffusion [13], SAGE-Net [15], ours (*Mem-MLP-41*) and the ground-truth.

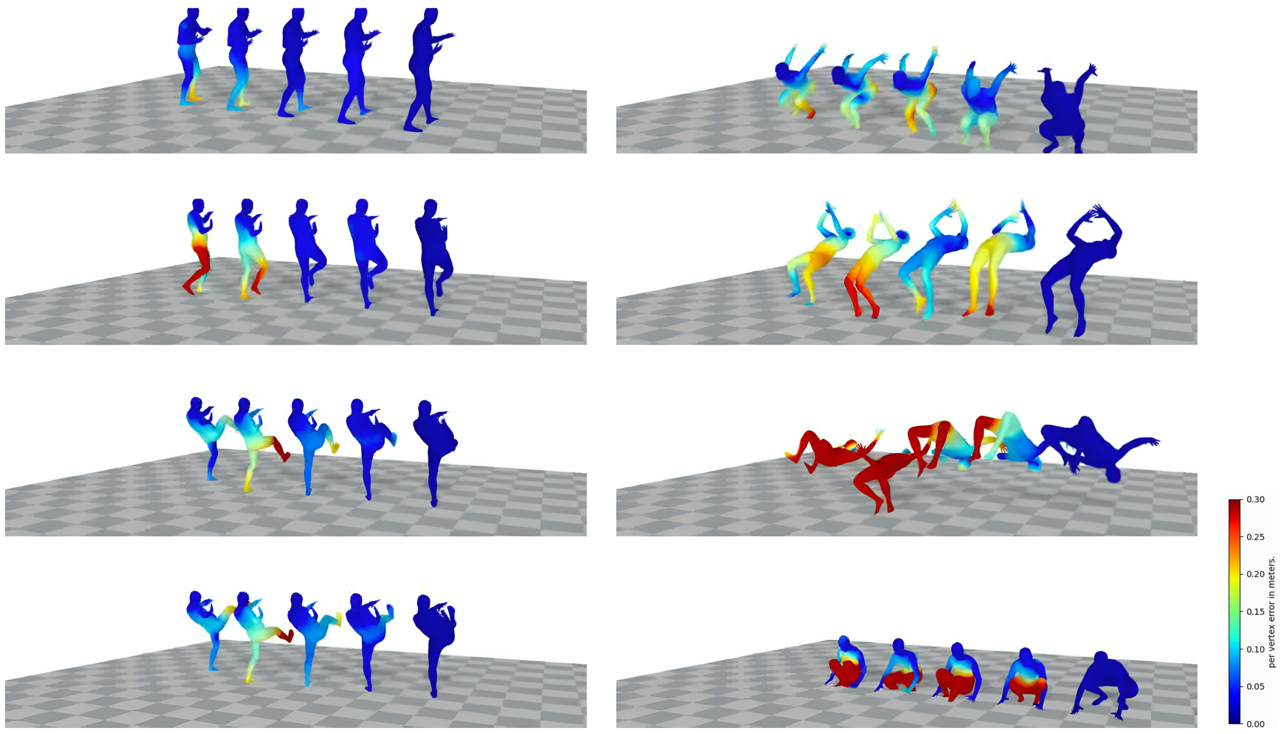

Figure 6. Results on kicking and back-flip ground motions. Heat-map errors are plotted for different methods. From left to right: AvatarJLM [46], AGRoL-Diffusion [13], SAGE-Net [15], ours (*Mem-MLP-41*) and the ground-truth.

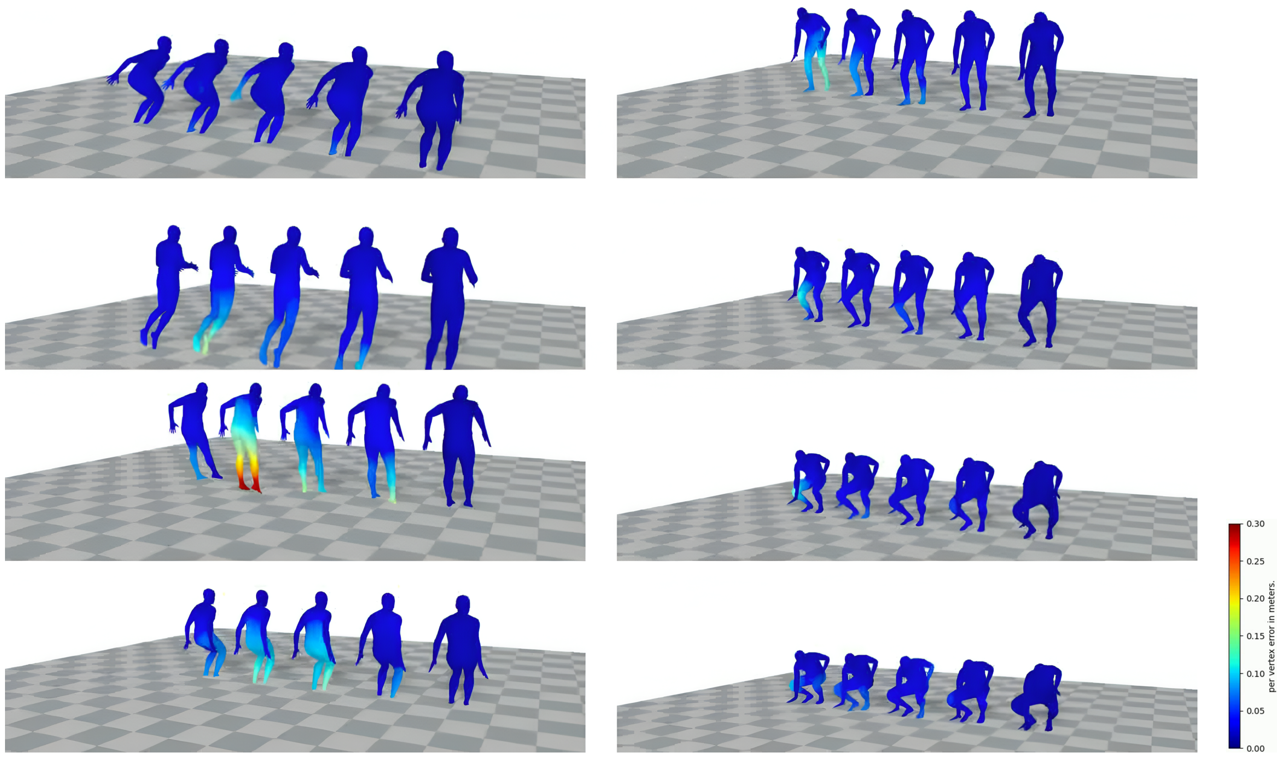

Figure 7. Results on jumping and squating motions. Heat-map errors are plotted for different methods. From left to right: AvatarJLM [46], AGRoL-Diffusion [13], SAGE-Net [15], ours (*Mem-MLP-41*) and the ground-truth.