FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Jan Waltl

Autoregressive action-conditioned 3D human motion synthesis using latent discrete codes

Department of Software and Computer Science Education

Supervisor of the master thesis: Dr. Ing. Josef Šivic
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2022

[illegible]

In Prague, July 21, 2022

Author’s signature

I would like to sincerely thank my supervisor Dr. Ing. Josef Šivic for offering me the opportunity to participate in this very interesting field of research and my gratitude goes to Mgr. Tomáš Souček for choosing to consult on this thesis. For your invaluable guidance, never-ending patience, plentiful advice whenever I needed it, and for continuously supporting me and encouraging me throughout this journey, I do thank you both.

I would also like to thank Czech Institute of Informatics, Robotics and Cybernetics(CIIRC ČVUT) for giving me access to their computational cluster.

Title: Autoregressive action-conditioned 3D human motion synthesis using latent discrete codes

Author: Jan Waltl

Department: Department of Software and Computer Science Education

Supervisor: Dr. Ing. Josef Šivic,

Abstract: In this work, we introduce a new method of 3D human motion synthesis conditioned on categorical action labels such as “running“ or “toe touch“. Motivated by recent results in the text-to-image generation domain, we investigate the possibility of using discrete latent codes to generate realistic human animation sequences in contrast to the continuous representation of the current methods. Compared to the state-of-the-art ACTOR method, ours is not limited in length while still generating high-fidelity, diverse motions with the added ability to generate a continuation motion to the starting motion sequence. With the sliding window autoregressive decoding approach, we retain reasonable generation speeds. Furthermore, thanks to two-stage training, future models can be pre-trained on larger unlabeled datasets and then conditioned on possibly different contexts. We demonstrate the validity of our method on the UESTC dataset and achieve better quantitative results compared to ACTOR as well as synthesising high-quality human motions comparable to the original dataset.

Keywords: motion synthesis deep learning discrete representation

Contents

1. Introduction

1.1 Motivation and objectives

Generating 3D realistic human motions is a long-standing challenge, with the rise of special effects in movies, whole departments of animation experts have been dedicated to producing high-quality motions by hand for movies or even more recently in the field of video games. With the invention of motion-capture technology, the actors could breathe a new life into the animations in addition to lending characters their voice. It also brought the possibility of creating high-quality motion datasets [1, 2] and with the expanse of machine learning and neural networks, considerable research has been invested into putting them to use [3, 4, 5, 6, 7].

But what makes the problem difficult? If we consider motions like *throwing a ball*, *picking up a mug*, or just simple *walking*, each of us can imagine doing these activities but if we actually perform them, likely none of us will do so in exactly the same way. For example, the majority of people are right-handed and therefore will likely use their preferred hand to throw the ball. Some people will naturally walk more slowly or move their legs in a slightly different fashion. There is certain natural ambiguity when specifying each motion even if we try to be more rigorous. Another issue is the hidden complexity of input signals to the muscles, their cooperation to do just something as natural as taking a step or smiling, thus preventing both simple, exact, and rigorous description of motions, perhaps in the form of instructions in some new programming language. The task is made even harder because humans are exceptionally good at spotting even slight inconsistencies present in human-like artificially-made motions, the phenomenon referred to in the literature as *the uncanny valley* [8].

Why is this problem even worth solving? Because an existing solution could have a wide range of applications. Simplifying the mentioned animation of movie and game characters by for example generating the simpler or most prominent part of the motion and leaving it to the animator to fill in the details and mannerisms which make the character unique.

Similarly, it is an appealing thought to provide a few key frames or locations where a game character should be and let the computer fill in the rest automatically [9, 10], even including interactions with the scene[11][12][13].

The objective of this thesis is the synthesis of realistic and diverse human motions based on categorical labels i.e. motions from a fixed set of possible categories like *stretching* or *throwing*. Crucially, we leverage the advances in the image-generation [14, 15] domain with the usage of discrete representations [16] and investigate its potential to improve motion synthesis.

Contributions

- Successful adaptation of discrete architectures for representation of motions opening new research possibilities for downstream tasks.

- Autoregressive architecture synthesises high-quality conditioned human motions of potentially unlimited length while preserving their diversity and retaining their details.

We validate our ideas on a dataset of categorical human motions and achieve state of the art results.

1.2 Structure of the thesis

In the next (Chapter 2), we discuss the related work and present the models we base our work on. Chapter 3 describes our proposed motion generating model in detail, its variants and how it is trained. In particular, we define two-stage model consisting of: 1) motion discretization using a motion autoencoder model responsible for converting between a motion and its learned discrete representation. 2) motion prediction stage with a motion predictor model capable of synthesizing motions.

After that, Chapter 4 contains the results of experiments and ablations. It follows the similar structure of Chapter 3 but with the additional description of the common experimental setup, including the datasets and metrics which we used.

In Section 4.3, we focus on the autoencoder, through ablations studies we try to understand its hyper-parameters, which variants are viable, and what are the properties of generated token datasets for the second phase. We conclude the section with the summarization of the best autoencoders model we have found as those will subsequently be used to train the motion predictors.

In Section 4.4, we first present the results of training the motion predictor and its ablation studies including the effect of different used autoencoders as well as decoding strategies. After that, we follow by comparing our results to the state-of-the-art results. Throughout the chapter, we show the outputs in many figures but due to the nature of our outputs, we want to emphasize the supplemental videos we have made to better demonstrate our results.

In the end, in Chapter 5, we summarize our work, its achievements, limitations, and potential future directions of research.

To this thesis we attach:

1. Source codes of all our experiments, also available from `https://github.com/janwaltl/master-thesis` with the instructions on how to reproduce the presented results.

2. Supplemental videos, alternatively also available at the same link.

2. Related Work

In this chapter, we first summarize the existing literature concerning conditioned motion generation (Section 2.1) and then in Section 2.2 we introduce the previous works we have based our model on. We focus mostly on text-based conditioning but there are other interesting options such as generating motions based on music as done in [6, 17].

2.1 Motion generation

Text2Action: Generative Adversarial Synthesis from Language to Action Text2Action[18] is a sequence-to-sequence model capable of generation upper-body motions from language description of particular actions. The model is based on GAN[19] adversarial approach, consisting of RNN[20] discriminator, a generator operating on sequences. An extra encoder for conditioning on the text using LSTM is employed. The text is encoded to a hidden sequence of feature vectors h. To generate a motion conditioned on a sentence, the encoder first processes the input text, the hidden sequence h is then fed together with a sequence of samples from normal distribution into the RNN generator which autoregressively generates the motion. The discriminator (also RNN-based) takes as input the hidden sequence h and a motion, its goal is to output whether the sequence comes from the dataset (real data) or has been generated artificially. Note that both generator and discriminator also employ the attention[21] mechanism to attend to the whole hidden sequence.

In order to achieve the best performance, the authors claim to have been forced to train the model in two stages, first an RNN autoencoder between language sentence and motion sequence is pre-trained using cyclic loss. The encoding part then serves as the mentioned encoder in the next stage where both discriminator and generator are trained using the well-known min-max game loss[19].

One notable part of the paper is training the model on motions automatically extracted from a large-scale video dataset.

Language2Pose: Natural Language Grounded Pose Forecasting. Another model working with language sentences is Language2Pose model[4]. Its authors chose to train a separate autoencoder for language and motions which share a joint continuous embedding space, one can thus encode text and decode a motion and vice versa. The embedding space is of fixed size, regardless of the lengths of motions or sentences, LSTM[22] architecture is employed for language with pre-trained word embeddings while GRU is used for processing the motions. The models are trained jointly on KIT dataset[1] which consists of motion-captured motions with short sentences describing them. Overview of this method is shown in Figure 2.1.

Action2Motion: Conditioned Generation of 3D Human Motions Action2Motion is a recent paper[5] introducing a generative action-conditioned model for motion generation with an emphasis on the multi-modality of generated motions. Variational autoencoder(VAE) is employed together with a novel approach

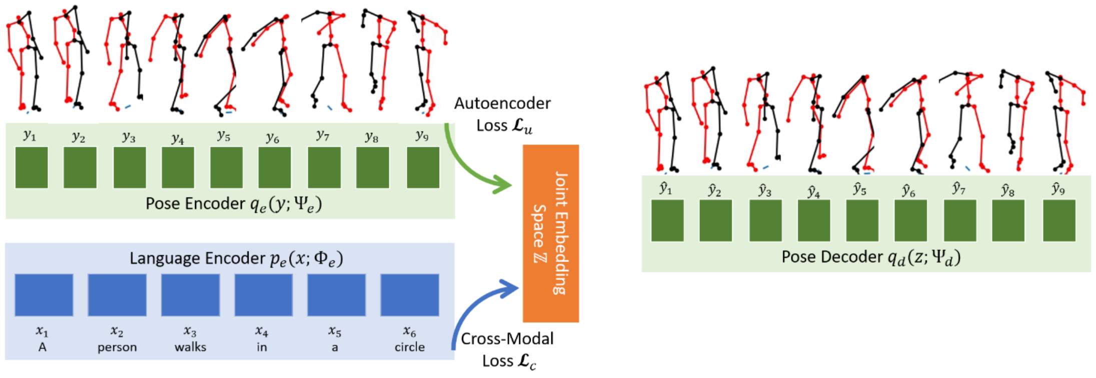

y_1 y_2 y_3 y_4 y_5 y_6 y_7 y_8 y_9
Pose Encoder $q_e(y; \Psi_e)$
Language Encoder $p_e(x; \Phi_e)$
x_1 A
x_2 person
x_3 walks
x_4 in
x_5 a
x_6 circle
Autoencoder Loss $\mathcal{L}_u$
Joint Embedding Space $\mathbb{Z}$
Cross-Modal Loss $\mathcal{L}_c$
$\hat{y}_1$ $\hat{y}_2$ $\hat{y}_3$ $\hat{y}_4$ $\hat{y}_5$ $\hat{y}_6$ $\hat{y}_7$ $\hat{y}_8$ $\hat{y}_9$
Pose Decoder $q_d(z; \Psi_d)$

Figure 2.1: **Overview of Language2Pose model** - This sequence-to-sequence model[4] takes a motion $y_1, \dots, y_9$ and the corresponding text annotation $x_1, \dots, x_6$ and passes them through two encoder, each generating a latent vector of the same dimensions (irrespective of the inputs' lengths). There is also a decoder which can reconstruct a motion given a vector from the latent embedding space. During and after training one can freely choose between which decoder is used and thus the model has the ability to either reconstruct a motion or generate a motion solely based on the textual input. The figure is from [4], ©2019 IEEE.

involving the use of Lie Algebra for representing motions instead of the more traditional joint coordinates.

The autoencoder encodes each frame of a motion, together with an action category label, into latent continuous representation constrained by a normal distribution. GRU[23] model is jointly trained on this latent space to auto-regressively generate a motion on a frame-by-frame basis.

This model is very close to what we use in our model, but there are a few key distinctions. Our autoencoder is not aware of any categories and therefore works on unlabeled data. Furthermore, we leverage the recent approaches in both sequence generation by using Transformer[21] model instead of GRU and discrete representation instead of a continuous one.

The authors also compiled together a new labelled 3D motion dataset denoted Human12 as well as introduced important qualitative metrics serving as the benchmark for the motion synthesis models. ACTOR[3] and we use these metrics for comparing our models.

ACTOR: Action-Conditioned 3D Human Motion Synthesis with Transformer VAE ACTOR [3] is a Transformer-based[21] model (in contrast to previous RNN-based approaches) which differs from other methods described so far because it is not autoregressive. Instead, a VAE with a Transformer-based encoder, and decoder is used to obtain sentence-level latent representation. Furthermore, the latent variables are conditioned on the action category, allowing the conditional generation of motions. The authors show the model achieves the state of the art results for conditioned motion synthesis. See an overview of the method in Figure 2.2. Because the model is operating on whole motions, the generation is fast but also limited in length by the length chosen for training. Although the authors also try training with variable-length motions, improving the generalization to other lengths, the model is by its nature fixed-length, it is

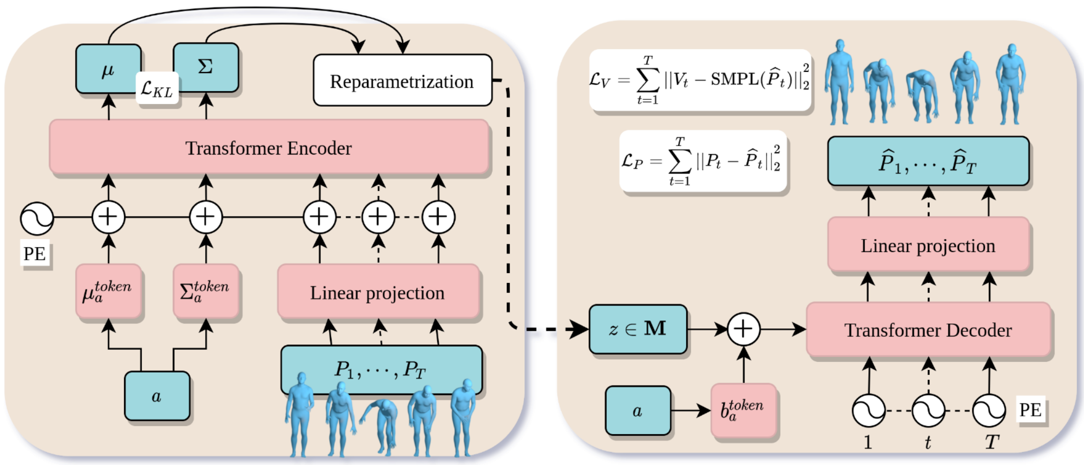


Figure 2.2: **Overview of ACTOR model** - ACTOR[3] is a transformer-based variational autoencoder used for encoding and generating motions. Motion $P_1, \ldots, P_T$ and the corresponding category label c is processed by the encoding transformer (left side), outputting parameters (μ, Σ) of the latent Gaussian representation. The decoder (right side) then takes this sequence-level representation, samples from the implied distribution plus again the category label and another transformer decoder reconstructs the original motion. After training, the decoder serves as a generator with z sampled from the normal distribution. The figure is from [3], ©2021 IEEE.

not possible to take a starting motion and "finish" it.

The paper mentions a possible modification to the decoder making it autoregressive which the authors have explored, but they report mixed results with decoding issues. Despite this, we strongly believe the auto-regressiveness is a very important feature and state of the art models in conditioned language [24, 25] and image generation [15, 14] rely on it which proves its value.

Therefore we aim to develop a model which does not suffer from this limit and is both fast at decoding and achieves comparable or better results than ACTOR. The authors used a new motion dataset from extracted videos in UESTC[26] which we will also use as we consider this paper closest to ours and the main competitor.

2.2 Discrete autoencoders and image generation

Recently, discrete autoencoders gained a prominent role in image generation [15, 14] generating state of the art results. The autoencoder model has been first introduced in [16], allowing encoding data into discrete latent variables and training a second model to learn and generate samples from the prior distribution of tokens. This is in contrast with the original VAE [27] using continuous latent space with samples constrained to normal distribution. The next two papers serve as a basis for our work, both focusing on using VQ-VAE to generate high-resolution images conditioned on text, or even other images and masks. Both work autoregressively and show very impressive results. Therefore we will adapt the ideas in them to the motion synthesis domain. It is worth noting that both models are fairly large with hundreds of millions of parameters, our models will be much smaller in comparison.

VQ-VAE In contrast to VAE [27], VQ-VAE [16] model employs discrete latent space, the output from the encoder consists of feature vectors which are independently quantized by being replaced by their nearest neighbour (2-norm metric) from a trainable fixed-size codebook of vectors. The authors focus on the image generation domain in their experiments and show promising results over a wide variety of tasks. With the fully trained autoencoder, the images are first encoded into their discrete representation - a sequence of tokens (indices into the codebook). Then a second model is used to learn the prior distribution of these tokens, optionally conditioned on extra inputs, allowing for generating images based on text, audio, or video frames.

The authors later improve upon their model in [28] and present VQ-VAE-2 with hierarchical multi-scale latent space.

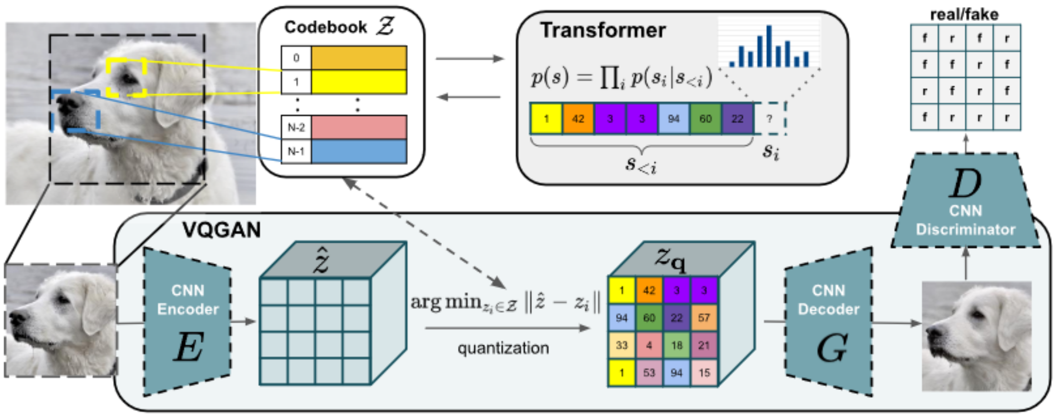


Figure 2.3: **Overview of the method from Taming transformers** - This model works in two stages. First, a VQGAN network is trained for the reconstruction of images, they are passed through convolutional layers, discretized(referred to as quantized) into tokens z_q and subsequently reconstructed by the decoder. During this an extra codebook with trainable parameters is learned. A GAN-based[19] training approach is used rather than a simple mean squared error loss on pixels. Second, a transformer-based generative network is then used to learn the distribution of the tokens and autoregressively generate them conditioned on some context information, after which the decoder can decode them back into an image. The model operates on patches from the images and the transformer uses a sliding window approach to generate high-resolution images. The figure is from [14], ©2021 IEEE.

Taming Transformers for High-Resolution Image Synthesis A paper from 2020[14] focuses on generating high-resolution images. In order to model long-range dependencies, an attention-based architecture is employed but the authors note that generating such high-resolution images in a pixel-wise fashion is not very feasible or at least it is computationally very expensive. Therefore they employ VQ-VAE to learn rich, discrete, and much smaller representations of images and the generator then operates on this latent space, although still in a sliding-window approach. See overview of this method in Figure 2.3

Authors improve upon the image techniques presented in VQ-VAE and employ custom GAN-based reconstruction loss, enhancing the compression and the level of detail of reconstructed images.

DALL-E and DALL-E2 A set of papers [15] and [29] from OpenAI, describe DALL-E and recently DALL-E 2, the former uses VQ-VAE to generate images from natural language, achieving state of the art results.

The model itself is very close to the one used in Taming Transformers, but the authors opt in for gumbel-softmax [30] for obtaining discrete tokens in a differentiable way compared to the original nearest-neighbour approach. The conditioning and generation is done using a single transformer encoder model where the language tokens are prepended to the to-be-generated sequence.

To summarize, in this chapter we described previous work related to motion synthesis as well as recent architectures in the text-to-image domain which we adapt for motion generation.

Furthermore, as far as we are aware, all techniques related to generating motion either from categories or conditioned on language use continuous representation for pose/motion - using either raw angles, positions, 6D, or some processed embeddings of those. Our work is thus the first which employs VQ-VAE and discrete representation for purpose of motion synthesis.

3. Motion Synthesis Using Discrete Latent Codes

In this chapter, we describe our model for autoregressive motion synthesis conditioned on category labels. We leverage the success of DALL-E [15] and Taming Transformers [14] models in image domains and apply them to motion synthesis. We propose a few alternative architectures for parts of the model which we believe should be explored, after that, we describe their training procedure and end the chapter with a short summarization.

Problem formulation

We are given given motion context in the form of a category label c, perhaps *running* or *left stretch* and a starting motion M_0 consisting of frames (poses) $\{P_0, \ldots, P_k\}$. The motion can be a simple one-frame standing pose or any other motion we should smoothly follow up on, or can even be empty if required.

Our task is to generate a continuation motion $M_G = \{P_{k+1}, \ldots, P_{k+n}\}$ such that the generated sequence matches the context c. The task is shown in Figure 3.1. The canonical format for representing human poses is a kinematic tree which captures the underlying motion of the human skeleton and its joints. Each node in this tree represents a joint and its orientation can be given for example by a 3x3 rotation matrix. As the representation, we choose to use SMPL model [31] (its 24-joint variant) as the representation of each pose due to abundance of datasets [26, 2] in this format, specifically we also [3] convert each joint orientation to 6D representation [32] because it preserves continuity. Thus each pose becomes $P_i \in \mathbb{R}^{24\times 6}$.

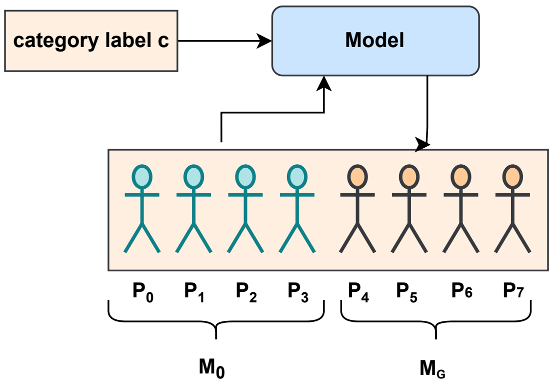


Figure 3.1: **Problem formulation** - Model synthesises a follow-up motion M_G based on the initial motion M_0 and the category label c.

Proposed model

Our proposed model builds upon [14, 15] – to first learn a rich discrete representation of motions as sequences of tokens; then we train a generative transformer over these sequences to learn their prior distribution. The learned representation allows us to then transform any generated sequence back into continuous motion.

In detail, we separate our model in two standalone modules (Figure 3.2). **(1) motion discretization** (Section 3.1) which converts between a continuous motion M and its discrete representation $\widehat{M_0}$. This stage is accomplished by training a *motion autoencoder*. **(2) motion prediction** (Section 3.2) takes as inputs the category label c, starting motion $\widehat{M_0}$, discretized first by the autoencoder, and generates follow-up motion $\widehat{M}$ in an auto-regressive way. A transformer [21] encoder model is used for this, we call this model *motion predictor*.

The practical reasons for separating the model into two separately-trained modules as opposed to joint training are two-fold:

- Very unstable gradients at the beginning of the training caused by the latent distribution being essentially random, also experienced by [14] and [15], making this separation a necessity.

- This approach allows pre-training motion discretization on larger, unlabeled datasets possibly automatically extracted from large video datasets. The conditioned generation can then be trained on smaller labelled subsets. Another option is the ability to use different motion contexts for conditioning the tokens.

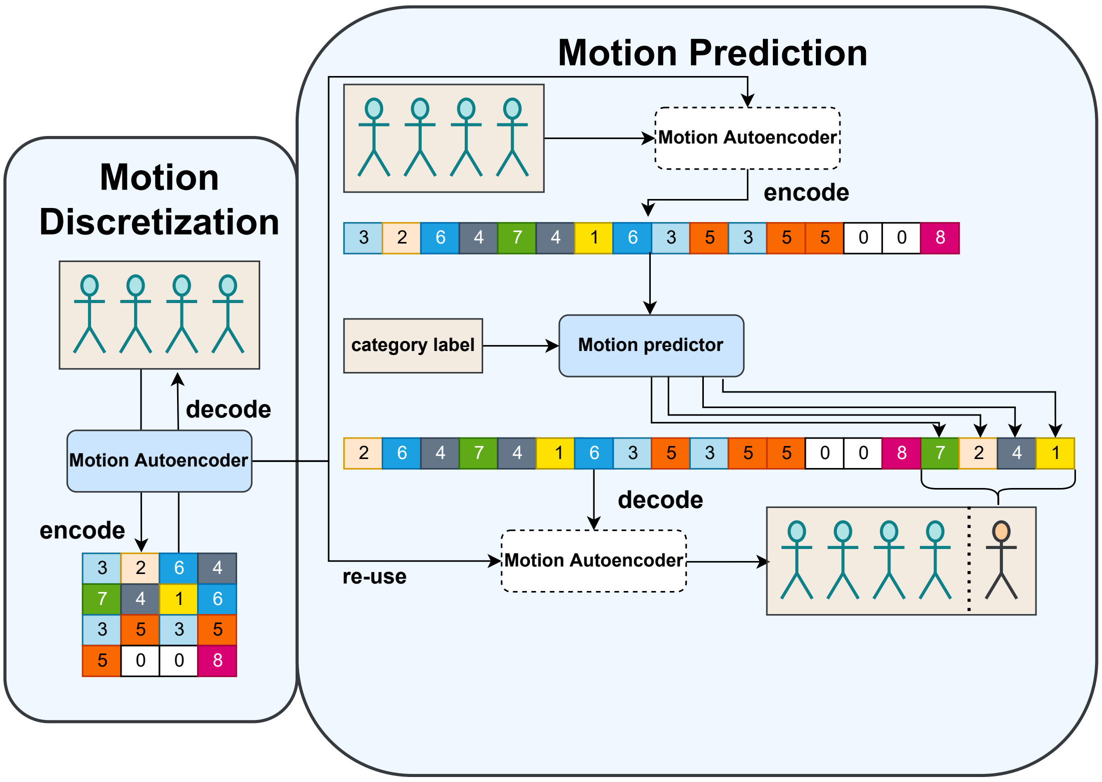


Figure 3.2: **Proposed architecture separated in two stages** - First, we train an autoencoder to learn a mapping between continuous motions and discrete sequences of tokens - the *motion discretization* stage. Then, in the *motion prediction*, we train a motion predictor(Transformer model) to auto-regressively generate motions given a motion context in the form of a label c. At last, the autoencoder from the first stage is used to reconstruct the generated motions from the new tokens.

Now we will describe both modules separately in detail, including the possible alternatives.

3.1 Motion discretization

To leverage Transformer architecture operating on discrete tokens, we have to be able to transform continuous motions into discrete representations. This is achieved by training a VQ-VAE [16] autoencoder model using reconstruction losses on unlabeled data.

VQ-VAE architecture in general consists of three blocks: encoder E, discretization latent block L, and decoder D, all depicted in Figure 3.3. In our work, we consider multiple different architectures for each block and now we briefly describe their general function, the detailed descriptions of various considered architectures in the listed order are in the following sections.

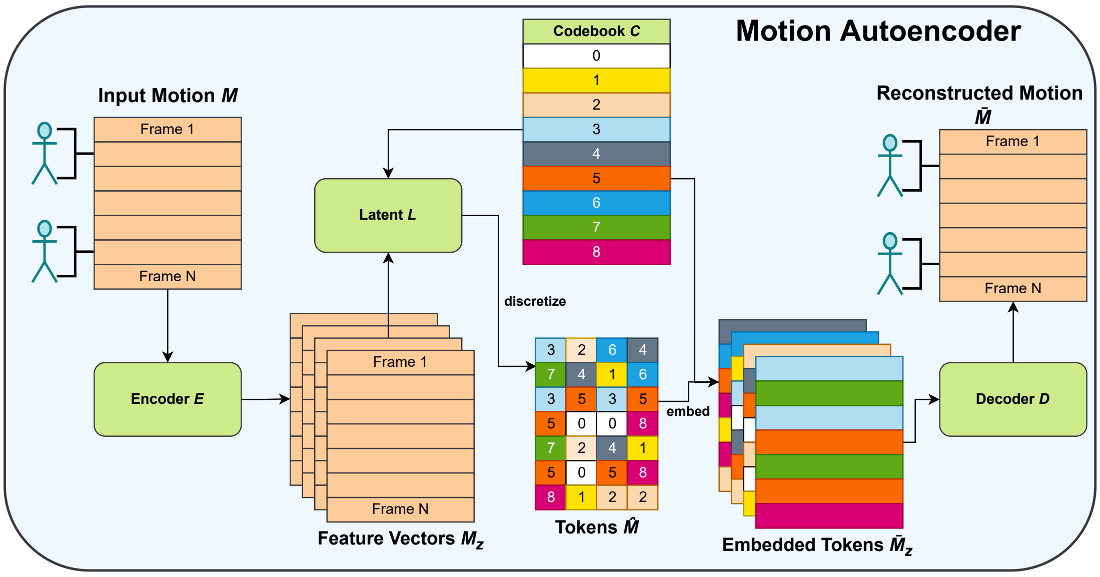


Figure 3.3: **Motion Autoencoder** - Model to map between discrete and continuous motion representation, similarly to [14].

The encoder block's input is a continuous motion in SMPL format [31], the output is a latent continuous feature vector. VQGAN[14] works with (up to) 192x192 images and encodes them into 32x32 tokens using convolution layers. ACTOR [3], on the other hand, uses fixed-sized latent z for the whole sequence in its VAE. We believe it is important and also practical to preserve the time dimension of motions, therefore we encode a motion $M \in \mathbb{R}^{f\times 24\times 6}$ into a feature vector $M_z \in \mathbb{R}^{f\times D}$ for some dimension D, while the number of frames f is preserved across the whole autoencoder model.

After encoding, the latent block performs discretization of this feature vector M_z into one-hot discrete tokens $\widehat{M} \in \mathbb{Z}_k^{f\times s\times k}$, again, we only vary parameter s which we call *cuts*, frames f are fixed, and k is the size of a hidden codebook (described shortly). Meaning that each frame of the feature vector M_z should roughly correspond to s tokens, although that is enforced only implicitly during training. $\mathbb{Z}_k$ is defined as set $\{0, \ldots, k-1\}$.

In more detail, discretization is achieved by first reshaping M_z via $(f \times D) \rightarrow (f \times s \times d)$, then the d-dimensional sub-vectors are independently discretized by the latent block into tokens and embedded using a trainable codebook $C \in \mathbb{R}^{k \times d}$[16] to obtain (after recombination of the sub-vectors) $\overline{M_z} \in \mathbb{R}^{f \times D}$. The codebook C consists k $d-$dimensional trainable code words corresponding to k possible tokens.

Decoder block takes the embedded latent representation $\overline{M_z}$ after discretization and tries to reconstruct back the original motion – outputting $\widehat{M}$. The process can be summarized as follows:

$$\begin{aligned}
&M && M \in \mathbb{R}^{f \times 24 \times 6}, && (3.1)\\
M_z &= E(M) && M_z \in \mathbb{R}^{f \times D}, && (3.2)\\
\widehat{M} &= L(M_z) && \widehat{M} \in \mathbb{Z}_k^{f \times s \times k}, && (3.3)\\
\overline{M_z} &= \widehat{M} \cdot C && C \in \mathbb{R}^{k \times d}, \overline{M_z} \in \mathbb{R}^{f \times D}. && (3.4)\\
\overline{M} &= D(\overline{M_z}) && \widehat{M} \in \mathbb{R}^{f \times 24 \times 6}, && (3.5)
\end{aligned}$$

Note that $\widehat{M}$ is the output passes to the motion predictor, while $\overline{M_z}$ is used only for reconstruction during training and for decoding the generated motions from the trained predictor.

Now, in the following four sections, we look in more detail at the considered block variants. In the first two, we describe the two possible variants used for the encoder and decoder blocks. In the second two, we explore the implementations for the latent space.

3.1.1 Convolutional autoencoder variant

The first possible architecture for the encoder, decoder blocks for motion discretization we consider is built from 1D convolutional blocks. It is analogous to Conv2D used by [14]. A desirable property of CNN networks is the enforced locality of tokens w.r.t frames because the encoding context is proportional to and also limited by the depth of the network. That is also a limitation.

In particular, the motion $M \in \mathbb{R}^{f \times 24 \times 6}$ is reshaped first via $(f \times 24 \times 6) \rightarrow (f \times 144)$ and the last dimension is treated as channels. The motion is then passed through convolutional blocks with kernel size 3 and C channels kept constant. Wide blocks [33] are used with dropout, batch normalization [34], ReLU [35]. The blocks are connected with residual connections [36]. See left side of Figure 3.4 for precise order of layers.

It is worth noting that we use exactly the same architecture for the encoder and decoder, no deconvolutional layers are used for the decoder. We briefly considered them but found no performance gain, we believe this is due to not using the more common pooling approach with decreasing spatial dimensions compensated with wider channels and the decoder's need to reverse this process. The bottleneck of our network is the discretization, see reporting on compression ratios in cuts ablations in Section 4.3.

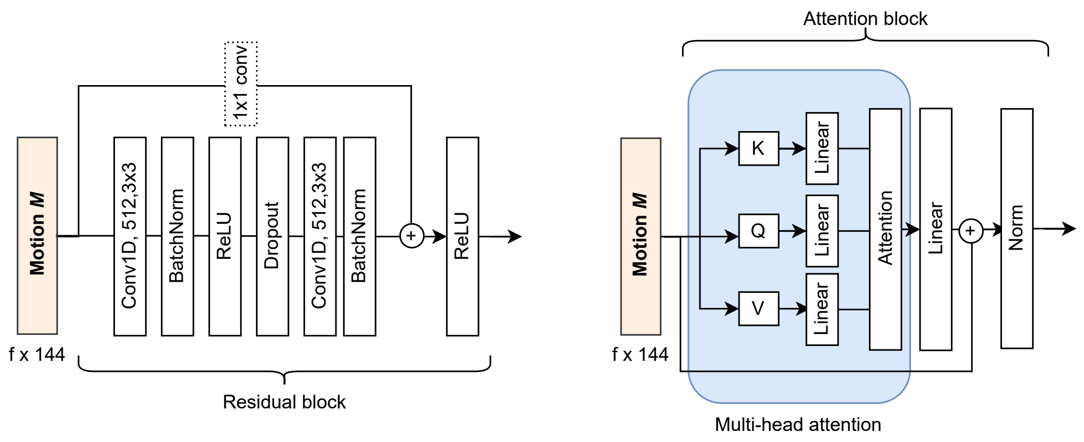


Figure 3.4: **Considered encoder and decoder architectures** - We consider convolution neural network (left) which uses wide [33] residual [36] convolutional blocks. Note the extra 1×1 convolution is used to properly resize the input in case of channel size mismatch (the case of the first layer). Alternative is attention-based model (right) for the motion autoencoder which passes the motion through multiple attention blocks [21].

3.1.2 Attention-based autoencoder variant

We would like to be able to encode as complex motions as possible and by *complex* we usually mean exactly those motions where there is non-local non-trivial context involved and therefore we might benefit from an architecture with non-local connections.

Attention-based models excel at taking into account non-local context (and we also take advantage of that during the motion generation phase). Therefore we propose to also use attention-based architecture for motion discretization in the form of transformer encoder model [21].

To be precise, we take a reshaped motion $M \in \mathbb{R}^{f \times 144}$, and project it to embedding dimension d with a help of an ordinary linear dense layer. After that, we treat it like f already embedded tokens, add positional encoding, and pass it through multiple attention layers of the Transformer encoder model [21] to obtain the sought-after feature vector M_z. The process is illustrated in the right portion of Figure 3.4.

The decoder is exactly the same. Although its input comes from discrete tokens and therefore we could have learnt a special token embedding, distinct from the code book, we did not find any further gains doing that for the reconstruction. We however do investigate the idea of reusing the codebook for embedding the tokens in the motion predictor model later.

A question we have asked ourselves is whether any frame masking should be used in either this or the convolutional approach. We believe that is a valid question because, on one hand, the future context can aid reconstruction, on the other, our final goal is auto-regressive motion generation in which we have to generate tokens one step at a time. We would thus have this disparity between the usage of tokens for reconstruction and generation. Therefore it might be worthwhile to consider enforcing this property also during discretization.

During our experiments, we have found that the attention model does take the

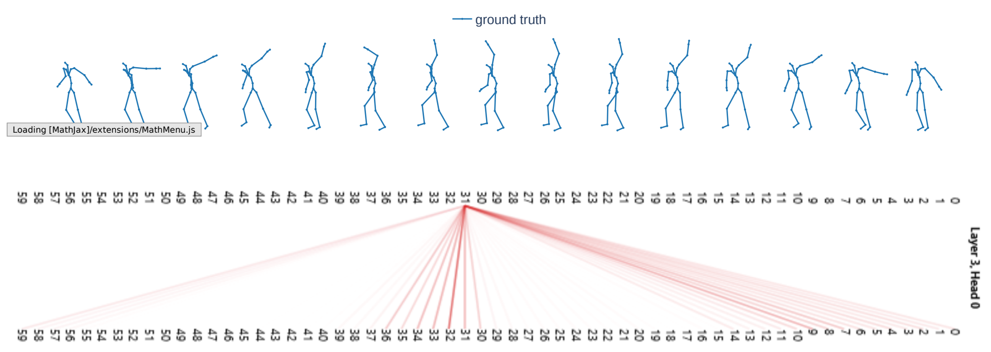


Figure 3.5: **Future context is important for motions** - Attention activation weights of one head of the last layer in the trained encoder, we can see that to encode frame 31 (the middle of the motion sequence), the model finds it important to look at the beginning and end of the actual performed *left stretch* movement in the motion. Although most weights in other layers appear to be quite generic, there are some which tend to begin and end of each performed motion or other notable actions.

advantage of meaningful future context in certain motions as can be shown in Figure 3.5, quantitative results are found in the next chapter. Therefore we believe it is desirable to *not* mask any future tokens, doing so also leads to subjectively more discontinuous and choppy reconstructions.

3.1.3 Discretization using nearest-neighbour

The original VQ-VAE work [16] proposes a simple method of discretization by assigning each feature vector the index of the closest vector from the codebook and thus embedding is replacing one vector with its closest counterpart in the codebook. The method is illustrated in Figure 3.3.

In particular, for a given feature sub-vector $v \in \mathbb{R}^d$ and code words $v_0, \ldots, v_{k-1}$, the corresponding token index is given by

$$i = \operatorname{argmin}_{j \in \mathbb{Z}_k} \|v_j - v\|_2$$

Although this operation is not differentiable, the model can still be trained with gradient descent as the authors leveraged a simple gradient-copy trick, see Section 3.3.1.

3.1.4 Discretization using gumbel-softmax

DALL-E [15] considers another approach for discretization in their dVAE autoencoder. They first re-scale the feature subvectors to match the size of the codebook k (not the dimension of the code words) and then treat the values as logits of the probability distribution for sampling from the codebook. To avoid the problem with the non-differentiable backward pass, category reparametrization via gumbel-softmax [30] is used as gumbel-softmax allows differentiable sampling from discrete distributions with only an extra hyper-parameter τ annealed to closer to 0 over the course of training.

This fundamentally changes the autoencoder in multiple ways compared to the nearest-neighbour approach from [16]:

- Autoencoder is no longer deterministic, instead, sampling is employed and one motion can have multiple discrete representations to which the decoder might produce different reconstructions.
- Although the distributions of individual tokens in each frame of the motion as a whole are correlated – controlled by the encoder – the tokens are sampled independently of each other, even in the same frame. This has an impact on the motion prediction model because it results in higher, more uniform codebook usage than the nearest neighbour approach.

3.2 Motion predictor

The second part of our model is to use the tokens obtained from a trained motion autoencoder for predicting the follow-up motion based on the supplied motion context.

Following the model proposed by [14], we train a transformer encoder model [21] to auto-regressively generate the tokens. The discretized motion is two-dimensional (not counting one-hot dimension) $M_k \in \mathbb{Z}_c^{f \times s \times k}$, it must therefore be unwrapped before it can be used in a transformer. We chose to use the natural order by concatenating the splits of each frame together, model overview is shown in Figure 3.6. [14] also experimented with other directions for their 2D image input.

Formally, given starting tokens $t_0, \ldots, t_n$ representing the starting motion and a category label c, the probability of generating motion given by tokens $t_0, \ldots, t_m$ for $m > n$ is modeled in standard way using chain rule as

$$p(t_0, \ldots, t_m | c) = \prod_{i=0}^{m-1} p_\theta(t_{i+1} | t_0 \ldots t_i, c)$$

for a transformer parameterized by θ. Auto-regressive decoding repeatedly samples the last token from the predicted distribution and feeds the sampled token back to the model for generation of the next token and so on.

When a sufficient length is achieved, the generated motion is obtained by passing the sampled tokens to the decoder part of the autoencoder. We use our own specialized positional embedded presented later in its own section. As for the token embeddings, we have two options - either to train a new embedding for each token or to re-use the codebook vectors as embeddings for the tokens. We try both and compare them in our experiments.

For incorporating the motion context into the model, we follow the approach in [3] and add a learned category embedding to each token, an alternative would be to use a special `SOS` (start of the sequence) token, this would also make it possible to generate motions without any starting motion. We also briefly tested this variant, see experimental section. We would like to stress that although our main focus is on using a single categorical label c for conditioning the motion on its context, the described model can be modified to accommodate other forms of contexts as was the case for [14, 15, 16] in which the authors successfully conditioned tokens representing images/videos on a whole range of different inputs, including audio, and even other images serving as content masks.

SOS-variant

We have defined our model in such a way that it requires non-zero-length starting motion, this is convenient for a long generation as we will show in Section 3.3. But if we want to compare our method to ACTOR, to make the testing fair, we have to be able to generate a motion just based on a category label. To do that, we modify the model to accept a fixed *start of sentence* (SOS) token and train the model in the same way.

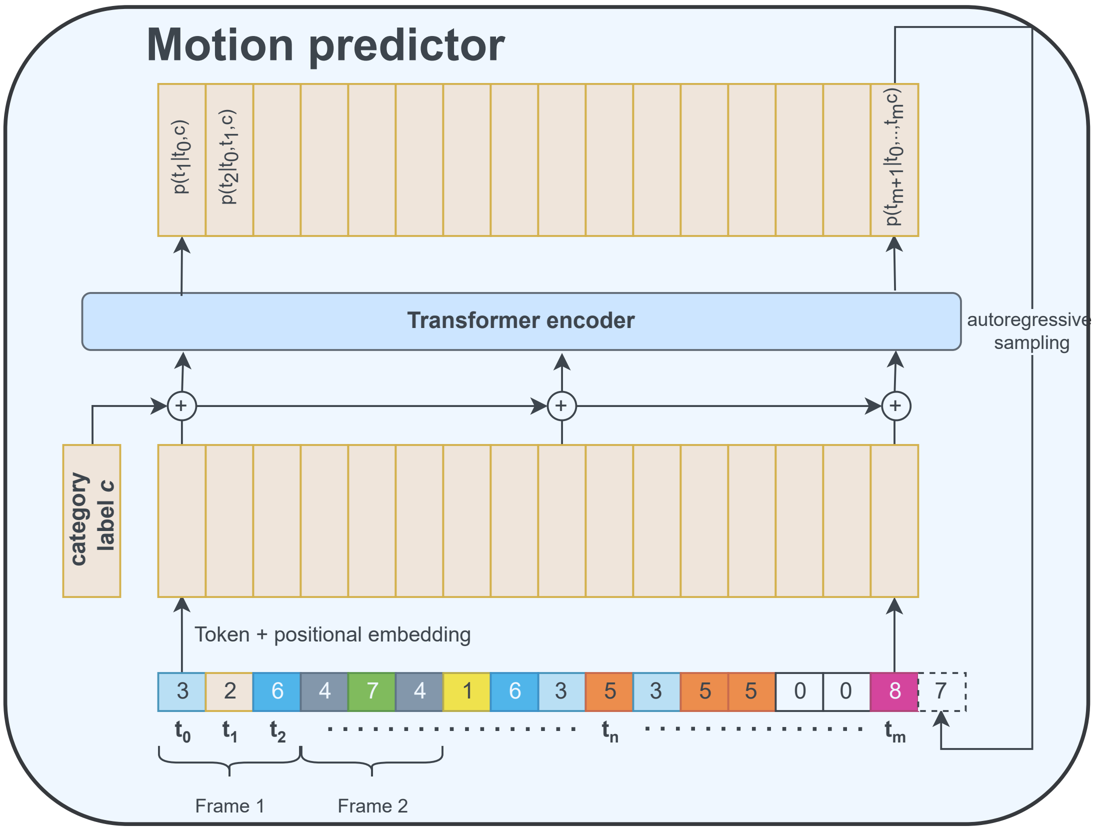


Figure 3.6: **Motion predictor** - Transformer[21] model is used for predicting next frames of the starting motion. The generation operates on latent space obtained from the first stage Section 3.2, in this particular case each pose consists of 3 tokens (i.e 3 cuts).

3.3 Training

In this section, we describe how to train the presented models, in particular, we explain the training procedure, define the used losses, and other training techniques used for both the motion autoencoder and the predictor.

3.3.1 Training motion autoencoder

The goal of the autoencoder is to create a mapping between discrete and continuous representations of a motion. Therefore the training loss consists of two parts as is standard in training variational autoencoders[27]: L_{rec} responsible for reconstruction and L_{lat} for imposing further desired constraints on the latent (discrete in our case) space.

Reconstruction Loss

We experiment with the usage of the following reconstruction losses from [3]:

- L_{6D} is a simple L_2 (mean squared error loss) loss acting directly on the 6D representation of motions. The disadvantage of this is an accumulation of rotation errors across the kinematic chain.
- L_{Joint} - SMPL model can differentiably map between the 6D repr. and the positions of its joints[1]. Therefore L_{Joint} is L_2 loss on 3D positions of pose's joints. Although this does not accumulate rotation errors, it cannot constrain the rotation of bones. Therefore a third loss can be employed.
- L_{Vertex} is L_2 acting directly on the position of vertices of the SMPL mesh. Although it is the most precise and in some sense the true reconstruction loss, we did not use it during most of our experiments due to its high computational costs and additional memory requirements compared to others. SMPL implements model skinning using a weighted sum of joint positions where the weights are based on distances to the joints (at the default pose).[2] Making this loss joint-wise dynamically re-weighted version of the joint loss.

The full reconstruction loss L_{rec} between original motion M and its reconstruction $\widehat{M}$ is thus defined as

$$\begin{aligned}
L_{6D} &= \left\| M - \overline{M} \right\| \\
L_{Joint} &= \| J_M - J_{\overline{M}} \| \\
L_{Vert} &= \| V_M - V_{\overline{M}} \| \\
L_{rec} &= w_{6D} L_{6D} + w_{Joint} L_{Joint} + w_{Vert} L_{vert}
\end{aligned}$$

where w are the weights of individual loss terms. J_M and V_M are joints and mesh vertices of motion M, respectively. Both are represented as points in 3D space, reconstruction losses apart from 6D are thus just distances in Euclidean space. Please note that the norm is applied along the last dimension only, sum over joints/vertices (and mini-batch) is implicitly assumed.

Motion smoothness

We noticed that our trained autoencoders do not perform any smoothing of the training data in the temporal dimension. Although this seems logical and one might expect it, but for the ACTOR[3] model has this property, likely due to it being a fully-continuous model and employing a VAE which must balance reconstruction errors vs. KL-loss. Regardless, it might be a desirable property

[1]We use `pytorch3D` library[37] for math transformations.

[2]SMPL also contains shape parameters which can further modify the mesh but we do not use them.

to have and even better if we can have some control over it. Meaning that in some situations to generate smooth motions even if the training dataset is not smooth might be desirable. For example, this is true in practice with the datasets extracted from videos and UESTC[26] is one of them. On the other hand, we can argue that our model can retain finer details in the dataset, while ACTOR "smooths them away" and does not have the option not to.

To address this, we have a simple proposal in the form of additional reconstruction loss, very similar to *motion smoothing* term in [38]. Informally, we define L_{smooth} as minimizing the difference between the directions of velocity vectors for joints of two adjacent frames. Formally written, for given three adjacent frames with joints $J_{i-1}, J_i, J_{i+1} \in \mathbb{R}^{24\times 3}$, we define the smoothing loss as

$$L_{smooth} = -\frac{J_i - J_{i-1}}{\|J_i - J_{i-1}\|} \cdot \frac{J_{i+1} - J_i}{\|J_{i+1} - J_i\|}$$

Each fraction is an estimation of the direction of movement for each joint (norm is per-joint), minimizing the negative dot-product thus favouring non-changing directions of movement. Note that this loss goes against the reconstruction one, we intentionally want to slightly diverge from the non-smooth ground truth, preferably in a controllable way. Thus finding a balance or trade-off between reconstructing the details and ignoring the noise in the dataset is required.

Latent loss for gumbel-softmax block

Earlier, in Section 3.1, we introduced two possible but fundamentally different latent architectures, the latent loss differs accordingly. First, if gumbel-softmax is employed, the whole model is differentiable and [15] finds the common VAE loss - Kullback–Leibler divergence[39] between the logits and the uniform distribution necessary and sufficient.

In our experiments, we found the model is sensitive to the weight of this loss and does not train at all for too small or too large values. We had some initial success with using $exp(L_{KL})$ loss instead to force its values low but in the end, it is likely more anecdotal, experiments do not suggest one is superior to the other.

$$L_{KL} = D_{KL}(P\|Q)$$
$$L_{lat} = \beta \exp(L_{KL})$$

Latent loss for nearest-neighbour block

If we opt-in for discretization with nearest neighbour, [16] uses two losses: L_{commit} and L_{code}, both defined individually for a given feature sub-vector v and its closest codebook neighbour v_i as follows:

$$L_{code} = \|\texttt{stop\_grad}(v) - \overline{v_i}\|$$
$$L_{commit} = \|v - \texttt{stop\_grad}(\overline{v_i})\|$$

The former is training the chosen code words to be close to the encoder's outputs, thus keeping the magnitude of code words bounded as argued by [16], and is the only loss training the codebook. The latter *commit* loss ensures the outputs tend do commit to some values.

The original paper also suggests simply copying the gradient of $\overline{M_z}$ to M_z since both have the same dimensions, thus solving the issue of the non-differentiable backwards pass, we employ the same technique with success. [40] explores in more detail this training approach and its similarity to K-means clustering.

3.3.2 Training motion predictor

Now, we turn to describe the last piece of the puzzle – how to train the motion predictor. First, we define the training losses and then additional training and decoding techniques we use later in our experiments in the hope to achieve better results.

The training procedure consists of using the trained autoencoder to convert the motion dataset to a token-based one. Tokens are then flattened as shown in Figure 3.6. These, together with the corresponding category labels, constitute the training dataset for the motion predictor.

The model is trained with cross-entropy loss between the ground truth and distributions predicted by the model. As it is not feasible to train the model by repeatedly feeding back its inputs as will be done at generation time, the standard procedure with one-token-ahead predictions is used. In this approach, the model is fed the ground truth for all time steps, this is called teacher forcing.

The tokens are sampled by applying `softmax` [41] function:

$$softmax_i(y) = \frac{\exp(y_i/T)}{\sum_j \exp(y_j/T)}$$

where $y \in R^k$ are outputs from the transformer and T is the sampling temperature, larger values flatten the distribution, we investigate its effect on the quality of synthesized motions later.

Formally, for given tokens $t_0, \dots t_n$ from the training dataset in one-hot representation ($t_i \in \mathbb{R}^k$) and $\widehat{t_i}$ being normalized (softmax) outputs from the transformer, the task is to find θ parameters of the model minimizing

$$L_{ce} = -\sum_{j=0}^{k-1}\sum_{i=0}^{n}(t_i)_j \log p_\theta(t_i = j | t_0, \dots, t_{i-1}, c) = -\sum_{i=0}^{n} t_i \log(\widehat{t_i})$$

where $p_\theta(t_i = j | t_0, \dots, t_{i-1}, c) \in \mathbb{R}^k$ is the probability of sampling j as the i-th token from the model. Due to masking and attention mechanism, it is conditioned on all previous tokens as well as the category label.

Although the papers [14, 15], we base our work on, generating impressive results, training the model in this way carries certain disadvantages in practice. In language-related tasks, a disparity between outputs generated during training and inference has been observed [42], in particular, the trained autoregressive models sometimes do not generate the desired text and instead decay into generating repetitive, constant, or straight-up incoherent output [43]. Many explanations for this have been put forward [43], mainly blaming the model for never seeing its own inputs, entering a form of error feedback loop during inference.

In our experiments, although generally stable, we too have experienced non-ideal output occasionally, and in fact, we conjecture that the main quantitative limit of our model on UESTC is exactly due to this phenomenon (apart from the limitations of the recognizer itself). Some categories were more prone to generating constant motions where the character did not actually perform any motion. We also believe that our model is more susceptible than the image-based ones since one or few incorrectly predicted pixels are not very much noticeable compared to jumpy or physically impossible poses.

We decided to explore some of the recently proposed enhancements - *top-k* and *top-p* (*nucleus*) [43] sampling as well as *parallel scheduled sampling* [42], all are now described in more detail. The first two were also explored by [14].

Parallel scheduled sampling

The goal of this technique [42] is to mitigate the discrepancy between the distribution of inputs fed into the model during training and then during autoregressive inference. This technique trains the model by unrolling the autoregression for k steps, where teacher-forcing is a special case for $k = 0$[3]. In the first step, the full sequence is passed to the transformer, outputting one-token-ahead predictions (the case shown in Figure 3.6). Gumbel-softmax can then be used to sample from these distributions and the predictions are independently replaced with the ground-truth with probability $1 - p$, obtaining a mix of ground-truths (teacher forcing) and model's predictions. This mix is then again passed as new inputs to the transformer. This is repeated for k steps in total and the probability p is increased during training. The gradient is accumulated across all steps, the loss is defined between the final outputs and the ground truth (not the inputs from the last layer). In the beginning, the model trains mostly on ground truths but progressively often receives its own predictions with the intent of eliminating the mentioned discrepancy.

Top-k and Nucleus sampling

After the model is trained, there is still a question about how to precisely generate the output. In theory, the optimal approach is to directly sample from the output distributions for the tokens, on the other hand, in our experience, such approach sometimes leads to sub-optimal results as we have explained above.

One option uses the top-k decoding strategy [44] which in each step only considers k tokens with the highest probabilities and samples just from them (their distribution is re-normalized). This eliminates very rare tokens that might confuse the transformer in the next step. An alternative variant is top-p [45] which is very similar but instead of picking a fixed set of most probable tokens, it chooses a minimal set of tokens which meets a total probability threshold, for example, $p = 0.95$ samples just from the tokens whose total probability is 95%.

Decoding with sliding window

Auto-regressive generation can generate sequences unlimited in length but it cannot be by its nature computed in parallel, therefore our model is not very fast at generation, especially compared to ACTOR [3] which operates on whole sequences at once. This issue is more prominent for higher values of cuts s (number

[3]We use k=1.

of tokens per frame) since they directly impact the length of token sequences representing a particular motion with length k requires $s \cdot k$ tokens. Even if we disregard the generation speed, we are still bounded by quadratic memory $O((s \cdot k)^2)$ requirements due to the attention mechanism. Therefore, in every autoregressive decoding step, we limit the past context to a fixed maximum length – *sliding window* in the same way as our predecessors [14]. We did not explore any more advanced architectures approximating the attention mechanism with linear complexity such as [46, 47], which is one of the valid topics of future work.

There are known implementation tricks [48] to cache the intermediate attention computations during autoregressive generation relying on the fact that in each forward step, we only care about the last token. Although this is a very efficient solution, reducing the decoding complexity from $O(n^3)$ [4] to manageable $O(n^2)$ for n output tokens, but in its current form, it does not work with sliding window approach we decided to use.The problem we face is caching outputs from attention layers also caches the positional embedding encoded in them, therefore one must use absolute positions in the to-be-generated sequence. This rules out the possibility to use trainable embeddings due to their finite amount, chosen prior training. On the other hand, the standard sinusoidal embeddings[21] are in practice not limited by any maximum length and can be used with absolute positions even beyond the size of the window, at least assuming the model learned to generalize. In our experiments, we have observed heavy degradation even when trained with variable-length motions.

Initially, we believed we would be able to solve this with a clever usage of this attention cache and a special positional encoding we devised. We were wrong and our method does not work, but the positional encoding is still sound and we had good results with it, therefore we describe it here briefly.

It amounts to a simple tweak to the sinusoidal embeddings. We use sine waves with periods a fractions of the sliding window and use the absolute positions in the generated sequence. This creates a repeating standing waves-like pattern instead of the "unbound" sinusoidal embeddings. See Figure 3.7 for an illustration of this pattern. The disadvantage is the transformer is truly limited in this sliding window, if given a larger sequence as a whole, the positional embeddings will not be unique, but repeat themselves; but that was exactly the intended point and such scenario does not happen in our use case. It is worth noting we vary the starting phase during training so the network is used for any position. It is also important to have at least one wave with a period twice as long as the sliding window. This is needed to ensure that no matter the starting phase, the order of positional embeddings is unique.

$$PE(i, 2k) = sin(2\pi \frac{2k+1}{T} + \phi_0)$$
$$PE(i, 2k+1) = cos(2\pi \frac{(2k+1)+1}{T} + \phi_0)$$

where $PE(i, j)$ is the $j-$th component of a d-dimensional vector added to $i-$th (absolute) token in the sequence. Note that T is the period set to at least twice

[4]To generate n tokens, i-th step takes $O(i^2)$ due to attention between all token pairs.

the length of the sliding window. The plus one ensures we do not start from 0. The starting phase ϕ_0 can also be varied during training.

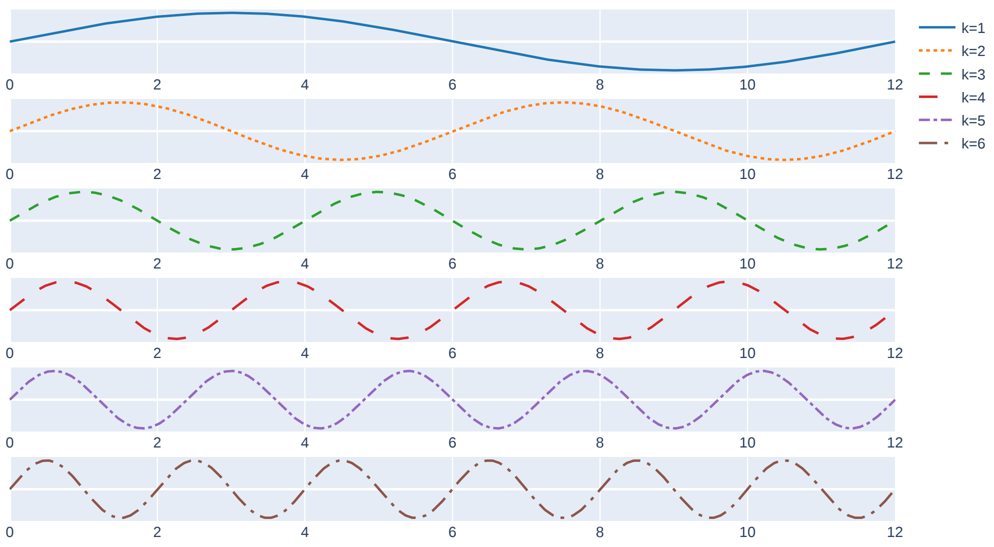


Figure 3.7: **Harmonic waves as positional embedding** - Our proposal is to use even harmonic waves for positional embeddings in the motion predictor and as well as attention-based motion autoencoder. Above we show the first six even harmonic waves with period $T = 12$.

Alternative approaches that would actually allow caching the outputs even in the presence of sliding window, which we also did not explore, have been suggested by [49] after observing the above problem; that is to use relative positional embeddings instead which do not suffer from this issue. Yet another solution is used in Shortformer [50] model whose authors decided to disentangle positional embeddings from the attention outputs and instead add them on the fly to each key and query vectors in the attention mechanism, caching only the pure non-positional outputs.

3.4 Summary

In this chapter, we have described a new model for motion generation leveraging previous work on discrete autoencoders and adapted it for unlabeled motion latent discrete representation. The generation is achieved using transformed-based architecture. We have described multiple variants of the new model as well as training procedures and related techniques for sampling the motions from the trained models.

4. Experiments

In this chapter, we experiment with the models presented earlier, these experiments should provide results and justifications for our ideas. Then we compare the architecture with state of the art models described in Chapter 2

In detail, this chapter is divided into the following sections:

- First, we describe our experimental setup with used datasets and metrics for evaluation.
- Second, we present our results of the autoencoder model and show and discuss through ablation studies the impact of the most important parameters on its performance. After summarizing the best-found autoencoders, we take the models and use them to generate token datasets. We dedicate a few paragraphs to exploring the properties of the generated datasets in hopes of later better explaining the performance and differences between the trained motion predictors.
- Third, we take the autoencoders and train motion predictor models with them. Again, in our ablation studies, we explore the effects of various hyper-parameters, decoding strategies, and other techniques presented earlier, showing their benefits. The section then continues with a comparison to the state of the art results and a discussion on the quality of the generated motions.

4.1 Experimental setup

For the experiments, we have chosen to work work with UESTC dataset. We restrict training of the autoencoder to sequences of 60 frames at maximum, this is partly done due to training time complexity and partly because our main comparison is to the ACTOR model which also uses 60 frames.

Now, we describe the used data for training and evaluation.

UESTC Originally, UESTC is a dataset of videos obtained with multiple cameras in a controlled environment. Multiple subjects were filmed performing various motions grouped into 40 categories, the whole dataset consists of 25600 videos. ACTOR and our model both work with skeleton-based(SMPL) motions, therefore authors of ACTOR[3] used VIBE model[51] to extract the skeletons from raw videos. This makes the motions in the dataset not exactly smooth and slightly "jumpy", we explore this peculiarity further down in Section 4.3.

To make the comparison with ACTOR fair, we use the same train and test splits throughout our experiments with the help of ACTOR's open source codes. The splits are made of disjoint sets of subjects to eliminate any possible leaks of the test data, the splits were made roughly equal in size. During development, we have used a part of the training split as our validation set.

UESTC-ACTOR While inspecting our results on the original UESTC dataset, we have noticed they are much less smooth than that of ACTOR, please refer to Section 3.3 as we discussed the topic there already.

We want to check and ensure that the non-smoothness of our results is not due to some hidden limitation of the chosen architecture, for example, due to discretization, but instead is the consequence of the non-smoothness of the UESTC dataset in the first place.

To confirm this hypothesis, that our model is just not biased towards smoothly generated motions and can equally reconstruct smooth and non-smooth data, we run a few more experiments for which we create a new temporary dataset which we denote A-UESTC [1]. It is just the original UESTC dataset as reconstructed by a trained ACTOR model which smooths it. Albeit, it is organized slightly differently because instead of having roughly 12k training long motions from which we randomly sample 60 frame-long motions during training, it has 65k 60-frame motions because ACTOR is unable to reconstruct such long motions without degradation. The 65k motions were thus sampled ahead of time and reconstructed. The number has been chosen in order for the dataset to be comparable in size with respect to the total number of frames(poses) in both datasets.

4.2 Evaluation criteria

Due to our two-stage approach, we encounter a slight difficulty - training all combinations of autoencoders and then motion predictors is infeasible for us. So we have to resort to first finding good hyper-parameters for the autoencoder and then we only take a couple of well-performing models and turn our focus to the motion predictor.

To evaluate the autoencoder, we mainly rely on L_{Vert} as the reconstruction metric because we will not use it for the majority of our experiments due to its computational costs, opting for a combination of L_{6D} and L_{Joint} instead.

But, we again want to stress that training the autoencoder is just the first stage of our task, we are mainly interested in how the motion predictor performs on generating motions. Therefore to better understand the results of even just autoencoders, we will use the same motion predictor model for all of them, train it on the generated token dataset and evaluate it with metrics soon to be introduced. This way, we obtain basic quantitative metrics which can help find the best parameters for the models or at least report any deficiencies.

Please keep in mind that some architectures - models using nearest neighbours as their latent block in particular - benefit from additional training techniques we described in Chapter 3 and explore later in Section 4.4. The results should therefore be used merely for comparison and to ensure that the predictor's performance is not suspiciously low.

To evaluate the quality of generated motions, we rely on the benchmark used by ACTOR in [3], shared from [5]. At its centre is an extra convolutional neural network - *a recognizer* trained, on the same dataset as the motion predictor, at the action recognition classification task. In other words, the network takes a generated motion and predicts its category. We use the pre-trained model

[1]For the purpose of this work only.

provided by its ACTOR authors so the results are comparable across papers, it also simplifies our implementation.

The predictor's evaluation process consists of taking labels from the test set and generating motions from them with the tested motion predictor, the obtained set of generated motions is evaluated with the following metrics using the trained recognizer.

1. **FID** stands for Fréchet Inception Distance[52], widely used metrics for comparing the quality of generated images. It computes distances between two Gaussian distributions, in particular, the activation values for the generated test sets of the last layer before softmax is applied in the recognizer are used. The lower the distance, the more similar the sets look to the recognizer.

2. **Accuracy** Measures simply the accuracy of the action recognition, we note that the recognizer itself does not achieve 100% accuracy on the test set, [3] report 98.8% on the whole UESTC dataset. We further noticed that the recognizer has trouble with distinguishing between particular categories, more details follow in Section 4.4.1.

3. **Diversity** is defined as the variance between the feature vectors (same ones as for FID) across the whole motion set. Thus higher diversity implies more diverse motions, at least from the point of the recognizer. Because the model is conditioned on different categories, with very distinct motions, diversity will not be close to zero unless the motions are heavily flawed. Importantly, it will also not be zero even if the model generates a single motion per category, therefore the following fourth metric is employed.

4. **Multimodality** is simply diversity computed on a per-category basis and only then averaged over all categories, this alleviates the aforementioned problem.

4.3 Evaluating motion discretization

In Chapter 3, we have proposed a novel motion autoencoder model, described its training, and defined the used losses.

In this section, we present results for the autoencoder model, we first show baseline results for various combinations of autoencoder blocks presented in the previous chapter. Then we focus on various hyper-parameters and small tweaks and observe their impact on the reconstruction.

We chose UESTC as the basis for most of our experiments due to its reasonable size and presence of clear evaluation metrics from[3].

Our interest is of course to find the best model for generating motions, reconstruction is just a sub-task of that. In order to achieve that, we pay special attention later in this section to the learned codebook and distributions of its tokens across the training dataset. We show that even similarly performing autoencoders can result in vastly different token datasets which will have an impact later in the next section.

The results are thoroughly discussed and the final summarization with the best motion models is presented at the end of this section. Those models are then used for training and evaluation of the motion predictors in the next section.

Implementation details and reporting results

For the next autoencoder experiments (apart from Table 4.3), we use training loss weights of $0.5, 1.0, 0.0$ for L_{6D}, L_{joint}, L_{Vert} respectively. We have found these to work quite well and because L_{Vert} is zero, it can be used as a rough metric to assess the overall quality of reconstruction. A codebook of 512 32-dimensional code words is used. The tabulated results are also accompanied by reconstructed motions.

Unless explicitly stated, the presented results are measured on UESTC test set. Do note that the shown 95% confidence intervals for the additional metrics based on training extra motion predictors are calculated by generating 10 sets of motions which are then evaluated. The predictor is trained only once with the first-run autoencoder and evaluated ten times using a single recognizer.[2]

It is not feasible for us to show all hyper-parameters used for training in each table, kindly refer to the source code's `README.md` for instructions on how to obtain the precise experiment configurations used for all the results we present.

In reporting all reconstruction losses we omit 10^{-5} term, for example, the reported loss of 12.5 is in fact .000125. For addressing the variants of the models in more manageable and terse way, we use the following shortcuts: gumbel-softmax(`gs`), nearest-neighbour(`nn`), attention-based(`attn`), convolutional-based (`conv`).

[2]Due to time constraints we did not choose the best autoencoder, although from experiments we did manage to run for some models and the low deviations in the autoencoder metrics, we conclude this is would not be highly impactful.

Auto	Latent	Cuts	$L_{6D} \downarrow$ 10^{-5}	$L_{Joint} \downarrow$ 10^{-5}	$L_{Vert} \downarrow$ 10^{-5}	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$
attn	gs	4	44.79	18.05	23.62	$\mathbf{88.58^{\pm 0.38}}$	$12.1^{\pm 0.26}$	$\mathbf{16.23^{\pm 0.06}}$
attn	nn	4	51.26	17.34	22.67	$85.93^{\pm 0.48}$	$15.86^{\pm 0.25}$	$16.81^{\pm 0.21}$
attn	gs	8	29.91	9.63	12.68	$85.89^{\pm 0.38}$	$\mathbf{11.04^{\pm 0.48}}$	$16.91^{\pm 0.12}$
attn	nn	8	**25.59**	**6.42**	**8.57**	$83.34^{\pm 0.37}$	$16.6^{\pm 0.42}$	$17.17^{\pm 0.12}$
conv	gs	4	57.40	24.99	33.21	$87.41^{\pm 0.34}$	$14.39^{\pm 0.32}$	$\mathbf{16.11^{\pm 0.14}}$
conv	nn	4	55.78	23.38	30.57	$\mathbf{87.47^{\pm 0.45}}$	$\mathbf{12.46^{\pm 0.53}}$	$16.46^{\pm 0.15}$
conv	gs	8	29.27	11.53	15.33	$85.27^{\pm 0.44}$	$14.25^{\pm 0.70}$	$16.85^{\pm 0.13}$
conv	nn	8	**22.10**	**6.62**	**8.71**	$84.23^{\pm 0.42}$	$14.64^{\pm 0.29}$	$16.96^{\pm 0.16}$

Table 4.1: **Comparison of autoencoder blocks** - Performance of autoencoder models trained and evaluated on UESTC dataset, with trained baseline motion predictor. We compare attention-based (`attn`) to convolutional-based (`conv`) encoder, decoder as well as possible latent variants - gumbel-softmax(`gs`) and nearest neigbhours(`nn`).

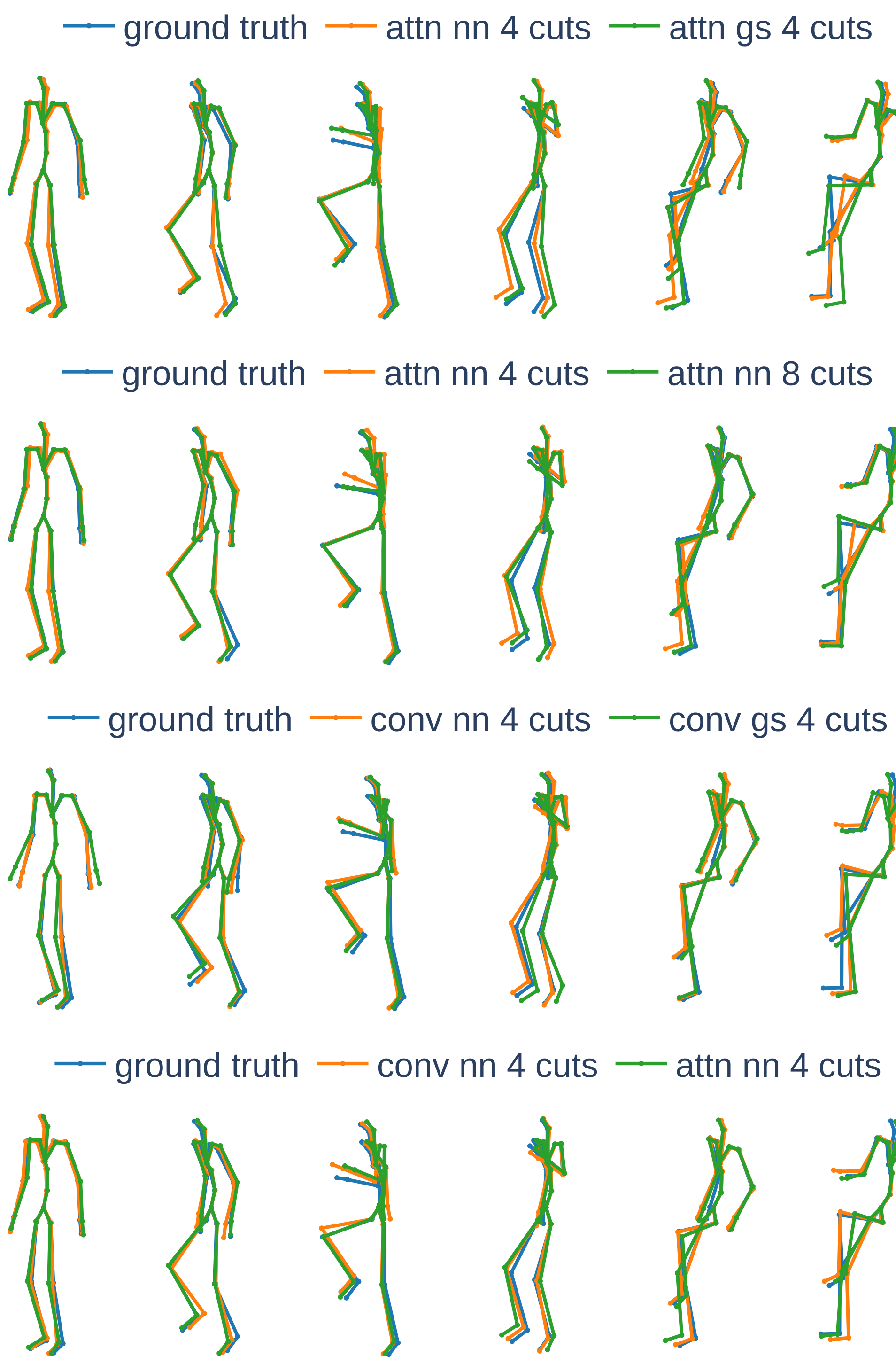


Figure 4.1: **Reconstruction Quality of Motion Autoencoders** - A running motion as reconstructed by autoencoders shown in Table 4.1. The stick figures represent the kinematic tree with joints and bones.

Ablating architecture design choices

Table 4.1 presents the baseline results for discussed variants of the autoencoder. We can clearly see overall quantitative improvement in the reconstruction for 8 cuts rather than 4, we investigate that more closely in a moment. It appears that attention-based models achieve better reconstruction results across the board. Interestingly, the nearest neighbour latent block appears to outperform gumbel-softmax noticeably for 8 cuts but only slightly in case of 4, this might suggest that it does not scale very well in case of higher-compression values (i.e. fewer cuts).

In Figure 4.1 we show comparisons between the various models. Because all models generally do a very good job of reconstruction, the differences in side-by-side plots would not be noticeable, therefore we have grouped some motions together and used stick figures with joints instead of full meshes to see the differences more clearly.

In Figure 4.3 there are more reconstructions across different categories.

Yet more interesting are the numbers from trained and evaluated motion predictors because it appears the performance of the motion predictor clearly favours gumbel-softmax(`gs`) for attention-based models, on the other hand, `conv` models are inconclusive at best.

This might be partly caused by two factors: (1) the decoding strategy we used in this particular evaluation - we auto-regressively generate the motions using a simple softmax as we described in Chapter 3. It turns out later that the nearest-neighbour approach benefits further from the use of extra decoding strategies. This phenomenon is explored in the next section focused on motion prediction. (2) the different distributions of tokens in codebooks which we noticed and therefore we dedicate a later part of this section to get a better look at what exactly is different and how can the results be better understood.

Ablating cuts

Following on the cuts observation made earlier, more extensive results are in presented in Table 4.2. Here we can confirm that reconstruction benefits from higher number of cuts because the compression factor is lower, we again see `attn` and `nn` outperforming `conv` and `gs`. But, there is a limit to its usefulness for motion prediction. More cuts lead to longer sequences and higher computational requirements, the results also suggest this does not always translate into better generating quantitative results.

[16] reported compression ratios of up to 42 for images. Repeating the calculation for our case is slightly more ambiguous, we think so because of the uncertainty of how large the input really is in practice. Sure, we use $f \times 24 \times 6 \times 32$bits to represent the motion but surely 32bit floating-point representation is rather excessive, also each joint can be unambiguously specified by only 3 angles. If we for a moment assume 1 deg precision is sufficient, taking up 9bits, similarly the most common codebook of 512 vectors can be indexed also by 9 bits. For 4 cuts, this leads to a reduction of $\frac{24 \times 3 \times 9}{4 \times 9} = 18$ in bits needed to represent the motion by our conservative estimates. Or 9 if we use 8 cuts. In both cases, we deem the reconstruction more than acceptable for our purposes.

Auto	Latent	Cuts	$L_{6D} \downarrow$ 10^{-5}	$L_{Joint} \downarrow$ 10^{-5}	$L_{Vert} \downarrow$ 10^{-5}	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$
attn	gs	2	86.25	45.27	59.47	$\mathbf{88.70^{\pm 0.32}}$	$14.26^{\pm 0.31}$	$\mathbf{16.02^{\pm 0.14}}$
attn	gs	4	44.79	18.05	23.62	$88.58^{\pm 0.38}$	$12.10^{\pm 0.26}$	$16.23^{\pm 0.06}$
attn	gs	6	35.44	13.08	16.70	$86.08^{\pm 0.24}$	$12.68^{\pm 0.67}$	$16.65^{\pm 0.14}$
attn	gs	8	29.91	9.63	12.68	$85.89^{\pm 0.38}$	$\mathbf{11.04^{\pm 0.48}}$	$16.91^{\pm 0.12}$
attn	gs	10	**20.56**	**6.45**	**8.51**	$85.88^{\pm 0.29}$	$12.22^{\pm 0.35}$	$16.67^{\pm 0.11}$
attn	nn	2	59.22	25.22	32.70	$\mathbf{89.05^{\pm 0.43}}$	$\mathbf{11.61^{\pm 0.34}}$	$\mathbf{16.25^{\pm 0.14}}$
attn	nn	4	51.26	17.34	22.67	$85.93^{\pm 0.48}$	$15.86^{\pm 0.25}$	$16.81^{\pm 0.21}$
attn	nn	6	37.11	10.35	13.88	$84.58^{\pm 0.43}$	$15.14^{\pm 0.66}$	$17.1^{\pm 0.21}$
attn	nn	8	25.59	6.42	8.57	$83.34^{\pm 0.37}$	$16.60^{\pm 0.42}$	$17.17^{\pm 0.12}$
attn	nn	10	20.89	5.01	6.53	$82.71^{\pm 0.41}$	$16.61^{\pm 0.81}$	$17.42^{\pm 0.14}$
conv	gs	2	109.58	59.17	77.65	$86.97^{\pm 0.33}$	$21.12^{\pm 0.67}$	$\mathbf{15.69^{\pm 0.11}}$
conv	gs	4	57.40	24.99	33.21	$\mathbf{87.41^{\pm 0.34}}$	$14.39^{\pm 0.32}$	$16.11^{\pm 0.14}$
conv	gs	6	34.83	13.65	18.20	$86.01^{\pm 0.42}$	$14.31^{\pm 0.40}$	$16.56^{\pm 0.12}$
conv	gs	8	29.27	11.53	15.33	$85.27^{\pm 0.44}$	$\mathbf{14.25^{\pm 0.70}}$	$16.85^{\pm 0.13}$
conv	gs	10	**22.96**	**8.47**	**11.14**	$82.92^{\pm 0.40}$	$17.34^{\pm 0.46}$	$17.01^{\pm 0.12}$
conv	nn	2	128.42	73.77	94.21	$86.55^{\pm 0.19}$	$17.08^{\pm 0.50}$	$\mathbf{16.22^{\pm 0.11}}$
conv	nn	4	55.78	23.38	30.57	$\mathbf{87.47^{\pm 0.45}}$	$\mathbf{12.46^{\pm 0.53}}$	$16.46^{\pm 0.15}$
conv	nn	6	30.94	9.76	12.86	$85.43^{\pm 0.41}$	$13.93^{\pm 0.35}$	$16.89^{\pm 0.13}$
conv	nn	8	**22.10**	6.62	8.71	$84.23^{\pm 0.42}$	$14.64^{\pm 0.29}$	$16.96^{\pm 0.16}$
conv	nn	10	22.37	**6.60**	**8.70**	$84.45^{\pm 0.33}$	$15.28^{\pm 0.39}$	$17.01^{\pm 0.15}$

Table 4.2: **Impact of latent code composition for Autoencoder performance -** Convolutional autoencoder and basic motion predictor trained with various number of cuts in the latent codebook. Notice the falling performance of motion prediction using large-cut token representation.

If we did stick to the original representation $f \times 24 \times 6 \times 32$bits, the compression for 4 and 8 cuts is 128 and 64.

Reconstruction losses

In Table 4.3 we consider weighting of different reconstruction losses and its impact. Naturally, the larger the weight is, the smaller the corresponding loss. As we have noted earlier, 6D loss accumulates errors across the kinematic chain which can be seen through higher L_{Vert} losses.

We must be careful with these results now because we no longer can use loss L_{Vert} as a metric as it is used during training as a loss itself. Summing up or averaging these losses, even weighted (i.e. reporting the training loss) is also not very meaningful due to the different spaces in which they operate.

Pure vertex loss stands out in the table (as was mentioned during its description) because we use 6D representation involving rotation matrices, the loss leaves each matrix unconstrained as long as the mesh is valid, at least in theory. Using only joint loss is not sufficient [3] as it does not constrain rotation of bones along their axes, therefore we combine it with 6D loss.

w_{6D}	w_{Joint}	w_{Vert}	w_{smooth}	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	L_{Vert}
1.0	0.0	0.0	0.0	45.95	57.52	71.60
1.0	0.0	1.0	0.0	58.61	33.08	36.54
0.1	1.0	0.0	0.0	84.51	22.40	29.95
0.2	1.0	0.0	0.0	75.79	24.17	32.10
0.5	1.0	0.0	0.0	52.83	21.30	28.05
1.0	1.0	0.0	0.0	**41.76**	20.06	25.81
0.0	1.0	0.0	0.0	526054.43	19.04	358.85
0.1	1.0	1.0	0.0	80.49	**17.58**	**20.95**
0.0	0.0	1.0	0.0	691476.00	24.52	25.53
0.5	1.0	0.0	10^{-5}	46.84	18.37	23.83

Table 4.3: **Impact of weighted losses on motion reconstruction** - Convolutional autoencoders with 4 cuts and nearest neighbour trained with different weights of reconstruction losses.

About the smoothing loss, only briefly mentioned here, we can see that adding it does not degrade any metrics, it even slightly improves them.

In the end, we found that ratios of 1 : 10 to 1 : 1 for $L_{6D} : L_{Joint}$ all work well for reconstruction, 1 : 1 has the lowest 6D loss which is desirable to get the rotations along the joint correct.

These experiments also confirm that our combination of L_{6D} and L_{Joint} is comparable to using L_{Vert} without its computational drawbacks.

The reconstruction of a single motion by differently-weighted models can be seen in Figure 4.2.

Training stability and latent losses

Training with the nearest neighbour proved quite stable for a wide set of weight combinations as can be seen in Table 4.5, also keeping commit loss lower appears to achieve better results. On the other hand, we have noticed that training using gumbel-softmax is unstable for larger values KL-loss for which it tends to get stuck at the start. Either the encoder degenerates to a point of predicting a single code word or the distribution stays random but in neither case, the network learns. From our investigation we believe the vanishing gradient might be caused or being stuck in some local optima - see Table 4.4. Varying the training rate did not always help either.

Overall the model appears to be stable with respect to the batch size and learning rate. We use Adam optimizer, batch size of 64 and learning rate of 2.10^{-4}, annealed down over the training period.

Codebook parameters

Table 4.6 and Table 4.7 address the effect of different codebooks. As we can see, reconstruction always benefits from larger codebooks, the second table also shows that allocating parameters to a higher number of smaller code words is more beneficial than having fewer but wider code words. One can also observe

w_{KL}	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
10^{-1}	2213.27	1643.45	2413.11
10^{-2}	2212.40	1642.32	2412.00
10^{-3}	194.95	112.83	152.47
10^{-4}	103.47	53.32	71.31
10^{-5}	66.30	29.73	39.36
10^{-6}	**56.24**	**24.50**	32.45
10^{-7}	62.18	36.00	**27.40**
10^{-8}	64.89	38.32	29.30

(a) Linear KL

w_{KL}	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
10^{-1}	2212.68	1642.78	2412.64
10^{-2}	2211.88	1642.21	2411.86
10^{-3}	134.70	72.41	96.65
10^{-4}	**56.15**	**24.95**	**32.96**
10^{-5}	58.42	25.60	33.83
10^{-6}	61.78	27.16	35.90
10^{-7}	62.18	27.81	36.63
10^{-8}	70.15	33.25	43.81

(b) Exponential KL

Table 4.4: **Different weights for KL latent loss** - The impact of KL and $\exp(KL)$ losses on training and performance of a convolutional autoencoder. As one can see, too-high values lead to very sub-optimal results, very small weights are only slightly worse.

w_{commit}	w_{book}	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
0.1	1.0	**48.53**	18.47	24.56
0.2	1.0	50.44	**18.18**	**24.18**
0.5	1.0	59.85	24.18	31.82
1.0	1.0	65.95	29.17	37.74
2.0	1.0	70.88	31.08	39.92
2.0	2.0	53.23	20.19	26.37

Table 4.5: **Different weights for nearest neighbour latent loss** - The impact of L_{commit} and L_{book} losses on training and performance of a convolutional autoencoder.

much better improvement with nearest neighbour latent block compared to use using gumbel-softmax, this is partly explained by codebook utilization of the respective models.

Motion Smoothness

In Table 4.8, we show the impact of our smoothing solution on the reconstruction quality of the trained autoencoders. One can observe that smoothing does not appear to harm the reconstruction at all and the upside should be on-demand smooth motions. Please refer to our supplemental videos to see the impact on reconstruction and later the subsequent generation, here we at least provide numerical justification in Figure 4.4.

In it, we can see the smoothness of UESTC dataset itself and how models can improve this metric. An important observation is that our attention-based model can very precisely model the true dataset if required while ACTOR is very far from it. We believe our approach is thus superior in this regard as it signals a higher capacity of our network for details. Moreover, we can tweak this loss as required to find a balance between the reconstruction of details and smoothness of

(a) Gumbel-softmax latent

Size	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
64	73.67	34.99	45.93
128	67.95	31.15	41.12
256	60.32	26.22	34.61
512	53.82	23.37	30.95
1024	53.19	22.70	30.23
2048	48.91	**20.31**	**27.02**
4096	**49.71**	20.85	27.84
8192	51.79	21.62	28.77

(b) Nearest neighbour latent

Size	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
64	95.63	48.15	61.97
128	71.22	33.11	43.13
256	55.04	22.65	30.01
512	48.87	18.90	24.93
1024	53.52	21.18	27.16
2048	47.79	17.66	22.99
4096	47.36	18.83	24.31
8192	**44.93**	**16.84**	**22.09**

Table 4.6: **Impact of codebook size on reconstruction** - Trained a convolutional autoencoder with fixed-width 32-dim codebook with varying number of code vectors.

Latent	*Size*	*Dim*	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$
gs	512	128	57.60	24.52	32.59	$87.98^{\pm 0.33}$	$14.33^{\pm 0.39}$	$16.14^{\pm 0.13}$
gs	1024	64	49.20	20.78	27.64	$88.18^{\pm 0.4}$	$12.34^{\pm 0.49}$	$15.98^{\pm 0.11}$
gs	2048	32	50.55	21.25	28.44	$87.5^{\pm 0.34}$	$11.82^{\pm 0.50}$	$16.00^{\pm 0.13}$
gs	4096	16	46.43	**19.57**	**26.03**	$\mathbf{88.86^{\pm 0.29}}$	$\mathbf{10.97^{\pm 0.39}}$	$\mathbf{15.78^{\pm 0.20}}$
nn	512	128	66.87	30.61	39.81	$85.77^{\pm 0.42}$	$15.84^{\pm 0.51}$	$16.64^{\pm 0.17}$
nn	1024	64	54.43	22.19	28.84	$86.35^{\pm 0.47}$	$\mathbf{12.52^{\pm 0.55}}$	$16.71^{\pm 0.21}$
nn	2048	32	47.04	18.12	23.12	$\mathbf{86.98^{\pm 0.25}}$	$12.87^{\pm 0.48}$	$\mathbf{16.43^{\pm 0.15}}$
nn	4096	16	**32.45**	**11.07**	**14.36**	$85.48^{\pm 0.37}$	$14.03^{\pm 0.60}$	$16.76^{\pm 0.12}$

Table 4.7: **Reconstruction quality difference between tall and wide codebooks** - Convolutional autoencoder with 4 cuts has been trained with a codebook containing constant number of total parameters, either allocated to many small code words or few large ones.

reconstructed (later generated) motions for any future in-the-wild datasets such as UESTC.

Variable-length training

Since our motion predictor is autoregressive, we are interested in decoding token sequences which might vary in length. For convolutional autoencoder, the length should not be very relevant past the certain threshold given by the depth of the network and sizes of individual kernels.

In contrast, an attention-based autoencoder might be sensitive to the length of encoded/decoded motion. Therefore, we have performed several experiments to investigate:

- Trained on fixed 60-frame long motions and decoding longer, shorter sequences.
- Trained on variable-length motions and decoding longer, shorter.

w_{Smooth}	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$
0	57.60	24.52	32.59	$\mathbf{87.98}^{\pm\mathbf{0.33}}$	$14.33^{\pm 0.39}$	$\mathbf{16.14}^{\pm\mathbf{0.13}}$
5.10^{-4}	62.59	27.29	35.58	$85.64^{\pm 0.71}$	$15.85^{\pm 0.81}$	$16.59^{\pm 0.12}$
2.10^{-4}	53.03	21.76	28.43	$85.96^{\pm 0.49}$	$13.97^{\pm 0.62}$	$16.44^{\pm 0.14}$
1.10^{-4}	48.36	19.02	25.18	$86.13^{\pm 0.31}$	$14.01^{\pm 0.50}$	$16.61^{\pm 0.10}$
5.10^{-5}	57.91	25.14	32.81	$87.57^{\pm 0.33}$	$\mathbf{12.69}^{\pm\mathbf{0.41}}$	$16.38^{\pm 0.17}$
2.10^{-5}	52.68	20.78	27.60	$86.62^{\pm 0.24}$	$13.73^{\pm 0.41}$	$16.56^{\pm 0.16}$
1.10^{-5}	46.84	18.37	23.83	$86.14^{\pm 0.38}$	$13.27^{\pm 0.53}$	$16.56^{\pm 0.16}$
5.10^{-6}	47.33	**17.90**	**23.46**	$86.48^{\pm 0.35}$	$13.70^{\pm 0.28}$	$16.50^{\pm 0.20}$
2.10^{-6}	**47.29**	18.34	24.12	$86.12^{\pm 0.33}$	$13.59^{\pm 0.44}$	$16.58^{\pm 0.2}$

Table 4.8: **Effect of smoothing loss on reconstruction** - Convolutional autoencoder with 4 cuts trained with differently weighted smooth loss.

Embedding	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$
trainable	55.93	22.64	28.81	$86.61^{\pm 0.39}$	$13.08^{\pm 0.44}$	$16.67^{\pm 0.11}$
sincos	51.26	17.34	22.67	$85.33^{\pm 0.59}$	$16.34^{\pm 0.57}$	$16.79^{\pm 0.09}$
harmonics	41.29	13.84	18.08	$\mathbf{87.56}^{\pm\mathbf{0.53}}$	$\mathbf{12.19}^{\pm\mathbf{0.37}}$	$\mathbf{16.55}^{\pm\mathbf{0.19}}$
+rngphase	**38.50**	**12.30**	**16.15**	$86.34^{\pm 0.39}$	$14.11^{\pm 0.49}$	$16.63^{\pm 0.23}$

Table 4.9: **Effect of various positional embeddings on reconstruction** - Attention autoencoder with 4 cuts has been trained with various embeddings presented in Chapter 3, `rngphase` denotes our embedding of harmonic waves with randomly sampled starting phase when training.

In both cases, we use `sincos` positional embedding. The results can be found in Figure 4.5, as we can see, there is a significant deterioration in motion reconstruction on motions of unseen lengths, even if with the expanded training.

Additional parameters for attention-based autoencoders

If we decide to use the attention-based autoencoder, we should at least briefly consider which positional embedding is appropriate.

We have run a few simple experiments with results tabulated in Table 4.9, the main take-away from this is `harmonics` is just as good as the traditional embeddings using sin and cos. If we vary the starting phase ϕ_0, the reconstruction results improve slightly.

Another issue, we touched upon in Chapter 3 is masking, we run the possible combinations and as we can see in Table 4.10, it has little impact on the performance, although we see subjectively more choppy, non-smooth reconstructed motions.

Encoder	Decoder	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow$
Future Masked							
Yes	Yes	53.27	19.79	25.69	$86.51^{\pm 0.40}$	$14.86^{\pm 0.62}$	$16.67^{\pm 0.12}$
No	Yes	54.92	20.84	26.69	$\mathbf{87.15^{\pm 0.47}}$	$\mathbf{13.50^{\pm 0.48}}$	$\mathbf{16.60^{\pm 0.22}}$
Yes	No	55.45	21.62	27.45	$86.32^{\pm 0.41}$	$13.65^{\pm 0.60}$	$16.67^{\pm 0.13}$
No	No	**51.26**	**17.34**	**22.67**	$85.93^{\pm 0.48}$	$15.86^{\pm 0.25}$	$16.81^{\pm 0.21}$

Table 4.10: **Attention mask effect on reconstruction** - Attention-based autoencoder with 4 cuts and nearest neighbour latent block has been trained with attending to future tokens in the attention layers either forbidden(`Yes`) or allowed(`No`).

Encoder	Decoder	Cuts	25%	50%	75%	100%
attn	gs	4	50	146	287	500
attn	nn	4	32	94	210	491
attn	gs	8	65	182	331	512
attn	nn	8	40	103	209	480
conv	gs	4	61	157	285	512
conv	nn	4	36	106	219	510
conv	gs	8	67	170	302	512
conv	nn	8	38	101	206	498

Table 4.11: **Codebook usage of the trained models** - The table depicts how many code vectors are needed to explain 25, 50, 75, 100% of the test set. E.g. for attention-based, gumbel-softmax, 4-cut model, 50 tokens out of 512 accounts for 25% of all tokens of discretized motions in the test set.

(a) Gumbel-softmax latent

Size	25%	50%	75%	100%
64	9	22	39	64
128	18	44	75	128
256	32	84	149	256
512	73	172	301	512
1024	118	316	575	1024
2048	255	650	1172	2048
4096	452	1240	2390	4096
8192	1009	2602	4690	8192

(b) Nearest neighbour latent

Size	25%	50%	75%	100%
64	6	16	32	64
128	13	33	62	127
256	18	52	109	253
512	35	105	216	510
1024	53	179	403	1023
2048	75	275	636	1844
4096	68	243	617	1906
8192	91	329	819	2380

Table 4.12: **Codebook usage of the trained models with varying book size** - Trained models from Table 4.6 with their codebook utilization.

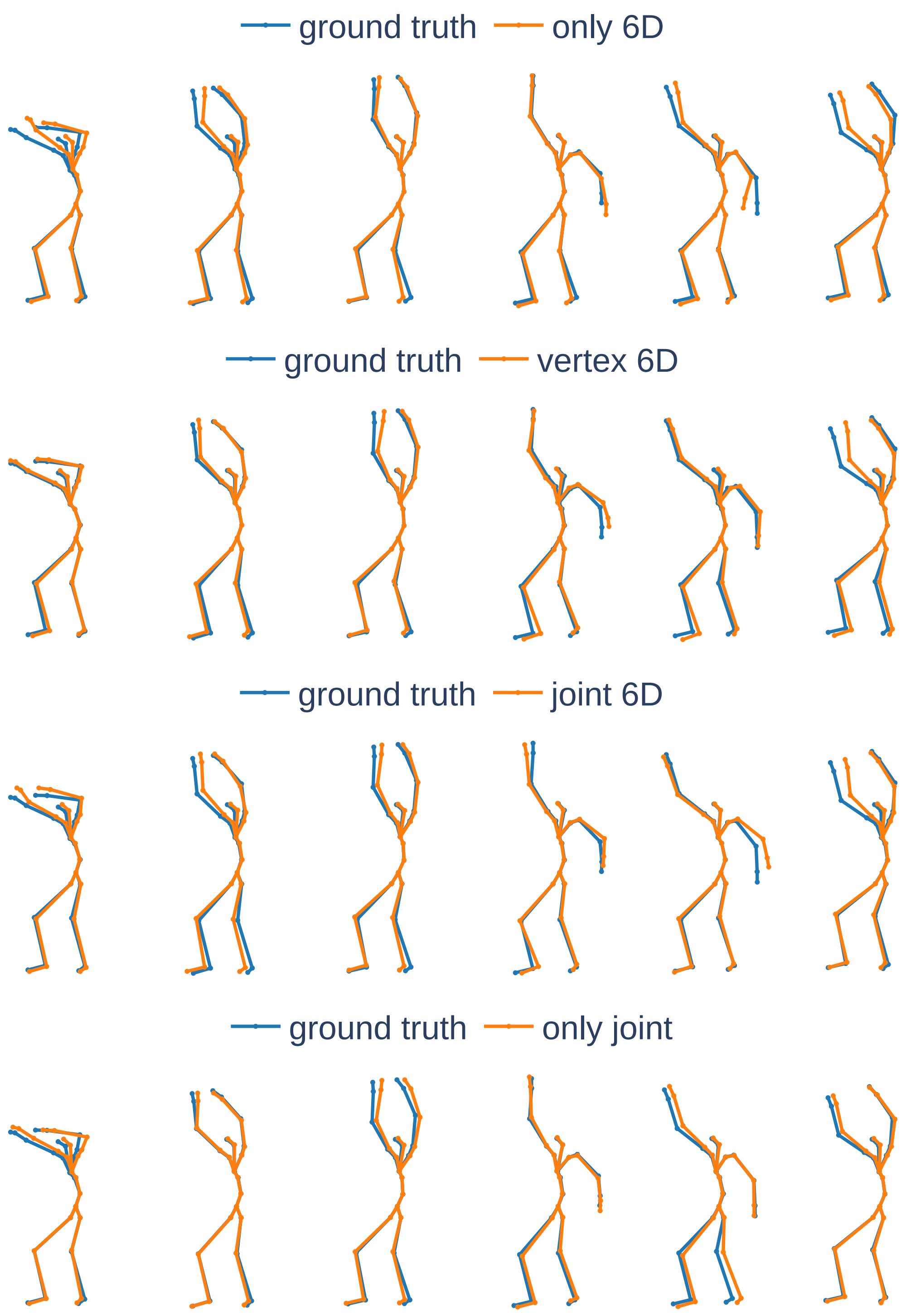


Figure 4.2: **Motion reconstruction as affected by weighted reconstruction losses** - Plots of every 5-th pose from the first 30 reconstructed frames by autoencoder with training losses (Table 4.3, top-to-bottom): 6D only , 6D + vertex, 6D + joint, 6D + joint + smooth. *Overhead stretch* action is depicted.

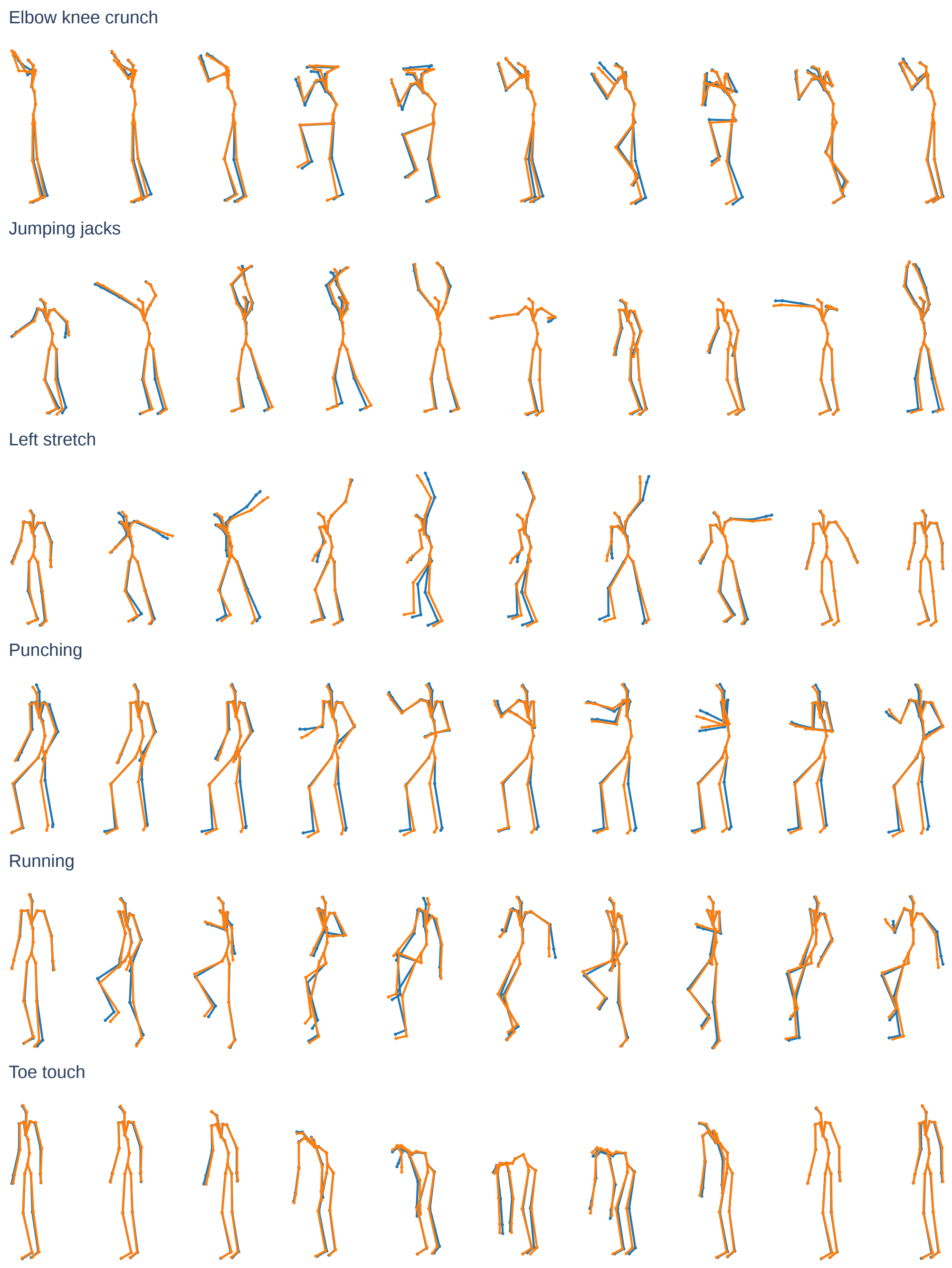


Figure 4.3: **Motion reconstructions of different categories** - Reconstruction from trained convolutional, nearest-neighbour model with 4 cuts.

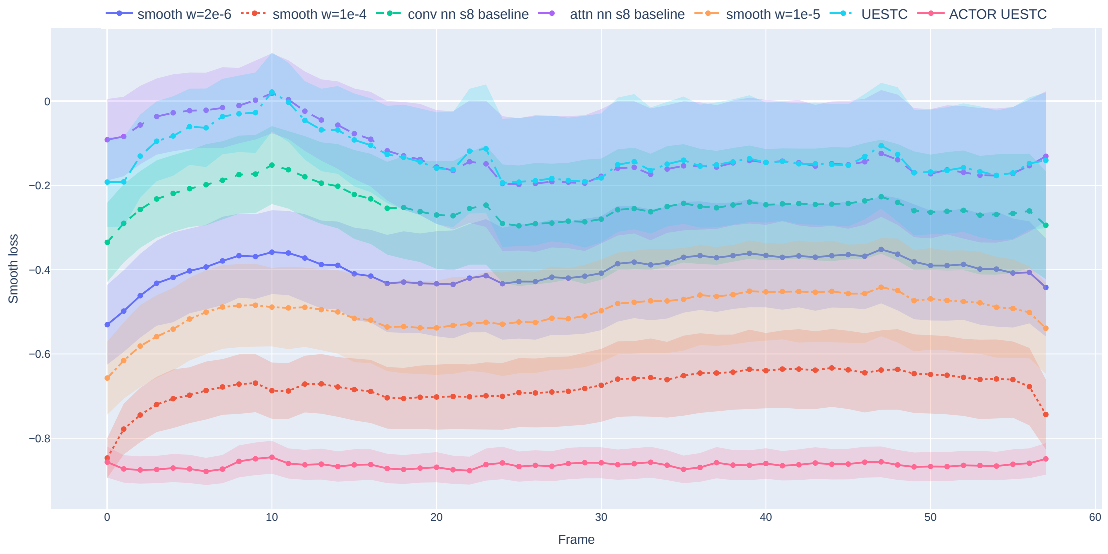


Figure 4.4: **Impact of smoothness loss** - Per-frame smoothness loss across test set for trained models and UESTC ground truth. ACTOR UESTC is A-UESTC(Section 4.1) dataset - i.e. UESTC reconstructed by the ACTOR model. Lower loss means smaller velocity changes between motions and thus smoother motions.

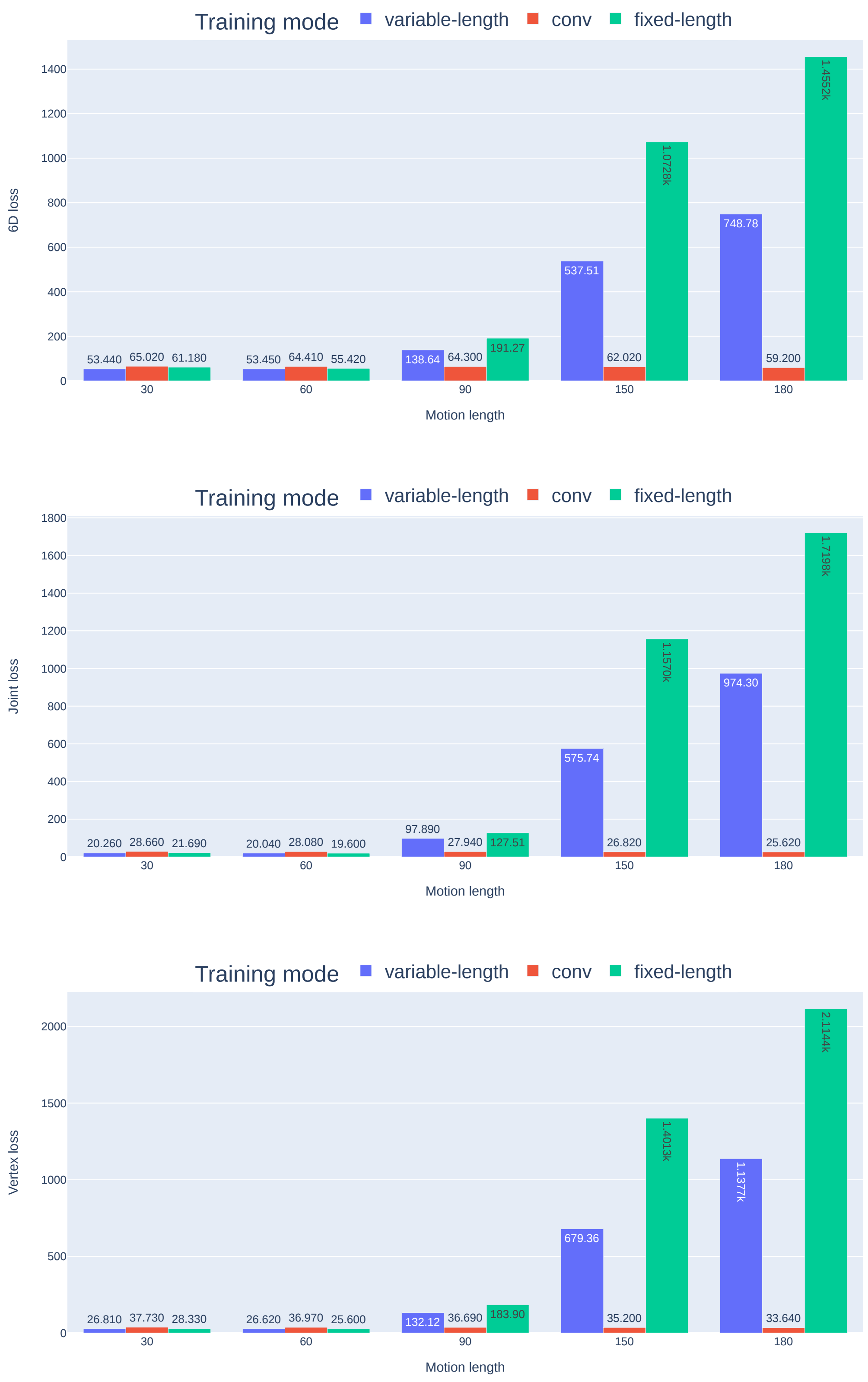


Figure 4.5: **Reconstructing variable-length motions** - Two `attn nn` models trained on UESTC dataset, one on fixed 60-frame motions, the other on 1-frame to 6-frame motions. As a baseline serves a convolutional autoencoder, also based on nearest-neighbour latent block.

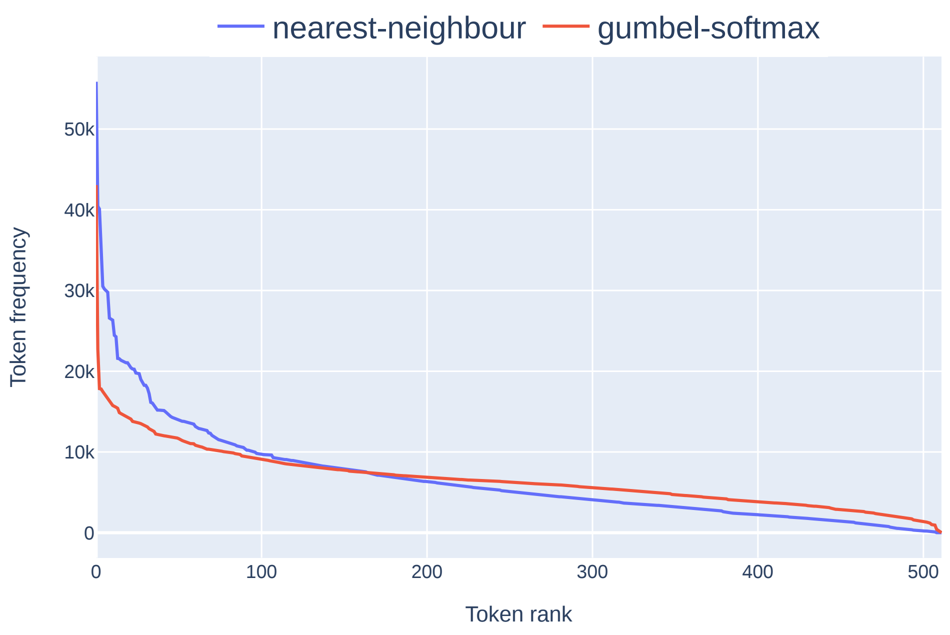
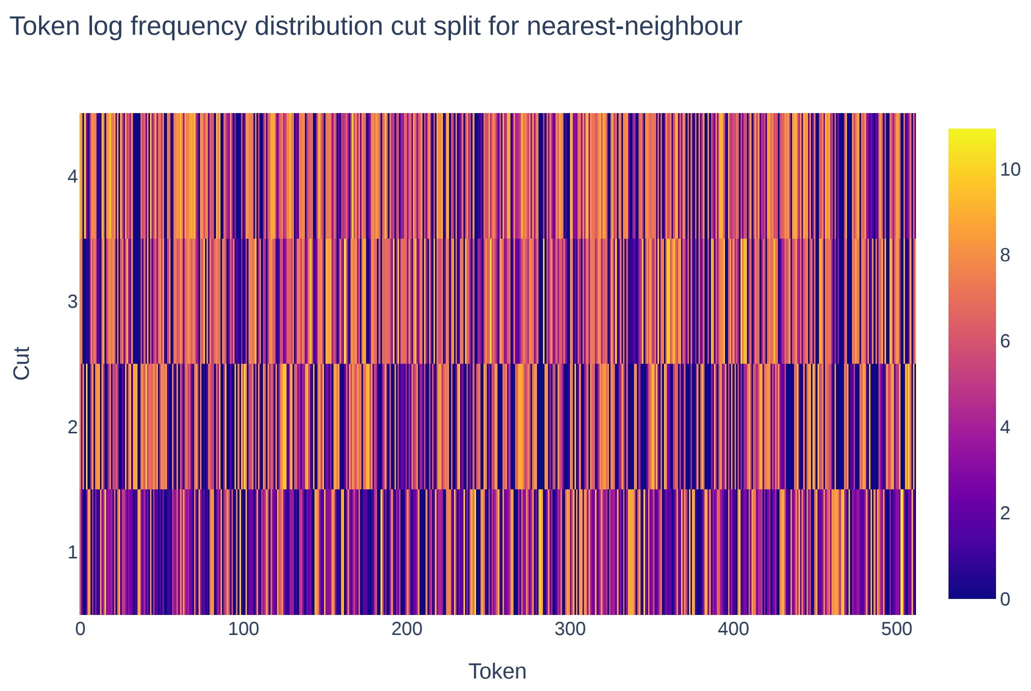


Figure 4.6: **Discrete token distribution** - Effect of latent blocks on token distribution in a convolutional models with 4 splits, sorted from most frequent tokens with.



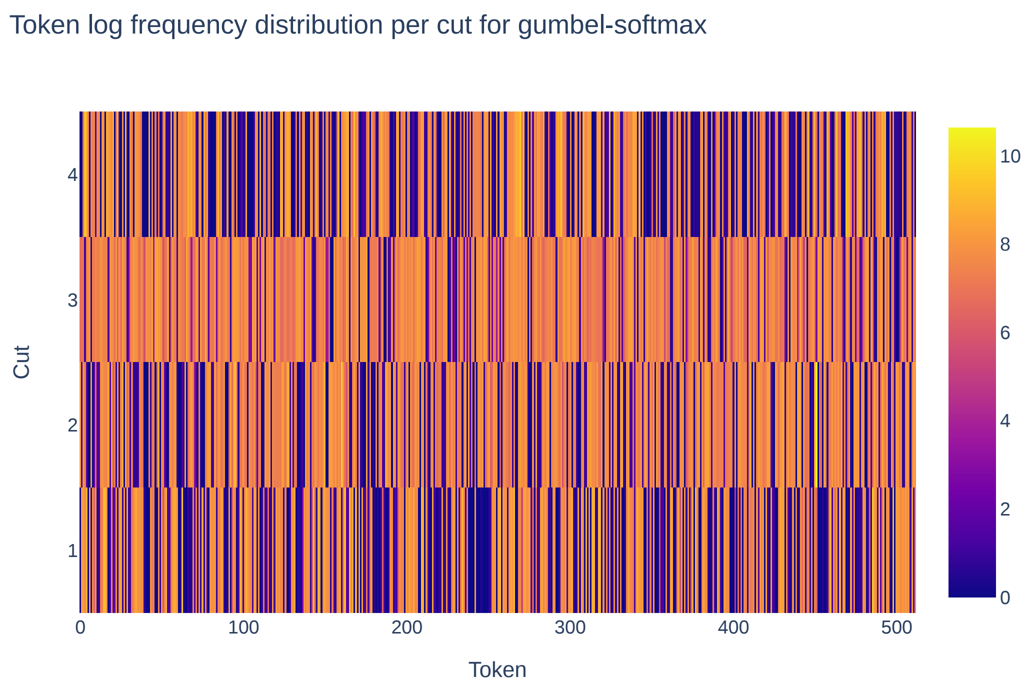


Figure 4.7: **Codebook token per-cut distribution** - Distribution of tokens per cut for gumbel-softmax and nearest-neighbour convolutional autoencoder.

Token distributions

Now, we focus on the distribution of tokens generated by various autoencoder models, it is important to keep in mind that each trained autoencoder is creating a new dataset for the motion predictor to learn from. Throughout our experimentation, we have found that the fundamental differences between gumbel-softmax and nearest-neighbour, that we have laid out in Chapter 3, also apply in practice and there are clear differences between the datasets inferred by those latent blocks.

We can start by looking at the usage of the codebook of the trained models, we have tabulated it in Table 4.11. Since the codebook size is 512 code words, we can observe that not all of them are used, although only a few are left unused for reconstructing the test set.

This is more prominent with larger codebook sizes as shown in Table 4.12. We can see that likely thanks to the sampling-oriented way of gumbel-softmax it nearly always utilized the whole codebook, on the other hand, we were unable to force higher utilization of networks using nearest-neighbour. Although, if one compares it with the reconstruction results in Table 4.6, nearest-neighbour outperforms its alternative even though it uses fewer code words.

The data from the tables are visualized in more fine-grained way in Figure 4.6 and Figure 4.7, we can clearly see more narrow token distribution for nearest-neighbours and in both cases, very different distributions for each cut.

4.3.1 Evaluation summary

In this section, we have presented the results of our extensive experiments regarding the reconstruction performance of motion autoencoder, as well as some preliminary results on motion generation. We will try to improve those in the next section. In order to do that, we need to select a few models at maximum to consider.

Based on the results, we summarize the following

1. All 4 combinations of encoder/decoder and latent blocks appear to be viable but not exactly the same, therefore we will try them all.

2. 4,8 cuts offer a reasonable trade-off between reconstruction error and preliminary motion generation performance. Furthermore, 4 cuts keep token sequences shorter therefore we will use them as our go-to model in the next section.

3. L_{Joint} combined with L_{6d} is a viable substitution for L_{Vert}, we use the ratio of $1 : 0.5$, as we have found that generation with $1 : 1$ has in our opinion more trouble with a global orientation.

4. Smoothing does not appear to harm performance at all and the upside should be more smooth motions. But the most important take-away from this is the confirmation that our architecture, the discretization in particular, is not the cause of non-smooth generated motions.

5. At first glance, from a quantitative standpoint, the motion generation appears not impacted by the reconstruction performance much, a hopeful sign

Auto	Latent	Cuts	$L_{6D} \downarrow$	$L_{Joint} \downarrow$	$L_{Vert} \downarrow$
attn	gs	4	43.17	16.07	20.74
attn	nn	4	**41.15**	**13.66**	**17.79**
conv	gs	4	55.54	23.48	31.04
conv	nn	4	55.55	21.89	28.59

Table 4.13: **Final autoencoders -** Reconstruction losses of the final autoencoders.

that the predictor is quite robust to the variations in the autoencoders. But we will see shortly that there are indeed important differences if decoding strategies are used. Furthermore, from qualitative results in our supplemental videos we also suspect that the recognizer used to evaluate the predictor is not particularly sensitive to the quality of motions which might skew the quantitative metrics.

6. Attention-based models are unable to generalize to different lengths, this is a considerable disadvantage compared to convolutional alternatives which perform equally over the tested range(Figure 4.5). It means that to properly decode the generated tokens longer than what the autoencoder model was trained on, the straightforward approach is not enough. Nevertheless, we still investigate the motion prediction capabilities of attention-based models.

In conclusion, we have trained final autoencoders, their results are shown in Table 4.13, and we will use them throughout the next section for motion prediction. Precise setup can be found in the attached source code.

4.4 Evaluating Motion Prediction

With the autoencoders trained (Table 4.13), we can focus on finding the best motion generation models. First, we show the baseline results, then we investigate the effects of the main predictor's hyper-parameters and more importantly the decoding strategies presented in Chapter 3. After that, we compare our models with the state of the art on the UESTC dataset - Table 4.18, showing both quantitative and qualitative results which we discuss.

Implementation details

Unless explicitly stated, the presented results are measured on the UESTC test split with the same setup as in Section 4.3 - showing 95% confidence intervals over ten evaluations.

We use the SOS-variant of our model (Section 3.2), therefore the testing setup is the same as for ACTOR with input being only a category label and therefore fairly comparable.

Consult with the attached source code for the necessary steps to reproduce these results.

We train the models with Adam optimizer, the learning rate of 2.10^{-4}, 10 warm-up steps[53], and batch size of 32 for 250 epochs, on a single NVIDIA V100 the training on the UESTC dataset takes roughly 11 hours.

Baseline results

First, we take the final autoencoders from the last section and train a baseline motion predictor with 12 layers, embedding dimension of 192, and 2048 neurons in fully-connected layers. Decoding directly samples from the output distributions, no special decoding strategies are involved yet. Results of these baseline models are presented in Table 4.14.

The results of course closely match the ones we have revealed during the autoencoder evaluation. We can observe that all models perform very similarly, producing decent results. One can see that our models have slightly lower overall diversity and higher multimodality than the whole UESTC dataset, there is still room for improvement on the accuracy and FID fronts.

In Figure 4.8 we show a few generated motions by these models but we will focus more on the qualitative results later with the final models.

Auto	Latent	Splits	*Acc* ↑	*FID* ↓	*Mmod* → 14.2	*Div* → 33.3
Test GT	x	x	$91.73^{\pm 0.40}$	$0.51^{\pm 0.02}$	$15.88^{\pm 0.22}$	$32.37^{\pm 0.41}$
attn	gs	4	$87.29^{\pm 0.46}$	$14.62^{\pm 0.40}$	$\mathbf{16.22^{\pm 0.18}}$	$31.61^{\pm 0.34}$
attn	nn	4	$\mathbf{87.65^{\pm 0.41}}$	$\mathbf{12.78^{\pm 0.37}}$	$16.31^{\pm 0.17}$	$\mathbf{31.83^{\pm 0.19}}$
conv	gs	4	$86.91^{\pm 0.38}$	$13.42^{\pm 0.45}$	$16.35^{\pm 0.15}$	$31.23^{\pm 0.25}$
conv	nn	4	$86.25^{\pm 0.45}$	$15.39^{\pm 0.36}$	$16.49^{\pm 0.18}$	$31.42^{\pm 0.18}$

Table 4.14: **Baseline motion prediction results -** Motion predictor trained and evaluated on UESTC dataset. Uses start-of-sentence(`SOS`) token only, no initial pose. We also show the ground truth results for the recognizer.

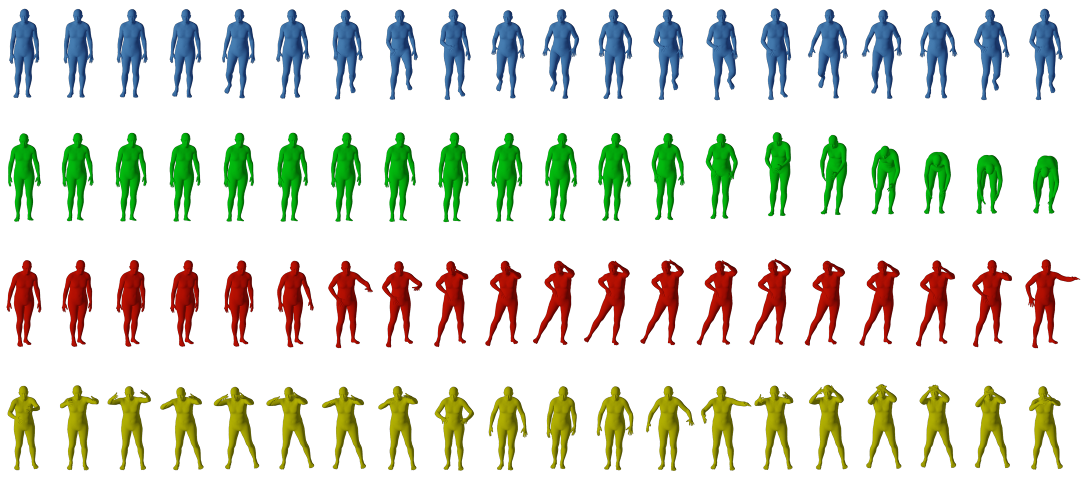


Figure 4.8: **Generated motions from baseline models -** From top to bottom: *running* by `conv nn`, *toe touch* by `conv gs`, *left-stretch* by `attn gs`, and *jumping jacks* by `attn nn`.

Ablating transformer decoder hyper-parameters

In general, we have found the transformer architecture quite robust in training with respect to the number of layers as shown in Table 4.15. It seems that the larger the dimension the better the performance as suggested by the results tabulated in Table 4.16. In retrospect, it looks like we should have used larger 256-dim models as they have slightly better results on the test split, at least for convolutional autoencoders.

Auto	Latent	Layers	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$	$Div \rightarrow 33.3$
attn	nn	4	$84.43^{\pm 0.53}$	$15.35^{\pm 0.44}$	$17.02^{\pm 0.18}$	$30.92^{\pm 0.25}$
attn	nn	6	$86.01^{\pm 0.46}$	$13.51^{\pm 0.53}$	$16.51^{\pm 0.18}$	$31.14^{\pm 0.22}$
attn	nn	8	$86.92^{\pm 0.25}$	$13.48^{\pm 0.45}$	$16.67^{\pm 0.14}$	$31.21^{\pm 0.18}$
attn	nn	10	$87.24^{\pm 0.33}$	$13.06^{\pm 0.48}$	$\mathbf{16.36^{\pm 0.1}}$	$\mathbf{31.4^{\pm 0.34}}$
attn	nn	12	$\mathbf{87.52^{\pm 0.37}}$	$\mathbf{12.97^{\pm 0.64}}$	$16.54^{\pm 0.13}$	$\mathbf{31.44^{\pm 0.26}}$
conv	nn	4	$83.24^{\pm 0.35}$	$17.3^{\pm 0.63}$	$16.95^{\pm 0.12}$	$30.54^{\pm 0.21}$
conv	nn	6	$84.54^{\pm 0.43}$	$15.74^{\pm 0.45}$	$16.79^{\pm 0.19}$	$30.86^{\pm 0.22}$
conv	nn	8	$85.46^{\pm 0.54}$	$15.83^{\pm 0.6}$	$16.6^{\pm 0.13}$	$31.02^{\pm 0.23}$
conv	nn	10	$86.1^{\pm 0.39}$	$\mathbf{15.01^{\pm 0.68}}$	$\mathbf{16.43^{\pm 0.18}}$	$30.95^{\pm 0.26}$
conv	nn	12	$\mathbf{86.26^{\pm 0.38}}$	$15.15^{\pm 0.55}$	$16.55^{\pm 0.14}$	$\mathbf{31.15^{\pm 0.16}}$

Table 4.15: **Performance of transformer layers** - Trained with 192 embedding dimension and 2048 neurons in the fully-connected layers.

Auto	Latent	Dim	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$	$Div \rightarrow 33.3$
conv	gs	64	$52.78^{\pm 0.53}$	$55.36^{\pm 1.21}$	$19.73^{\pm 0.09}$	$26.36^{\pm 0.18}$
conv	gs	128	$83.93^{\pm 0.37}$	$15.01^{\pm 0.33}$	$16.89^{\pm 0.1}$	$30.35^{\pm 0.26}$
conv	gs	192	$86.91^{\pm 0.38}$	$13.42^{\pm 0.45}$	$16.35^{\pm 0.15}$	$31.23^{\pm 0.25}$
conv	gs	256	$87.74^{\pm 0.38}$	$\mathbf{13.34^{\pm 0.62}}$	$16.27^{\pm 0.15}$	$\mathbf{31.69^{\pm 0.23}}$
conv	nn	64	$53.85^{\pm 0.54}$	$53.08^{\pm 1.64}$	$19.63^{\pm 0.16}$	$26.37^{\pm 0.25}$
conv	nn	128	$82.22^{\pm 0.39}$	$17.42^{\pm 0.64}$	$17.04^{\pm 0.11}$	$30.29^{\pm 0.37}$
conv	nn	192	$86.25^{\pm 0.45}$	$15.39^{\pm 0.36}$	$16.49^{\pm 0.18}$	$\mathbf{31.42^{\pm 0.18}}$
conv	nn	256	$\mathbf{87.13^{\pm 0.41}}$	$\mathbf{14.07^{\pm 0.57}}$	$\mathbf{16.46^{\pm 0.09}}$	$\mathbf{31.39^{\pm 0.33}}$

Table 4.16: **Exploring embedding dimensionality in Transformer** - The table shows the effect o varying size of the embedded token dimension on motion generation. Trained with 12 layers and 2048 neurons in the fully-connected layers.

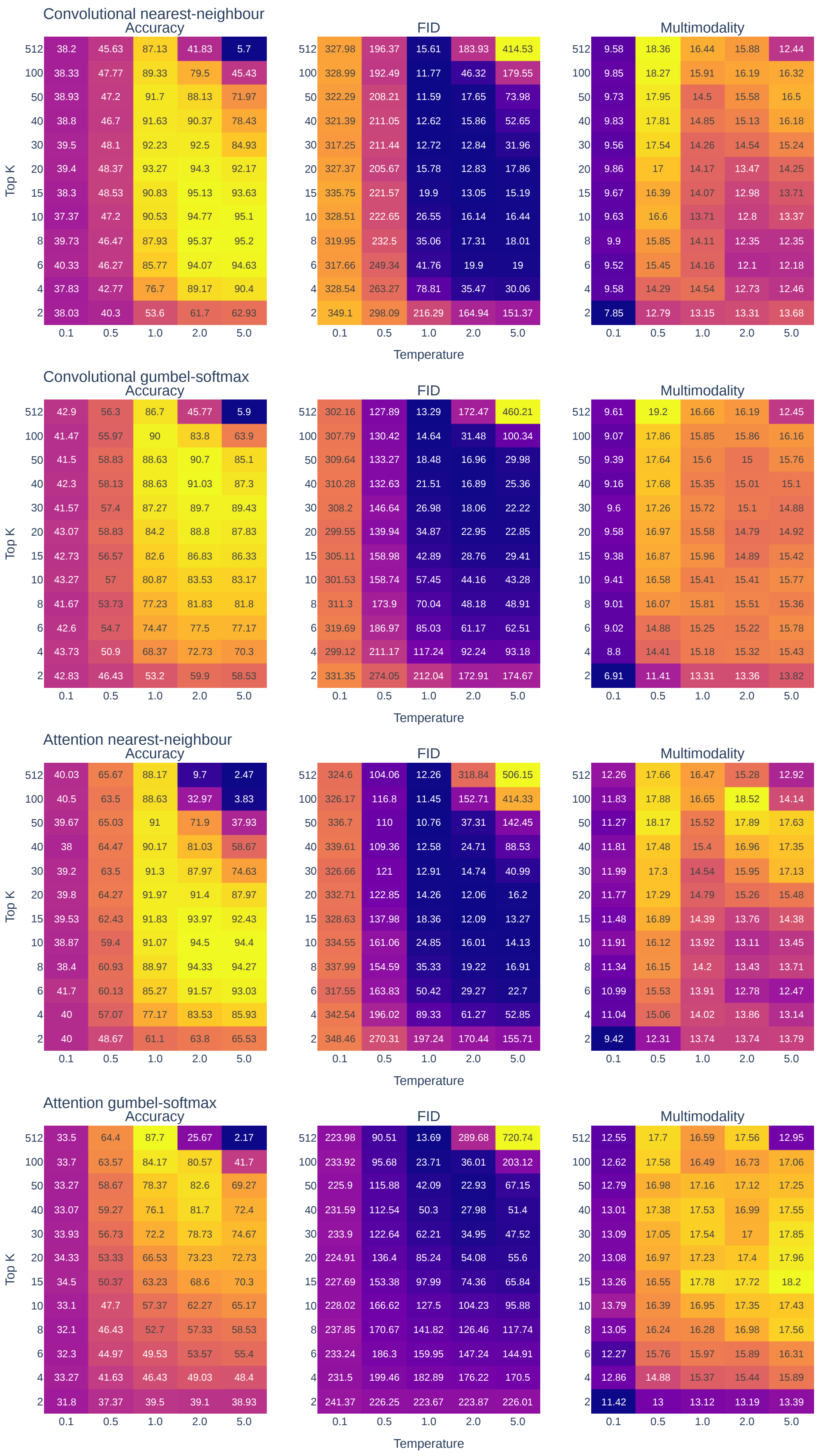


Figure 4.9: **Top-k vs temperature sampling survey -** Quantitative performance of baseline motion predictors from Table 4.14. Note that due to evaluation costs, these results are from 1 iteration only, interesting values are re-plotted in Figure 4.10.

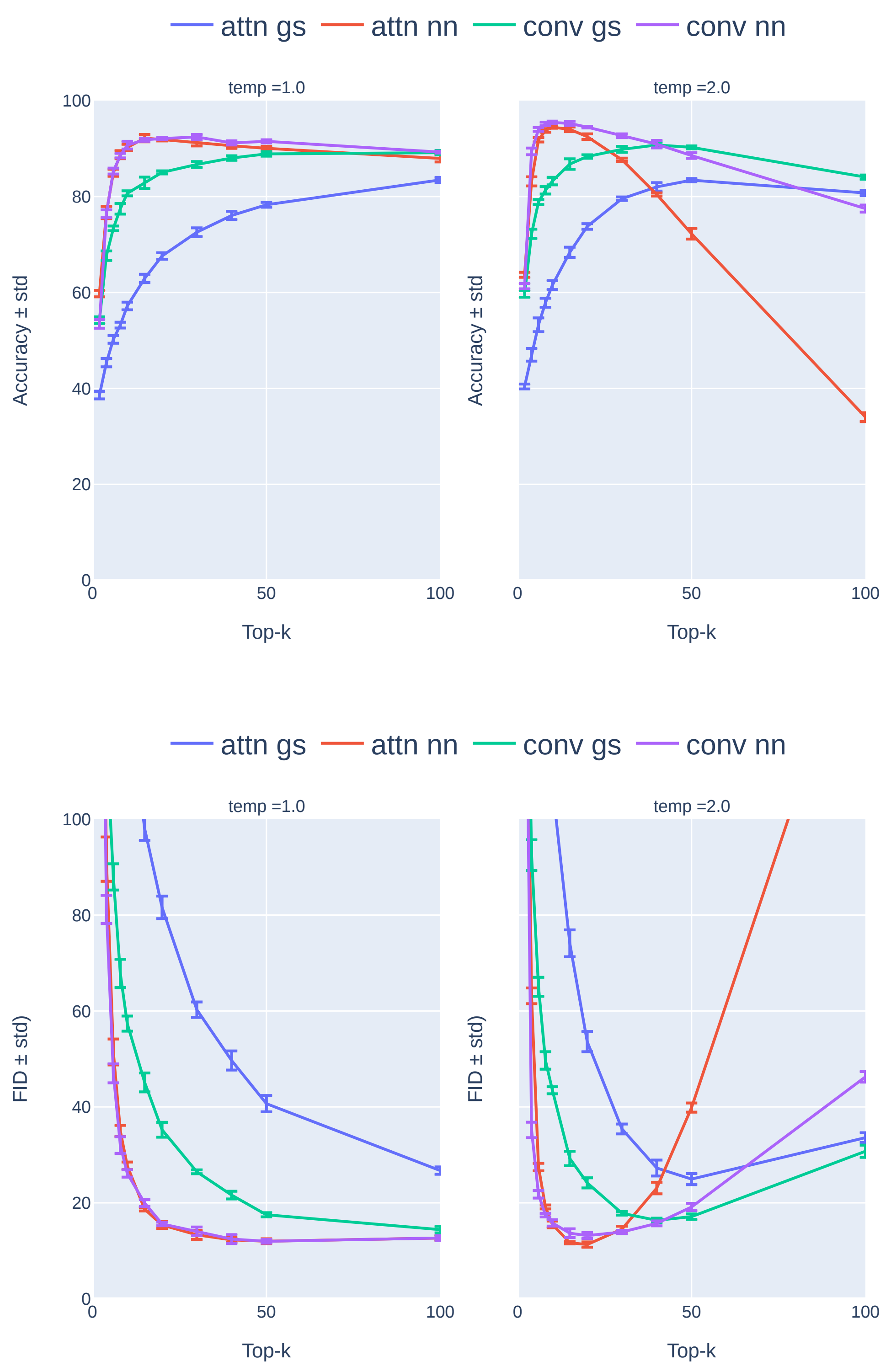


Figure 4.10: **Effect of top-k on accuracy and FID for motion generation** - Following on sampling survery in Figure 4.9, this time we evaluated the models 5 times with different values of top-k. Top figure shows the effect on accuracy, the bottom one on FID.

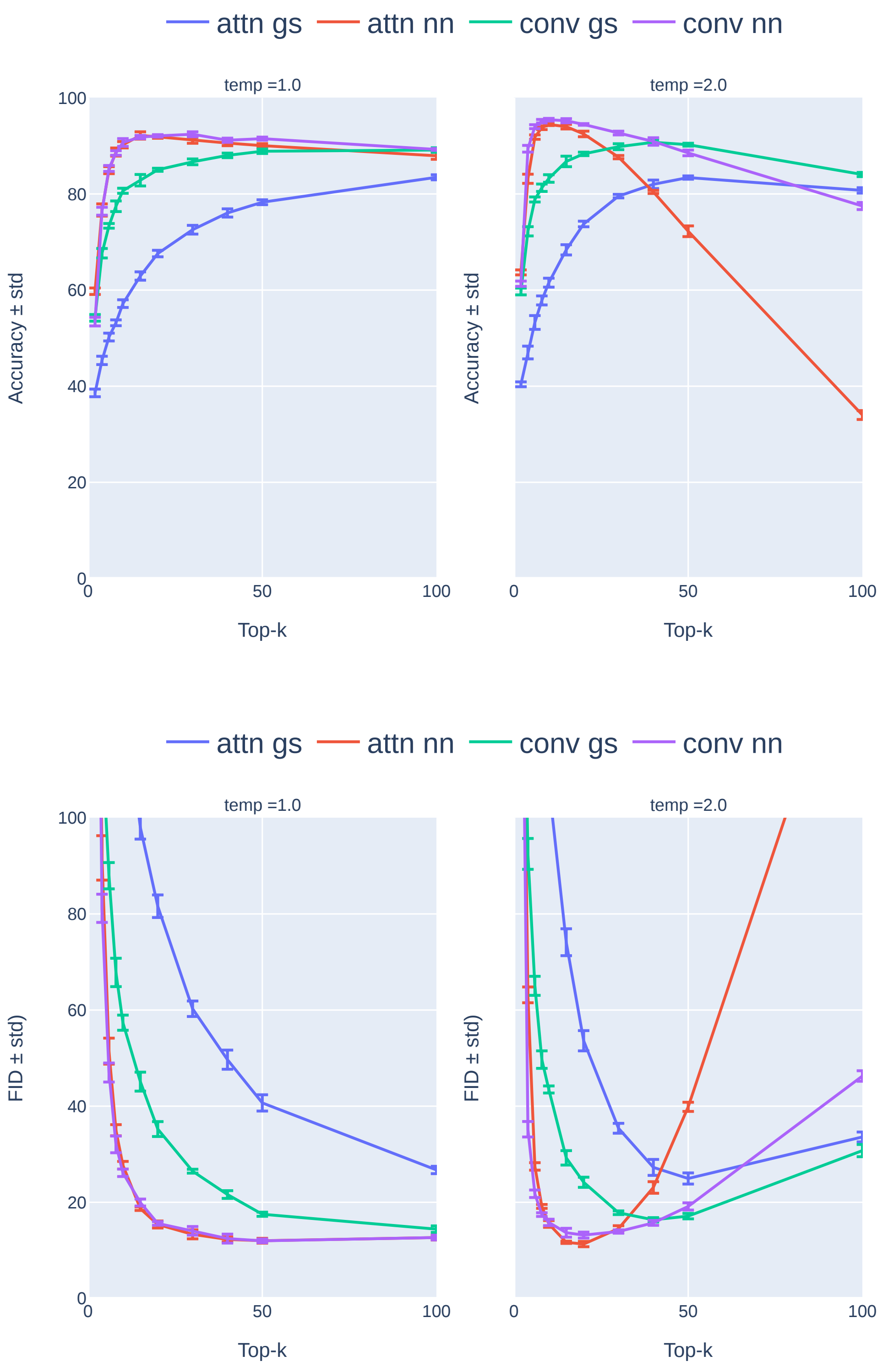


Figure 4.11: **Effect of nucleus sampling on accuracy and FID of generated motions -** Performance of models from Table 4.14 with nucleus sampling as the chosen decoding strategy with probability threshold p.

Decoding strategies

In Section 3.2, we have described some of the possible decoding strategies, mainly used to improve language generation. Therefore we have conducted a series of experiments to investigate their impact on our models.

First, we look at the impact of top-k sampling combined with varying the temperature of softmax. For all 4 models and the qualitative metrics[3], we have plotted a series of graphs depicting the performance of the models in Figure 4.9.

Our observations:

- Extreme temperatures heavily degrade performance because they drastically change the probability distribution, which was expected.
- Multimodality, in general, correlates with higher top-k values which is also reasonable because the model has more options to choose from, hence more diverse motions. UESTC test split has a multimodality of 14.2, we can see that our models can and do overshoot it in some scenarios.
- There is a fundamental difference between the gumbel-softmax and nearest-neighbour model as to where their optima lie. Regarding FID, for the former latent block, higher top-k appears to improve the metric, meaning this technique is not very beneficial for this architecture. On the other hand, nearest-neighbour strongly benefits from tighter top-k bounds, with FID optima laying around `top-k`== 15 and `temperature`== 2.0. Accuracy is similar, although we can see the accuracy of `attn-gs` benefits to some degree from lower top-k.

Based on these results, we see `attn-nn` as the most promising candidate with the option of tunable performance towards higher accuracy and lower FID for the top-k range $[8 - 20]$ and temperature around 2.0. `conv-nn` is very comparable accuracy-wise but has slightly higher FID values. Unfortunately, gumbel-softmax has trouble breaking 90% accuracy threshold with any settings, with naive settings working the best.

We re-plot the accuracy and FID data again Figure 4.10 for the interesting temperatures and evaluate the same settings 5 times, it appears the values do not change very much and therefore our usage of the values from Figure 4.9 for our argumentation is valid.

Second, we look at how top-p sampling could improve the results. Unfortunately, as it appears from Figure 4.11, it cannot. This is in contrast with [14] which reports success with top-p sampling as well as the original paper[43] We are unsure about the exact cause.

The last technique we have described was scheduled sampling, acting as a way to get the model used to see its own outputs. Therefore we have taken our trained baseline models and fine-tuned them for another 50 epochs with scheduled sampling, p annealed over the first 20 epochs $0 \rightarrow 0.5$. Evaluated results are shown in Table 4.17, if we compare them to Table 4.14, the only major thing is the improvement of FID for `conv nn` architecture. Other than that, the qualitative results look quite similar or even worse (the case of FID for `attn nn`), which

[3]We do not plot diversity due to space constraints as well as because it is mostly covered by multimodality.

Auto	Latent	Splits	$Acc\uparrow$	$FID\downarrow$	$Mmod \rightarrow 14.2$	$Div \rightarrow 33.3$
attn	gs	4	$88.34^{\pm 0.37}$	$14.15^{\pm 0.61}$	$16.01^{\pm 0.12}$	$\mathbf{31.97^{\pm 0.37}}$
attn	nn	4	$86.94^{\pm 0.35}$	$13.58^{\pm 0.55}$	$16.41^{\pm 0.17}$	$31.59^{\pm 0.33}$
conv	gs	4	$87.78^{\pm 0.37}$	$13.35^{\pm 0.43}$	$16.16^{\pm 0.11}$	$31.21^{\pm 0.26}$
conv	nn	4	$\mathbf{88.43^{\pm 0.28}}$	$\mathbf{12.74^{\pm 0.48}}$	$\mathbf{15.82^{\pm 0.14}}$	$31.42^{\pm 0.34}$

Table 4.17: **Scheduled motion prediction results** - Baseline motion predictors fine-tuned using scheduled sampling technique.

is unfortunate, varying top-k and temperature also do not improve this training technique as we can see in Figure 4.12.

During our experimentation earlier, when our results were much worse with FID around 50 and accuracy barely reaching 80%, scheduled sampling appeared much more favourable to us. It did improve the results noticeably, especially the generated motions were more stable and less prone to degradation or collapse. But as we have shown now, there is no longer much difference with the current models we have managed to train without this technique.

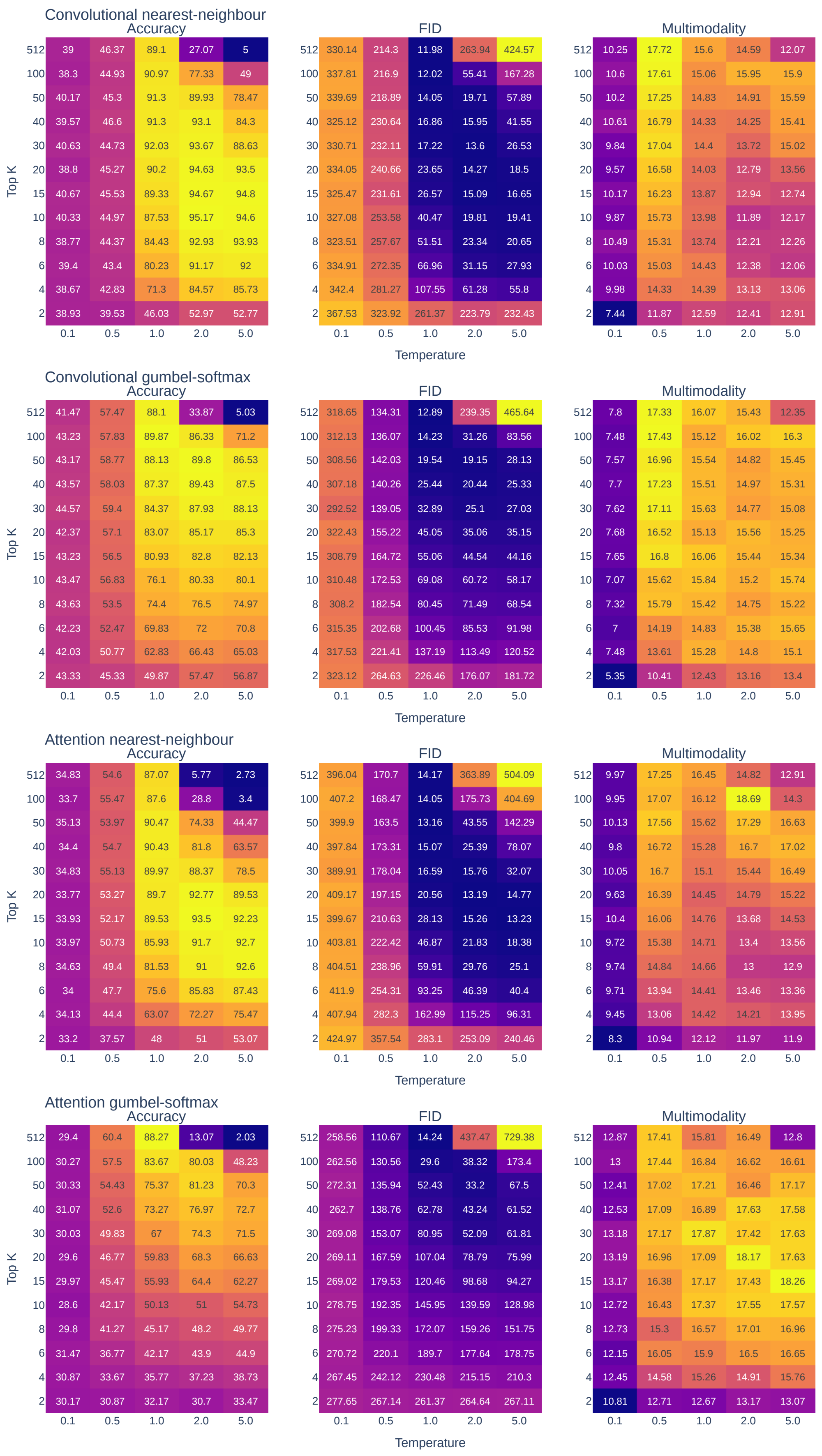


Figure 4.12: **Top-k vs temperature sampling survey** - Quantitative performance of fine-tuned motion predictors from Table 4.17.

Model	Temp	Top-k	$Acc \uparrow$	$FID_test \downarrow$	FID_{tr}	$Mmod \rightarrow 14.16$	$Div \rightarrow 33.34$
GT[3]	x	x	$98.80^{\pm 0.10}$	$2.79^{\pm 0.29}$	x	$14.16^{\pm 0.06}$	$33.34^{\pm 0.32}$
Test GT	x	x	$91.73^{\pm 0.40}$	$0.51^{\pm 0.02}$	x	$15.88^{\pm 0.22}$	$32.37^{\pm 0.41}$
ACTOR[3]	x	x	$91.1^{\pm 0.30}$	$23.43^{\pm 2.20}$	$20.49^{\pm 2.31}$	$\mathbf{14.66^{\pm 0.03}}$	$31.96^{\pm 0.29}$
conv gs	1.0	512	$87.11^{\pm 0.31}$	$13.58^{\pm 0.55}$	$13.52^{\pm 0.47}$	$16.43^{\pm 0.15}$	$31.26^{\pm 0.33}$
attn gs	1.0	512	$87.85^{\pm 0.25}$	$14.56^{\pm 0.49}$	$14.27^{\pm 0.31}$	$16.24^{\pm 0.11}$	$31.68^{\pm 0.20}$
conv nn	2.0	20	$\mathbf{94.45^{\pm 0.36}}$	$12.84^{\pm 0.39}$	$12.88^{\pm 0.40}$	$13.45^{\pm 0.11}$	$34.14^{\pm 0.27}$
conv nn	1.0	512	$86.25^{\pm 0.45}$	$15.39^{\pm 0.36}$	$14.44^{\pm 0.48}$	$16.49^{\pm 0.18}$	$31.42^{\pm 0.18}$
attn nn	2.0	20	$92.23^{\pm 0.26}$	$\mathbf{11.29^{\pm 0.28}}$	$\mathbf{11.02^{\pm 0.39}}$	$14.68^{\pm 0.14}$	$\mathbf{33.51^{\pm 0.32}}$
attn nn	1.0	512	$87.65^{\pm 0.41}$	$12.78^{\pm 0.37}$	$12.82^{\pm 0.49}$	$16.31^{\pm 0.17}$	$31.83^{\pm 0.19}$

Table 4.18: **Comparison with the state of the art** - Comparison of our best models with the state of the art ACTOR on UESTC dataset, ground truth(`GT`) is given by evaluating the predictor directly on motions from the test set or on all motions(`UESTC`).

Auto	Latent	Temp	Top-k	$Acc \uparrow$	$FID \downarrow$	$Mmod \rightarrow 14.2$	$Div \rightarrow 33.3$
conv	gs	1.0	512	$85.04^{\pm 0.5}$	$17.36^{\pm 0.43}$	$16.52^{\pm 0.11}$	$31.1^{\pm 0.41}$
attn	gs	1.0	512	$88.43^{\pm 0.32}$	$13.3^{\pm 0.43}$	$15.98^{\pm 0.11}$	$31.97^{\pm 0.28}$
conv	nn	2.0	20	$90.92^{\pm 0.39}$	$17.43^{\pm 0.34}$	$\mathbf{14.65^{\pm 0.14}}$	$33.01^{\pm 0.19}$
attn	nn	2.0	20	$\mathbf{92.17^{\pm 0.32}}$	$\mathbf{12.73^{\pm 0.26}}$	$\mathbf{14.66^{\pm 0.08}}$	$\mathbf{33.4^{\pm 0.35}}$

Table 4.19: **Motion predictors with smoothed autoencoders** - Trained same motion predictors as in Table 4.14 using the same autoencoders except with added smoothing weight of 10^{-4}.

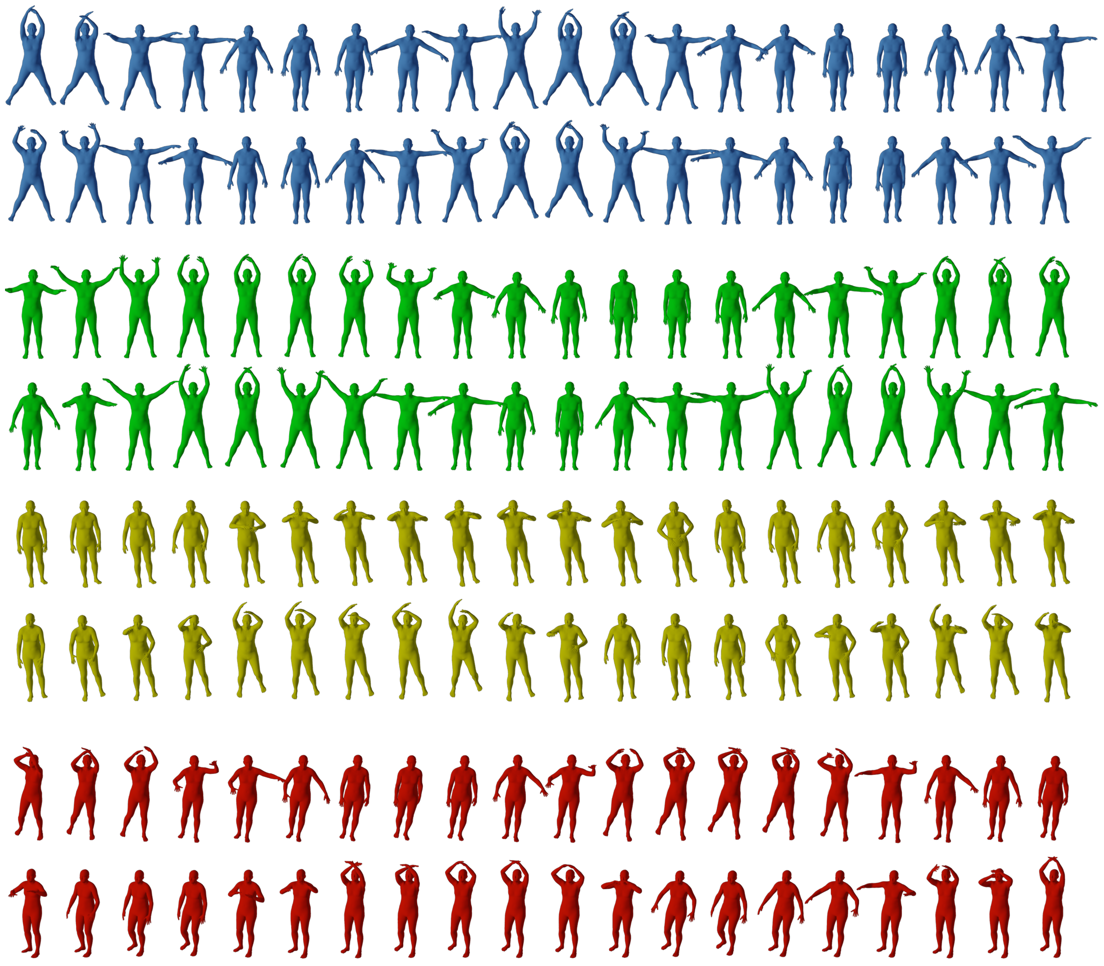


Figure 4.13: **Jumping jacks generation** - Jumping jacks as generated by final motion predictors, 2 rows(samples) per model, top-to-bottom: *conv nn*, *conv gs*, *attn nn*, *attn gs*.

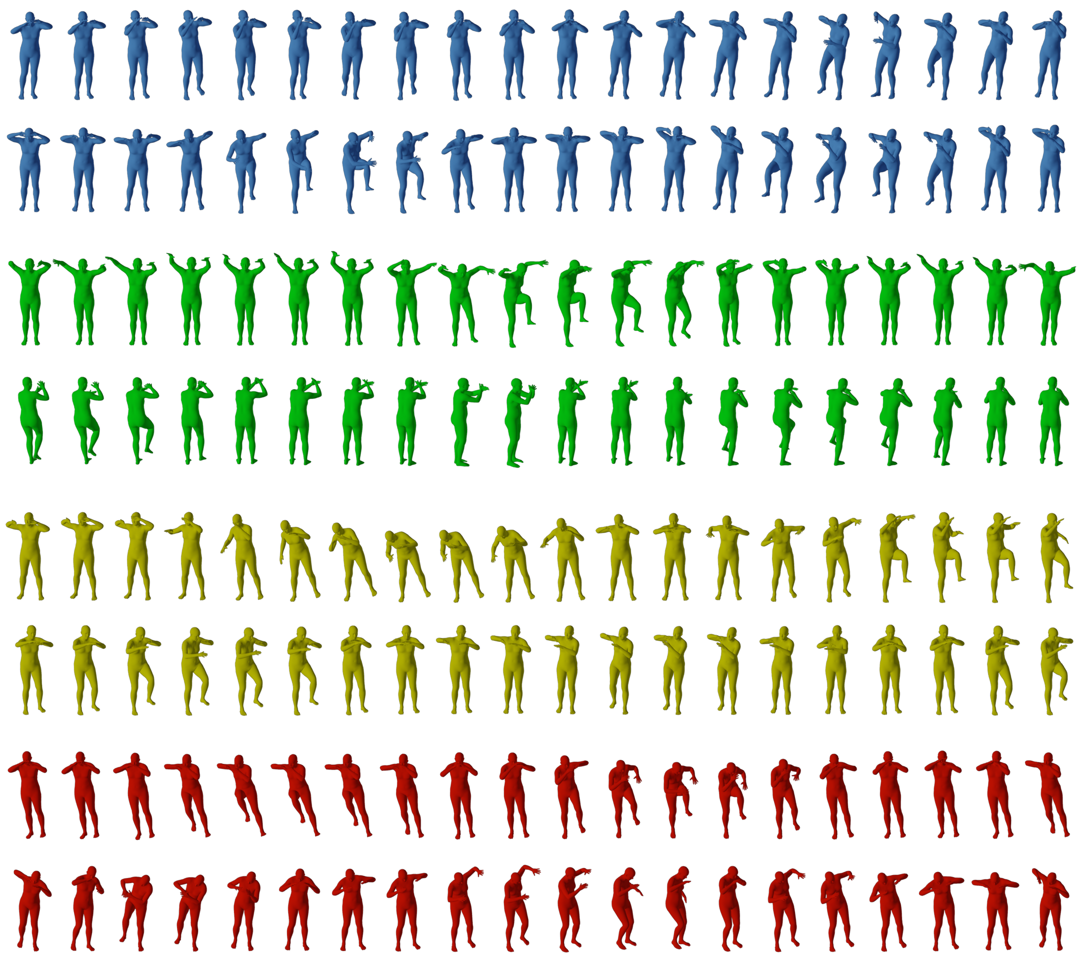


Figure 4.14: **Elbow knee crunches generation** - Elbow knee crunches as generated by final motion predictors, 2 rows(samples) per model, top-to-bottom: *conv nn*, *conv gs*, *attn nn*, *attn gs*.

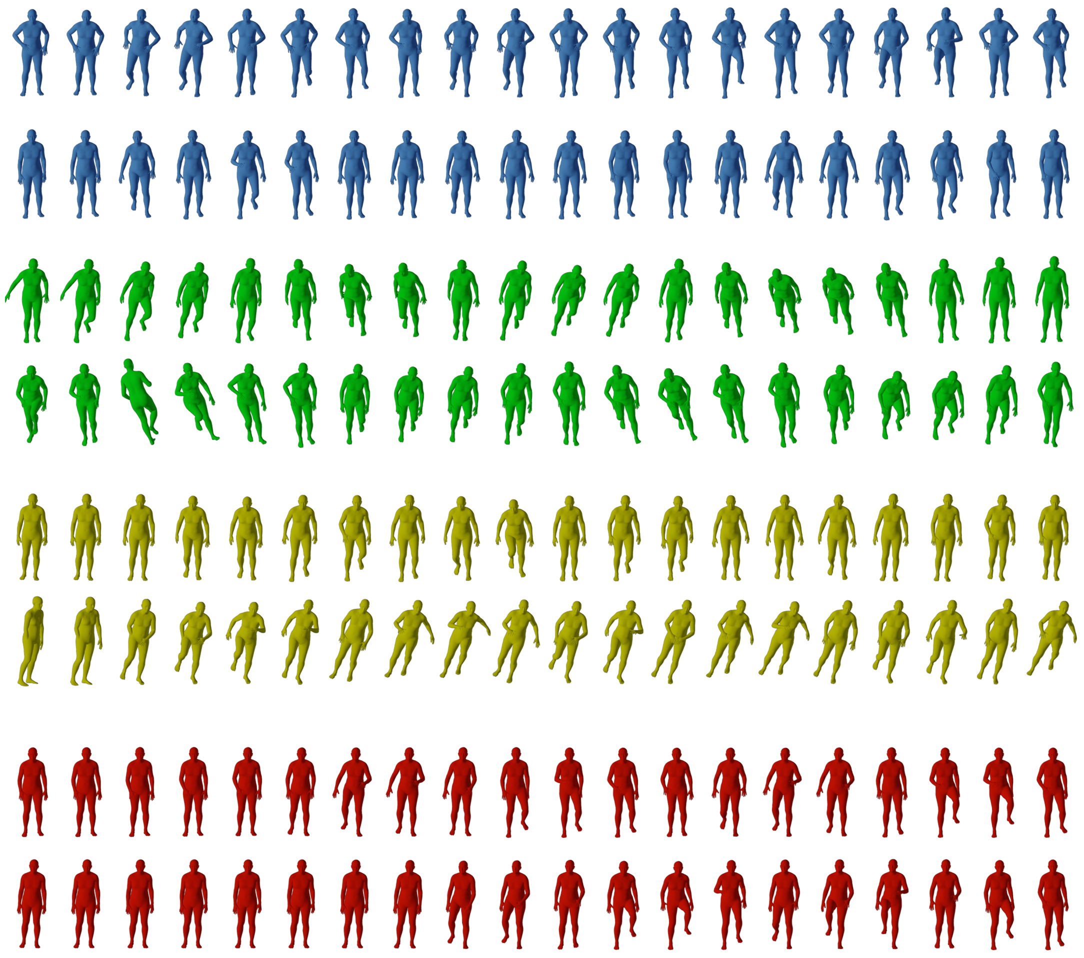


Figure 4.15: **Running motion generation** - Running motion as generated by final motion predictors, 2 rows(samples) per model, top-to-bottom: *conv nn*, *conv gs*, *attn nn*, *attn gs*.

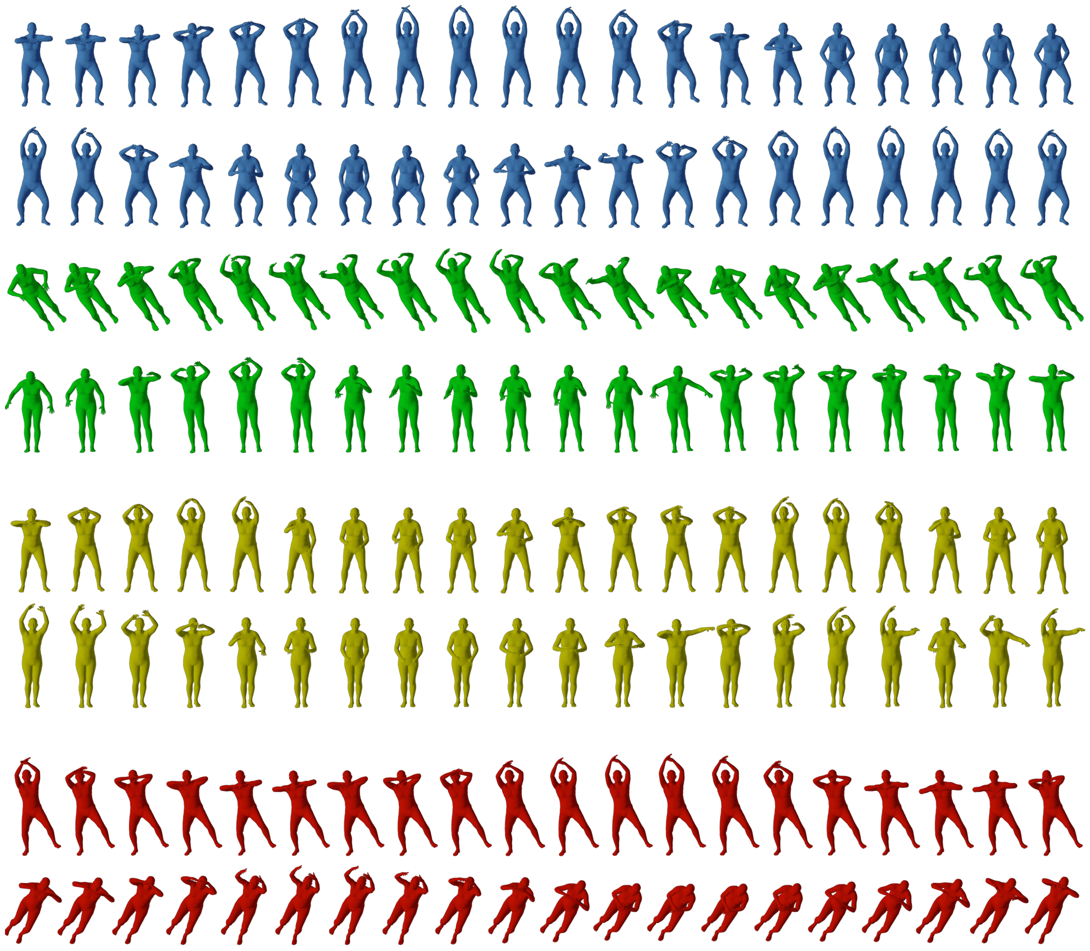
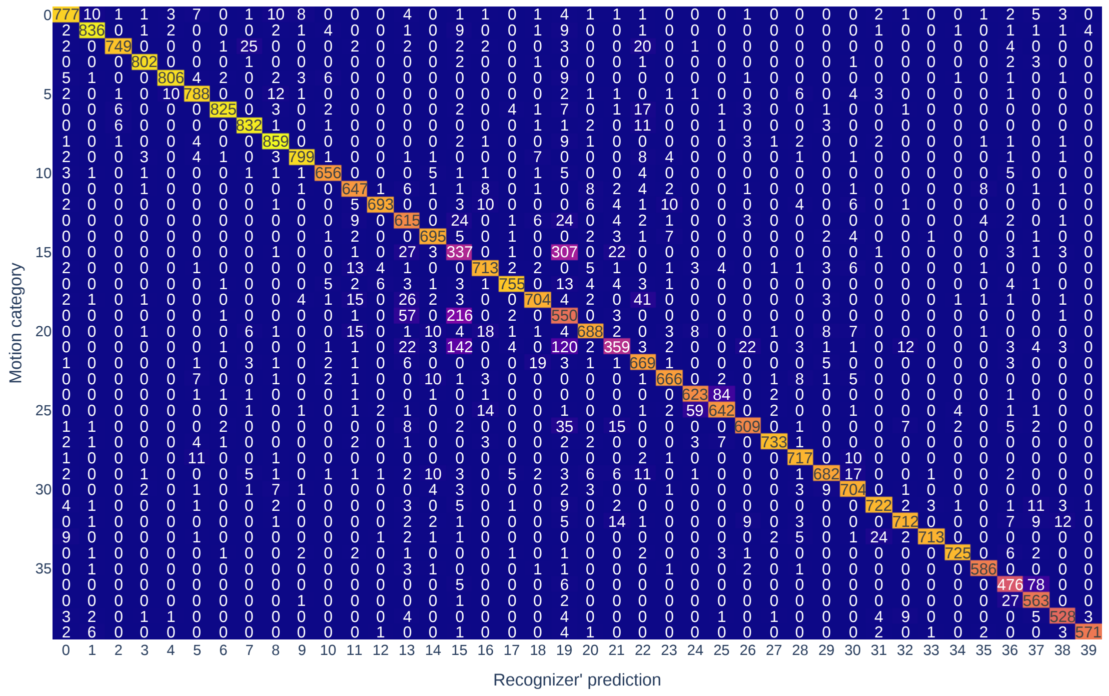


Figure 4.16: **Overhead stretching motion generation** - Overhead stretching motion as generated by final motion predictors, 2 rows(samples) per model, top-to-bottom: *conv nn*, *conv gs*, *attn nn*, *attn gs*.



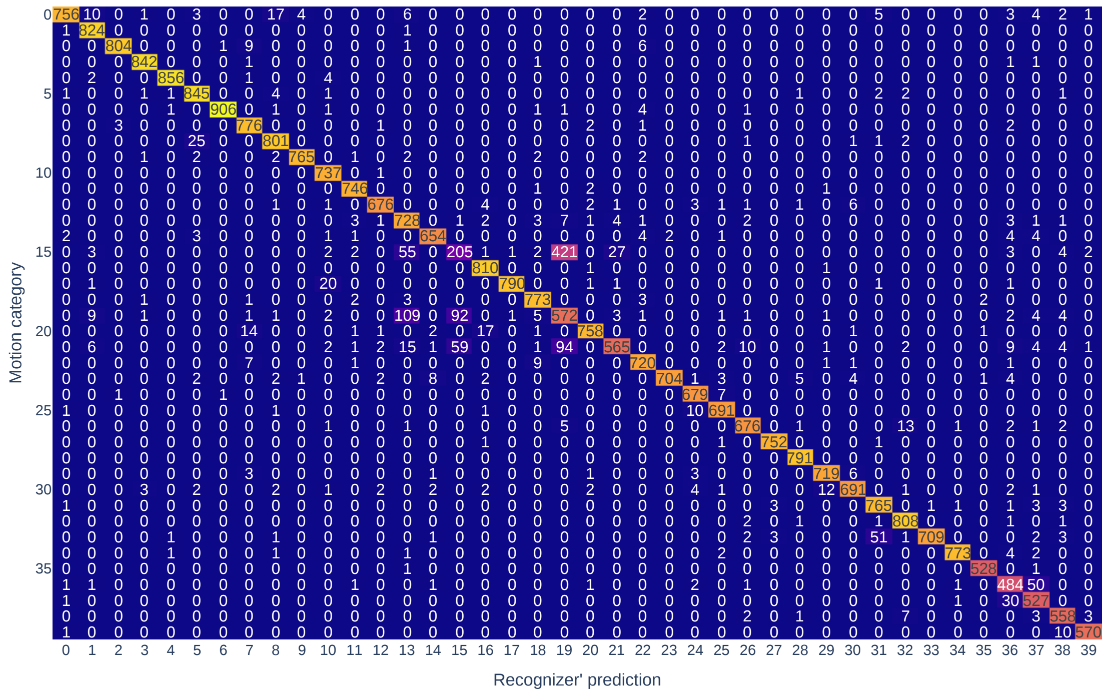


Figure 4.17: **Confusion matrices and outliers of motion recognition for `conv nn` model** - The matrices obtained from the recognizer when evaluating a motion predictor which uses convolutional, nearest-neighbour autoencoder. Top figure shows `conv nn` with ordinary softmax decoding while the bottom one uses modified top-k, temperature values from Table 4.18.

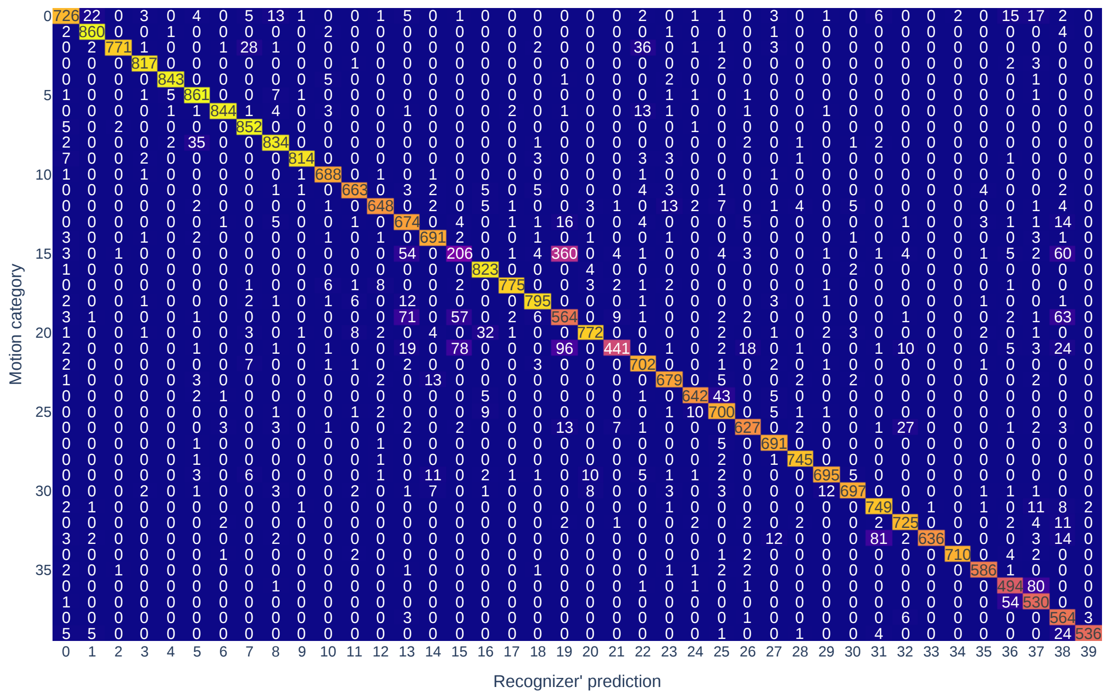


Figure 4.18: **Confusion matrix of motion recognition for `attn nn` model** - Attention-based nearest-neighbour model evaluated on UESTC test set with top-k= 20 and 2.0 temperature.

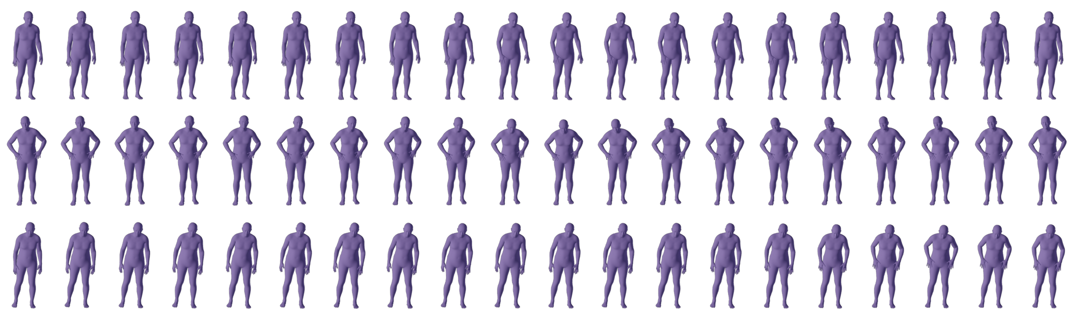


Figure 4.19: **Three similar, problematic categories** - The figure depicts three motions from the UESTC test split - categories 15, 19, and 21 which have proven troublesome for our models as well as the recognizer itself. We leave it to the reader to see which are which.

4.4.1 Comparison to the state of the art ACTOR

Table 4.18 is the table, it compares our best models with the state of the art ACTOR's results. ACTOR claims[3] FID to be the main metric for optimization, our models consistently outperform ACTOR with the possibility for targeting high-accuracy. As can be seen from multimodality and diversity values, the better metrics are not at expense of the ability to generate diverse motions. In figures 4.13, 4.14, 4.15, and 4.16 we show qualitative results from our models, for the nearest-neighbour variants we use the version with top-k= 20 from Table 4.18. Please also refer to our supplemental videos with direct comparisons.

Tabulating FID for the training set is also very important because it controls for the overfitting which would likely lead to very high, but meaningless, scores by the recognizer. We can see that our model does not overfit any more than ACTOR does.

Based on the figures above, we can see how models based on gumbel-softmax do not achieve the best results. It is not due to the autoencoder's quality per se because that is comparable to its nearest-neighbour alternative and it cannot be the fault of the transformer architecture because that is also unchanged. Therefore the lower performance must come directly from the hardness of the token dataset implied by the autoencoder. Maybe due to the mentioned fact of independently sampled tokens on a per-frame basis. We do note that our models are not perfect, sometimes, not often, they have trouble with global orientation or they do not perform any action or perform it badly. We believe this is mainly due to its autoregressive nature and admit that ACTOR is superior with respect to its output stability, at least from our experiments with it. Moreover, not only from the shown samples, we do no longer consider `attn nn` the best model despite its metrics exactly due to its tendency to fail at a higher rate than other models. Despite slightly worse metrics, from a qualitative standpoint, **we claim `conv nn` to be our best model for motion generation and now look at its performance more closely.**

To better understand the qualitative results and provide something better than subjective opinions, in Figure 4.17 we show the confusion matrices for action recognition of motions generated by `conv nn`. We can see the diagonal of correctly generated/recognized motions and some incorrect ones. But if we compare these metrics with the motions shown in Figure 4.18, we see a peculiar fact. Even though `attn nn, top-k=20` has better FID than its convolutional counterpart (Table 4.18), it has considerably worse confusion matrix. This should explain the results we saw when generating motions from the models - the higher instability of the attention-based model. In both tables, there are clear outliers - categories 15,19, and 21. They stand for *shoulder-raise*, *dumbbell-shrugging*, *head-circling* respectively. If we plotted the confusion matrix for the test set itself, these three would still stand out, even the recognizer cannot reliably distinguish between them. We do show samples of them in Figure 4.19 and it should be clear how similar they are. Our accuracy is mostly limited exactly because the standing-like motions get miscategorized.

A point we have touched on earlier is how smooth are the motions generated by the ACTOR model despite the non-smoothness of the dataset. Unfortunately, as we have seen from motion reconstruction, our model does not have this property. We have tried to replicate it with the introduction of smoothing loss into

Auto	Latent	Temp	Top-k	*Acc* ↑	*FID* ↓	*Mmod* → 14.2	*Div* → 33.3
attn	gs	1.0	512	$\mathbf{90.31^{\pm 0.33}}$	$26.54^{\pm 0.44}$	$\mathbf{14.24^{\pm 0.1}}$	$\mathbf{32.32^{\pm 0.26}}$
attn	nn	1.0	512	$89.29^{\pm 0.25}$	$31.52^{\pm 0.67}$	$14.47^{\pm 0.19}$	$31.47^{\pm 0.24}$
conv	gs	1.0	512	$90.08^{\pm 0.15}$	$32.81^{\pm 0.54}$	$13.98^{\pm 0.08}$	$30.84^{\pm 0.22}$
conv	nn	1.0	512	$86.32^{\pm 0.37}$	$\mathbf{23.38^{\pm 0.62}}$	$15.41^{\pm 0.15}$	$31.79^{\pm 0.27}$

Table 4.20: **Motion predictors with A-UESTC autoencoders** - Trained the same motion predictors as in Table 4.14 using autoencoders trained on A-UESTC. The predictor models themselves are evaluated against the original UESTC dataset.

autoencoder training, therefore we have also trained motion predictors using these autoencoders. The results analogous to Table 4.18 are shown in Table 4.19, we do not observe any major degradation nor improvement. It is difficult to show the effect on paper therefore we have generated a few supplemental videos comparing smoothed and ordinary motion generation.

Let us, for now, concede the point that even the smoothed versions of our models are no match for the smoothness of ACTOR, our model is simply unable to "repair" the input dataset and instead shows the motions as they are, we do not consider it a disadvantage. Yet, we do not need to stop here, as the ACTOR paper's authors point out, that smoothing datasets is one of the use cases of their model, let us use it then.

We trained the same final autoencoders but on the new A-UESTC dataset (Section 4.1) and the corresponding new motion predictors with their results tabulated in Table 4.20. Few motions are depicted in Figure 4.20 but also refer to supplemental videos to get the best picture. The videos prove that our model, if given a smooth dataset is more than capable of generating smooth motions. The tabulated qualitative results show worse FID numbers, this should be expected as neither the autoencoder nor the predictor have seen the original UESTC dataset, therefore they are more supplemental and a check that the generation works. On the other hand, the accuracy is better than for the generated methods.

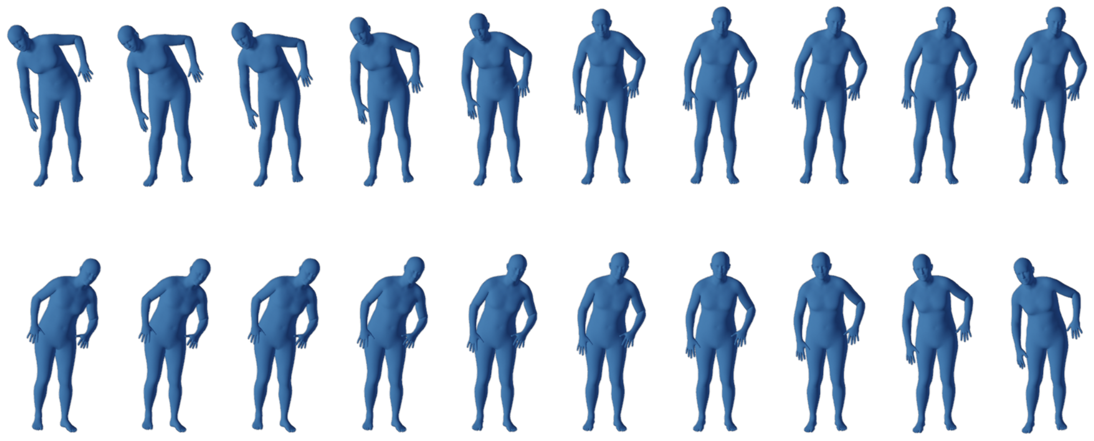


Figure 4.20: **Motion generated by `conv nn` model trained on A-UESTC dataset -** Our models can indeed generate smooth motions if given smooth dataset. The figure depicts two motions of humans bending to sides. Best viewed as the supplemental videos.

5. Conclusion

We introduced a new model for motion generation conditioned on categorical labels. It is based on the two-stage approach of first learning discrete representation of unlabeled motions and then training a generative, autoregressive model conditioned on the category and possibly the starting motion. Throughout Chapter 4 we did subject the model to extensive testing including the evaluation of the presented architectural variations and ablations of the main hyper-parameters. We provided our results, proving that using discrete representation of motions is a viable technique and it can achieve state of the art qualitative results on the UESTC dataset as well as being capable of generating high-quality, diverse motions.

Limitations

Although performing well in our opinion, our model is not without issues as well as having plenty of potential avenues for improvements we thought of during the work on this thesis, we now summarize the major ones.

- Attention-based autoencoder are not suited for variable-length motion generation/decoding. But this should be solvable in practice, albeit it must involve a more complex approach. Because we could also restrict the autoencoder to a similar sliding window as the one employed during the generation. Alternatively, we can partition the tokens into partly overlapping fixed-length sequences and decode them independently, likely interpolating the decoded motion at subsequences' edges to produce a smooth connection.
- Our prediction model lacks the ability to smooth the input dataset, technically not a disadvantage in our opinion as we have discussed earlier. We have tried to fix it with the introduction of the smoothing loss but we have fallen short of the ACTOR model. Nevertheless, it can be *fixed* with the help of some external smoothing pre-processing step(the case of A-UESTC). Still, we feel the need to mention this here explicitly for fairness as well as stating the fact that ACTOR's generation of smooth motions from a non-smooth dataset is not by choice but through a convenient side-effect of their architecture.
- We did not fully eliminate the instability of autoregressive generation, sometimes the generated motion is simply not good, but any future advances in decoding strategies or training of for example language-focused tasks can lead to downstream improvements even in our model. We believe the ability to generate longer sequences in a controlled fashion outweighs these shortcomings.
- It is our opinion through extensive experience with our model that training requires more data than ACTOR, a possible solution might come from the two-stage approach and pre-training on a possibly unlabeled dataset, or alternatively borrowing pre-training techniques once again from the language generation domain.

Future work

We hope that our introduction of discrete representation to the motion generation domain can offer a wide set of future avenues of research, the most logical ones which could follow are:

- Analyze performance on other datasets which we were not able to do mostly due to time constraints. A very intriguing idea we would have hoped to see is pre-training the autoencoder on a large-scale dataset like AMASS[2] and even possibly the motion predictor without conditioning on categories.
- Add the prediction of poses' global positions, we believe this might help with the global orientation issues of some of the presented models and of course, add further applications to the model. Our idea would be to add it as a special 5th token after each pose if there are 4 cuts. It would not be a token per se, just a vector with the same dimensions as the embedded tokens, likely 3D coordinates passed through a linear dense layer. The 5th output token would then not be considered discrete but instead would represent the parameters of a Gaussian distribution of the global position from which we could sample. This would create a sort of hybrid discrete-continuous generative model but that is worth investigating.
- Exploring more advanced transformer models, perhaps those with linear attention[46, 47] would allow attending to wider past context and therefore hopefully allow the model to generate better motions.
- Many-times mentioned [14, 15] papers condition their generations on different inputs, we, in particular, are interested in conditioning on sentences as few of the related works in Chapter 2 do. For example, [54] presents the TEMOS model, a follow-up model on ACTOR from its authors, capable of generating motions based on short sentences (uses the KIT dataset[1]) with reportedly considerably improving the state of the art results. Even in our case, thanks to the two-stage approach, it might be achieved by simply employing a proper encoder-decoder transformer architecture[21].
- There have been further developments in the methods we use, VQ-VAE2[28] adding on an hierarchical approach, DALL-E2[29] focusing more on diffusion models and we found the paper[55] introducing Robust VQ-VAE dealing with over-representation of outliers in the generated data quite interesting, especially coupled with the fact of non-perfect motion datasets.

Bibliography

[1] Matthias Plappert, Christian Mandery, and Tamim Asfour. The kit motion-language dataset. *Big Data*, 4(4):236–252, 2016. PMID: 27992262.

[2] Naureen Mahmood, Nima Ghorbani, Nikolaus F. Troje, Gerard Pons-Moll, and Michael J. Black. AMASS: Archive of motion capture as surface shapes. In *International Conference on Computer Vision*, pages 5442–5451, October 2019.

[3] Mathis Petrovich, Michael J. Black, and Gül Varol. Action-conditioned 3D human motion synthesis with transformer VAE. In *International Conference on Computer Vision (ICCV)*, pages 10985–10995, October 2021.

[4] Chaitanya Ahuja and Louis-Philippe Morency. Language2pose: Natural language grounded pose forecasting. *2019 International Conference on 3D Vision (3DV)*, pages 719–728, 2019.

[5] Chuan Guo, Xinxin Zuo, Sen Wang, Shihao Zou, Qingyao Sun, Annan Deng, Minglun Gong, and Li Cheng. Action2motion. In *Proceedings of the 28th ACM International Conference on Multimedia*. ACM, oct 2020.

[6] Ruilong Li, Sha Yang, David A. Ross, and Angjoo Kanazawa. Ai choreographer: Music conditioned 3d dance generation with aist++. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 13381–13392, 2021.

[7] Jiashun Wang, Huazhe Xu, Jingwei Xu, Sifei Liu, and Xiaolong Wang. Synthesizing long-term 3d human motion and interaction in 3d scenes. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 9401–9411. Computer Vision Foundation / IEEE, 2021.

[8] Jari Kätsyri, Klaus Förger, Meeri Mäkäräinen, and Tapio Takala. A review of empirical evidence on different uncanny valley hypotheses: Support for perceptual mismatch as one road to the valley of eeriness. *Frontiers in psychology*, 6:390, 04 2015.

[9] Félix G. Harvey and Christopher Pal. Recurrent transition networks for character locomotion. In *SIGGRAPH Asia 2018 Technical Briefs*, SA '18, New York, NY, USA, 2018. Association for Computing Machinery.

[10] Félix G. Harvey, Mike Yurick, Derek Nowrouzezahrai, and Christopher Pal. Robust motion in-betweening. 39(4), 2020.

[11] Ying-Sheng Luo, Jonathan Hans Soeseno, Trista Pei-Chun Chen, and Wei-Chao Chen. CARL: controllable agent with reinforcement learning for quadruped locomotion. *ACM Trans. Graph.*, 39(4):38, 2020.

[12] Xue Bin Peng, Yunrong Guo, Lina Halper, Sergey Levine, and Sanja Fidler. Ase: Large-scale reusable adversarial skill embeddings for physically simulated characters. *ACM Trans. Graph.*, 41(4), July 2022.

[13] Sebastian Starke, He Zhang, Taku Komura, and Jun Saito. Neural state machine for character-scene interactions. *ACM Trans. Graph.*, 38(6), nov 2019.

[14] Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 12873–12883. Computer Vision Foundation / IEEE, 2021.

[15] Aditya Ramesh, Mikhail Pavlov, Gabriel Goh, Scott Gray, Chelsea Voss, Alec Radford, Mark Chen, and Ilya Sutskever. Zero-shot text-to-image generation. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event*, volume 139 of *Proceedings of Machine Learning Research*, pages 8821–8831. PMLR, 2021.

[16] Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 6306–6315, 2017.

[17] Guillermo Valle Pérez, Gustav Eje Henter, Jonas Beskow, Andre Holzapfel, Pierre-Yves Oudeyer, and Simon Alexanderson. Transflower: probabilistic autoregressive dance generation with multimodal attention. *ACM Trans. Graph.*, 40:195:1–195:14, 2021.

[18] Hyemin Ahn, Timothy Ha, Yunho Choi, Hwiyeon Yoo, and Songhwai Oh. Text2action: Generative adversarial synthesis from language to action. *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–5, 2018.

[19] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[20] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, 404:132306, 2020.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 5998–6008, 2017.

[22] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches, 2014.

[24] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

[26] Yanli Ji, Feixiang Xu, Yang Yang, Fumin Shen, Heng Tao Shen, and Wei-Shi Zheng. A large-scale rgb-d database for arbitrary-view human action recognition. In *Proceedings of the 26th ACM International Conference on Multimedia*, MM '18, page 1510–1518, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. In Yoshua Bengio and Yann LeCun, editors, *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, 2014.

[28] Ali Razavi, Aäron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with VQ-VAE-2. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 14837–14847, 2019.

[29] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents, 2022.

[30] Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax, 2017.

[31] Matthew Loper, Naureen Mahmood, Javier Romero, Gerard Pons-Moll, and Michael J. Black. SMPL: A skinned multi-person linear model. *ACM Trans. Graphics (Proc. SIGGRAPH Asia)*, 34(6):248:1–248:16, October 2015.

[32] Yi Zhou, Connelly Barnes, Jingwan Lu, Jimei Yang, and Hao Li. On the continuity of rotation representations in neural networks, 2020.

[33] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks, 2016.

[34] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis R. Bach and David M. Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*, volume 37 of *JMLR Workshop and Conference Proceedings*, pages 448–456. JMLR.org, 2015.

[35] Vinod Nair and Geoffrey E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, page 807–814, Madison, WI, USA, 2010. Omnipress.

[36] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 770–778. IEEE Computer Society, 2016.

[37] Nikhila Ravi, Jeremy Reizenstein, David Novotny, Taylor Gordon, Wan-Yen Lo, Justin Johnson, and Georgia Gkioxari. Accelerating 3d deep learning with pytorch3d. *arXiv:2007.08501*, 2020.

[38] Zongmian Li, Jiri Sedlar, Justin Carpentier, Ivan Laptev, Nicolas Mansard, and Josef Sivic. Estimating 3d motion and forces of person-object interactions from monocular video. In *Computer Vision and Pattern Recognition (CVPR)*, 2019.

[39] Solomon Kullback and Richard A Leibler. On information and sufficiency. *The annals of mathematical statistics*, 22(1):79–86, 1951.

[40] Adrian Lancucki, Jan Chorowski, Guillaume Sanchez, Ricard Marxer, Nanxin Chen, Hans J. G. A. Dolfing, Sameer Khurana, Tanel Alumäe, and Antoine Laurent. Robust training of vector quantized bottleneck models. *CoRR*, abs/2005.08520, 2020.

[41] John S. Bridle. Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition. In Françoise Fogelman Soulié and Jeanny Hérault, editors, *Neurocomputing*, pages 227–236, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[42] Daniel Duckworth, Arvind Neelakantan, Ben Goodrich, Lukasz Kaiser, and Samy Bengio. Parallel scheduled sampling. *CoRR*, abs/1906.04331, 2019.

[43] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[44] Angela Fan, Mike Lewis, and Yann N. Dauphin. Hierarchical neural story generation. In Iryna Gurevych and Yusuke Miyao, editors, *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*, pages 889–898. Association for Computational Linguistics, 2018.

[45] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[46] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 5156–5165. PMLR, 2020.

[47] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[48] Adrian Lam and Alexandre Matton. Making pytorch transformer twice as fast on sequence generation., Dec 2020.

[49] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 2 (Short Papers)*, pages 464–468. Association for Computational Linguistics, 2018.

[50] Ofir Press, Noah A. Smith, and Mike Lewis. Shortformer: Better language modeling using shorter inputs. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli, editors, *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP 2021, (Volume 1: Long Papers), Virtual Event, August 1-6, 2021*, pages 5493–5505. Association for Computational Linguistics, 2021.

[51] Muhammed Kocabas, Nikos Athanasiou, and Michael J. Black. VIBE: video inference for human body pose and shape estimation. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020*, pages 5252–5262. Computer Vision Foundation / IEEE, 2020.

[52] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6629–6640, Red Hook, NY, USA, 2017. Curran Associates Inc.

[53] Martin Popel and Ondř ej Bojar. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70, apr 2018.

[54] Mathis Petrovich, Michael J. Black, and Gül Varol. TEMOS: Generating diverse human motions from textual descriptions. In *European Conference on Computer Vision (ECCV)*, 2022.

[55] Chieh-Hsin Lai, Dongmian Zou, and Gilad Lerman. Robust vector quantized-variational autoencoder. *CoRR*, abs/2202.01987, 2022.

List of Figures

List of Tables