

Conditional Generative Transformers for Hand-Guided Animation Automation

by

Maxwell Clarke

A thesis
submitted to the Victoria University of Wellington
in partial fulfilment of the requirements
for the degree of

Master of Science
in Computer Science.

Victoria University of Wellington

November 2022

Abstract

In this thesis I develop neural networks and apply them to the problem of hand motion modelling for film production.

Firstly, I develop a novel training regime for transformers which allows auto-regressive sampling of high-dimensional data in an arbitrary order, and demonstrate this technique on pixel-by-pixel sampling of MNIST images.

Secondly, I develop a predictive model for hand-motion data, via unsupervised training on a motion-capture dataset. I compare fixed-order sampling to best order I found using the above technique.

Contents

1	Introduction	1
1.1	Motivation	1
1.1.1	Why focus on hand motion?	2
1.1.2	Why focus on transformers?	2
1.2	Previous Work	2
2	Background	3
2.1	Neural Networks and Deep Learning	3
2.1.1	Notation	3
2.1.2	Tasks	5
2.2	Auto-regressive models	6
2.3	Animation	6
2.4	Angle representations	6
3	Understanding Transformers	7
3.0.1	Attention operator	7
3.0.2	Attention is permutation-equivariant	8
3.1	Transformer architecture variants	8
3.1.1	Masked Sequence Modeling: Encoder-only models	8
3.1.2	Causal Masking: Decoder-only models	9
3.1.3	Encoders vs. Decoders	9
3.1.4	Query-only transformers	9

4	Sampling Sequences In Any Order	11
4.1	Transformer Input	11
4.2	Tasks	12
4.2.1	Arbitrary order decoder-only transformer using in- put triples	12
4.2.2	Queries	14
4.3	Hypothesis	14
4.4	Method	15
4.4.1	15
4.4.2	Baseline: Fixed-order sequence prediction	15
4.5	Results	15
5	Angles, Joints and Hands	17
5.1	Parameterizing Hand Configurations	17
5.1.1	Euler Angles	18
5.1.2	Axis-Angle	19
5.2	Loss Functions for learning angles	19
6	Hand Motion Model	21
7	Conclusions	23

Chapter 1

Introduction

This thesis describes the research I have done, which is at the intersection of two areas: Deep Learning and Computer Graphics.

In this thesis:

1. I create a proof of concept machine learning application for a problem in computer graphics. I chose to focus on hand motion prediction.
2. I experiment with Transformer models and understand them in depth.

todo: Summarize what parts are my novel contributions, what parts are my novel summarizations, and what parts are just other peoples' work

1.1 Motivation

The problem domain I focused on – hand motion prediction – is a sequence prediction problem, which are the general kind of task that transformer models are used on. To this end, I hoped to find that these two motivations would feed back into each other as I worked – the application providing direction for the more general / theoretical research, and the insights gained

from the more general research contributing back to better solutions for the problem domain.

1.1.1 Why focus on hand motion?

Whenever a moving virtual character appears in an animated film or a video game, someone had to spend the time to specify the angles of all the joints across all the frames. Fortunately is not necessary to lay out every single frame, because animation tools used for both games and film production make use of interpolation techniques between key-frames, but artists must still specify many joints, over many key-frames, over many different kinds of animation.

For realistic human characters this work can be thought of as divided into three parts, each involving a similar amount of work:

- Facial animation – animating the muscles of the face when a character is talking or otherwise making facial expressions.
- Hand animation – animating the fingers and wrists when a character making gestures or manipulating objects.
- Body animation (also simply called character animation) – animating the rest of the body, e.g. the legs, arms, neck and spine when a character is walking, dancing, etc.

1.1.2 Why focus on transformers?

1.2 Previous Work

Chapter 2

Background

2.1 Neural Networks and Deep Learning

Since Krizhevsky et al.'s AlexNet in 2012, many problems are increasingly being solved by the paradigm of Artificial Neural Networks.

Any particular ANN is not designed but discovered by gradient descent, where the parameters of the ANN are iteratively improved with respect to a loss function, which is typically defined with respect to some dataset.

Although many additional variants exist, neural networks can largely be summarized as the application of linear operators and pointwise non-linear operators on tensors of floating point numbers.

2.1.1 Notation

todo: Differentiate between multi-dimensional data (represented by vectors) are multi-dimensional representations of the data (represented by N-D Tensors. Define batch, sequence, and feature dimensions

Throughout this thesis I will describe aspects various neural networks with mathematical notation. In order to clarify the notation, I provide an

overview here. Firstly, common functions that we see used in neural networks. I use the symbol ϕ for activation functions.

Activation functions such as ϕ_{relu} and ϕ_{gelu} are scalar functions, but are typically applied independently across all components of a tensor. To represent such, I will use the following notation:

$$\begin{aligned} \text{ReLU} \quad & \phi_{\text{relu}}: \mathbb{R} \rightarrow \mathbb{R} \\ & \phi_{\text{relu}}(x)_i \stackrel{\text{def}}{=} \max(0, x_i) \end{aligned}$$

$$\begin{aligned} \text{GeLU} \quad & \phi_{\text{gelu}}: \mathbb{R} \rightarrow \mathbb{R} \\ & \phi_{\text{gelu}}(\mathbf{x})_i \stackrel{\text{def}}{=} \mathbf{x}_i \cdot P(\mathbf{X} \leq \mathbf{x}_i) = \mathbf{x}_i \cdot \frac{1}{2} (1 + \text{erf}(\mathbf{x}_i/\sqrt{2})) \\ \text{where} \quad & \text{erf}(z) = \frac{2}{\sqrt{\pi}} \int_0^z e^{-t^2} dt \end{aligned}$$

In the above equation, the subscript i shows that these functions apply the operation *independently* to each component of the vector \mathbf{x} . Below we see how this works for *softmax* which is a vector valued function:

$$\begin{aligned} \sigma: \mathbb{R}^{A \times B} &\rightarrow \mathbb{R}^{A \times B} \\ \sigma(\mathbf{X})_{ab} &\stackrel{\text{def}}{=} \frac{e^{\mathbf{X}_{ab}}}{\sum_{b'} e^{\mathbf{X}_{ab'}}} \end{aligned} \tag{2.1}$$

The *softmax* function does not apply the operation independently to each component of the vector, but rather the resulting value for each component is dependent on the other components of the vector

Secondly, notation for neural networks. Let $\mathbf{x} \in \mathbb{R}^N$ be some input data embedded into an N -dimensional vector space. Let $W \in \mathbb{R}^{N \times M}$ be a matrix of learned weights, and let $\phi: \mathbb{R} \rightarrow \mathbb{R}$ be some non-linear function. Then,

$$f: \mathbb{R}^N \rightarrow \mathbb{R}^M f(\mathbf{x})_{\text{mlp}} \stackrel{\text{def}}{=} \phi(W\mathbf{x}) + b, \quad b \in \mathbb{R}^M \tag{2.2}$$

represents the computation done by one layer of a simple fully-connected neural network.

A simple classifier network would be defined as follows, for N dimensional data classified into C classes, with L hidden layers:

$$\begin{aligned}
 f_0: \mathbb{R}^N &\rightarrow \mathbb{R}^H \\
 f_0(\mathbf{x}) &= \phi(W_0\mathbf{x}) + \mathbf{b} & W_0 &\in \mathbb{R}^{N \times H} & \mathbf{b}_0 &\in \mathbb{R}^H \\
 f_l: \mathbb{R}^H &\rightarrow \mathbb{R}^H, \quad l \in 1, \dots, L \\
 f_l(\mathbf{x}) &= \phi(W_l f_{l-1}(\mathbf{x})) + \mathbf{b}_l & W_l &\in \mathbb{R}^{H \times H} & \mathbf{b}_l &\in \mathbb{R}^H \\
 f_L: \mathbb{R}^H &\rightarrow \mathbb{R}^C \\
 f_L(\mathbf{x}) &= \sigma(W_L f_{L-1}(\mathbf{x})) & W_L &\in \mathbb{R}^{H \times C} \\
 f: \mathbb{R}^N &\rightarrow \mathbb{R}^C \\
 f &= f_L \circ f_{L-1} \circ \dots \circ f_0 & \theta &= \{W_0, \dots, W_L, \mathbf{b}_0, \dots, \mathbf{b}_{L-1}\}
 \end{aligned}$$

2.1.2 Tasks

When designing a neural network we can make different choices about the architecture, loss function etc. so that we can use the resulting model on specific tasks:

- Training objective:
 - Regression (*implicit* MLE)
 - Parameter estimation (*explicit* MLE)
 - Classification (Parameter estimation on categorical distributions)
- Architecture
 - Fixed input size models (CNNs, ResNets, MLPs)
 - Sequence models (RNNs, Transformers)
- Training data design

todo: Use the correct terminology for "Regression (*implicit* MLE)" and "Parameter regression (*explicit* MLE)"

The choice of training objective affects the settings in which a model can be used, which theoretical properties we get from it, and more.

The simplest kind of training objective is regression. When we train a model with a regression objective it learns to predict the expected value of the output. Regression is characterized by using an error function as the loss, for example *mean-squared-error*:

$$L_{\text{MSE}}: \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \rightarrow \mathbb{R}$$

$$L_{\text{MSE}}(y, \hat{y})_{ni} \stackrel{\text{def}}{=} \frac{1}{N} \sum_n \left[\sum_i (y_{ni} - \hat{y}_{ni})^2 \right] \quad (2.3)$$

This function sums the error over the *feature* dimension D and averages the error over the *batch* dimension N ¹. Training a model by minimising this loss function, is equivalent to maximising the likelihood of a Gaussian distribution. Given input x , the model output $y = f(x)$ can be interpreted as $\mathbb{E}[p(y|x)]$.

2.2 Auto-regressive models

todo: Formulation of auto-regressive models and sampling

todo: Things we can do with an auto-regressive model

2.3 Animation

2.4 Angle representations

todo: Quaternions and unitary / complex matrices

¹Averaging has no effect on the optimization, it is simply that dividing by the batch and/or sequence length means that the loss value remains in the same range independent of the batch size or sequence length.

Chapter 3

Understanding Transformers

Many of the recent amazing results in deep learning have been achieved with transformer models. In this chapter I seek to understand these models at a deeper level,

todo: Point forward to where the various transformer concepts are used. eg. 4.2

3.0.1 Attention operator

An attention operation is of the following form, using short summation notation, where σ is the *softmax* operator, and A is the pre-softmax attention logits.

$$f_{\text{attn}}: \mathbb{R}^{M \times D} \times \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times V} \rightarrow \mathbb{R}^{M \times V}$$
$$f_{\text{attn}}(\mathbf{Q}, \mathbf{K}, \mathbf{V})_{mv} \stackrel{\text{def}}{=} \sum_n \left[\sigma \left(\sum_d \mathbf{Q}_{md} \mathbf{K}_{nd} \right)_{mn} \mathbf{V}_{nv} \right] \quad (3.1)$$

$$M, N, D, V \in \mathbb{N}$$
$$\mathbf{Q} \in \mathbb{R}^{M \times D}, \mathbf{K} \in \mathbb{R}^{N \times D}, \mathbf{V} \in \mathbb{R}^{N \times V}$$

Since the introduction of transformers it is common to use *multi-head*

attention, which allows for multiple *heads* which each perform an attention operation in parallel with smaller key dimensionality $D_{\text{head}} = \frac{D}{n_{\text{heads}}}$.

When we use Q and K derived from the same sequence of feature/embedding vectors, then $M = N$ and it is called *self-attention*.

When Q and K are derived from separate sequences of feature/embedding vectors, then $M \neq N$ and this is called *cross-attention*.

A comparison

3.0.2 Attention is permutation-equivariant

todo: Make sure permutation-equivariance is the right terminology

todo: Explain and justify permutation-invariance of standard transformer layers

3.1 Transformer architecture variants

The defining feature of a transformer model is that it has “attention” layers. However, there is not just one way to assemble these layers, and there is not just one way to train these models. Here I discuss a few of the notable variants and the motivation behind them.

3.1.1 Masked Sequence Modeling: Encoder-only models

Arguably the simplest attention-based model architecture are encoder-only models. These are models trained on a sequence reconstruction task. Examples are the BERT [bert] language model family, Wav2Vec [wav2vec] for speech, and SimMIM [1] image model.

These models are typically used for sequence understanding tasks and classification tasks. The limitation of this kinds of models is it is trained in a regression (MAP) setting with respect to sequences of data, or

todo: Figure of encoder-only model / masked sequence modeling.

3.1.2 Causal Masking: Decoder-only models

todo: Figure of decoder-only model / next-token prediction.

The simplest and most common transformer architecture is the decoder-only architecture, which I show below in The distinguishing feature is that the only attention layers are self-attention layers, which are trained using a causal mask.

Some examples of where we see this architecture in use are:

- OpenAI's GPT-series [**gpt1**, **gpt2**, **gpt3**] language models.
- Latent Sequence Predictors in VQ-GAN [**vqgan**] and Google's Parti [**parti**]

3.1.3 Encoders vs. Decoders

3.1.4 Query-only transformers

Chapter 4

Sampling Sequences In Any Order

When we predict sequences we usually predict them in time-order. For data with a temporal dimension, this is usually fine, because the real process that generated the data had a causal structure in the temporal dimension. However, when it comes to data with spatial dimensions, the best ordering may not be obvious.

In this chapter I will investigate learning auto-regressive sequence transformers that are not restricted to sampling in a single order over the data.

In particular, I investigate the following question:

- Is it better to sample in order of lowest-entropy-first or highest-entropy-first?

4.1 Transformer Input

Transformers have their input provided as (position, content) pairs, or more generally, as some kind of token embedding which is unique among the input set, such as special “BEGIN” or “CLASS” tokens.

As we saw in chapter 3 both the attention layers and feed-forward layers are invariant to permutations of the input sequence. The model need not be retrained. Additionally, there is no requirement that the input set be contiguous in the sequence dimension - there can be (potentially large) gaps, with no change to the structure of the model (however, the model must be trained for the particular problem still).

As a result of being invariant to permutations, and working with non-contiguous sequences, we can present many different kinds of sequence prediction tasks to a transformer that we could not easily present to other models.

- A Recurrent Neural Network (RNN) can be given sequences with gaps, but is not invariant to the order of the "previous token" conditioning data, which must be incorporated first in some particular order.
- A convolutional or dense neural network applied to the input including the sequence dims (ie. not "pointwise") is neither invariant to the order, nor can be applied to sequences with gaps.

In the next section I describe some tasks we might want to perform with these unique abilities of a transformer.

4.2 Tasks

In this section I discuss various tasks that utilize the ability of a transformer to predict sequences in any order.

4.2.1 Arbitrary order decoder-only transformer using input triples

Let us examine first decoder-only transformers. these predict the next input from the previous input, conditioned on the rest of the sequence via

their attention layers.

An input examples in the training data for these models is typically a set as follows:

$$\{\dots, (i_{n-1}, x_{n-1}, i_n), (i_n, x_n, i_{n+1}), (i_{n+1}, x_{n+1}, i_{n+2}), \dots\}$$

Where i represents the position of a token, and x represents the value of a token.

However, if we instead construct an input sequence in the following way, we can train the model such that given any conditioning sequence (previous tokens), we can ask the model to predict a specific next token.

We do this by providing the input as (input position, input value, target position] triples instead of [input position, input value] pairs (in which the target is implicit)

todo: Make and include a figure for this. Should show difference between input as triples [input position, input value, target position] and pairs [input position, input value]

todo: I haven't found any works that do this - but I still think there probably are some out there

An example input in the training data is a set as follows:

$$\{\dots, (i_{n-1}, x_{n-1}, i_n), (i_n, x_n, i_{n+1}), (i_{n+1}, x_{n+1}, i_{n+2}), \dots\}$$

todo: Improve the above formatting

If our sequence is presented in contiguous-forward-only ordering, i_n is always paired with i_{n+1} and we do not introduce any new information. However, we can create a sequence with an arbitrary order during training, so the model learns to utilize this information.

Then, at inference time we can choose any position i_{n+1} the model should predict next, by constructing the following triple (i_n, x_n, i_{n+1}) and appending it to the rest of the previous tokens.

4.2.2 Queries

todo: Write the introduction to this architecture variant in Chapter 3

If we now examine the case of pure-query-decoder transformers, which I introduced in ??, These are encoder-decoder transformers and are trained with a different task:

$$\begin{aligned}\text{input} &= (\{\dots, (i_{n-1}, x_{n-1}), (i_n, x_n)\}, i_{n+1}) \\ \text{output} &= (x_{n+1})\end{aligned}$$

todo: Improve the above formatting

The previous tokens are provided as a set of pairs to the encoder. Importantly, the decoder is run independently for every input/output pair.

4.3 Hypothesis

Using the above two methods “triples” and “pure-query-decoder” models, I investigate a hypothesis about selecting a better sampling order on a toy dataset.

The hypothesis is as follows.

Assume that using the above methods, we can choose a dynamic ordering in which to sample a sequence at inference time. In particular, one of the ways we can do this is by evaluating the *entropy* of all candidate positions, then sampling from the one with either the lowest or the highest entropy.

I hypothesize that when auto-regressively sampling pixels to produce MNIST images, using a “lowest-entropy-first” ordering, will produce visually better results than a “highest-entropy-first” ordering.

4.4 Method

todo: Method: Details of experiments.

4.4.1

4.4.2 Baseline: Fixed-order sequence prediction

todo: Subsection on training forward-only task

todo: Subsection on Training with arbitrary-order methods, but in forward-only mode (sanity check - should have similar, perhaps somewhat worse results)

todo: Subsection on Ablation / Hyper parameter tuning

todo: Subsection on Training (input position, input, target position)

todo: Subsection on Training query-decoder

4.5 Results


todo: Section MNIST Results / Comparison

Chapter 5

Angles, Joints and Hands

To a first approximation, the human hand has 23 degrees of freedom.

Each hand has 16 joints - three per digit, plus the wrist. The wrist and the first joint on each digit (the metacarpophalangeal (MCP) joints) can rotate on two axes, and so each have 2 degrees of freedom. The rest (the proximal- and distal-interphalangeal (PIP & DIP) joints) can only rotate on one axis, and have 1. This naive counting gives 22 degrees of freedom. In addition, we usually consider rotation of the forearm (about the longitudinal axis) to be part of the hand, modeling it as a third degree of freedom of the wrist, which brings the total to 23 degrees of freedom.

Considering that different combinations of joint angles can have very different meanings , animators have a lot of work to do when bringing a digital character to life. Small mistakes can cause a character to look unnatural, but in many scenes getting the hands perfect goes unnoticed.

5.1 Parameterizing Hand Configurations

It is not so straightforward to assign an angle to each of those 23 degrees of freedom. There are a variety of different parameterizations of the joints we might choose from, and additionally we have the option of placing constraints on the range of values that the angles can take. In this section we

will discuss some of the most common parameterizations, and the pros and cons of each.

5.1.1 Euler Angles

Euler angles are the most common parameterization for human joints. They are also the simplest to understand. Each joint is assigned three angles, one for each axis of rotation. The axes are usually chosen to be the x , y , and z axes of the coordinate system, but they can be chosen to be any three axes that are orthogonal to each other. For example, the axes could be chosen to be the axes of the joint itself, or the axes relative to the parent joint. The order in which the rotations are applied is also important. The most common order is to apply the rotations in the order z, y, x , but they can be applied in any order.

This parameterization has the topology $S^1 \times S^1 \times S^1 \cong T^3$, where S^1 is the circle of angles. However, the space of rotations $SO(3)$ itself has topology S^3 , so the Euler angle parameterization cannot be perfect. In particular, there must be singularities.

The principle advantage of the Euler angle parameterization is that it is easy to understand and implement. The principle disadvantages are due to the singularities:

1. there may be multiple sets of Euler angles that correspond to the same rotation
2. as a result, we cannot easily interpolate between two configurations that are close together but have different Euler angles

For example, if we rotate a joint by 90° about the x axis, and then by 90° about the y axis, we will get the same rotation as if we had rotated by 90° about the y axis, and then by 90° about the x axis.

todo: Euler angle figure

If we tried to interpolate between these two configurations, we would get a rotation that is not the same as either of the two configurations. This occurs because the Euler angle parameterization is not a smooth function.

5.1.2 Axis-Angle

An alternative parameterization is to use an axis-angle representation. In this representation, each joint is assigned a unit vector and an angle. The unit vector specifies the axis of rotation, and the angle specifies the amount of rotation. The axis-angle representation has the topology $S^2 \times S^1$, where S^2 is the sphere of unit vectors. Because topology still does not match the space of rotations $SO(3)$, there are still singularities. However

5.2 Loss Functions for learning angles

In Chapter 2 I introduced some loss functions such as the Mean Squared Error (MSE), which predicts the posterior expectation $\mathbb{E}(y|x)$. In this section I will discuss some other loss functions, that are commonly used for learning data which sits on .

A variant of this is the *angular* mean-squared-error, which I will use later on in Chapter 6.

$$L_{\theta\text{-MSE}}: \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \rightarrow \mathbb{R}$$

$$L_{\theta\text{-MSE}}(y, \hat{y})_{ni} \stackrel{\text{def}}{=} \frac{1}{N} \sum_n \left[\sum_i (\sin y_{ni} - \sin \hat{y}_{ni})^2 + (\cos y_{ni} - \cos \hat{y}_{ni})^2 \right] \quad (5.1)$$

Here y and \hat{y} are vectors of angles, and the error is defined in terms of the squared arc length between their corresponding components. This loss function is equivalent to maximising the likelihood of a von Mises distribution, the derivation for which can be found in Chapter 5. This is useful

when we want to model angles, for example when we want to predict the orientation of a hand, which we will do in Chapter 6.

todo: Derivation of θ -MSE minimizing von-mises distribution

Chapter 6

Hand Motion Model

I experimented with a variety of different model architectures,

Loss Value mean across whole dataset	0.07275154
--------------------------------------	------------

Chapter 7

Conclusions

Bibliography

- [1] Zhenda Xie et al. “SimMIM: A Simple Framework for Masked Image Modeling”. In: *CoRR* abs/2111.09886 (2021). arXiv: [2111.09886](https://arxiv.org/abs/2111.09886). URL: <https://arxiv.org/abs/2111.09886>.