# 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

# Chapter 1

# Introduction

In this thesis I discuss the learnings and experimentds from the past year, which are at the intersection of two areas: Deep Learning and Computer Graphics. There are three main areas of focus:

1. I experiment with *transformer* models, and understand them in depth (Chapter 3).

2. I experiment with using Transformer models for Bayesian inference, essentially as Gaussian Processes (Chapter 4).

3. I create a proof of concept machine learning application for *hand motion prediction*, useful for film and video game production (Chapters 5 and 6).

Although much of my work is summarizing others' research and presenting my learnings, I have two main novel contributions:

1. I present experiments with *dynamically-ordered* auto-regressive sampling, which utilises the *permutation-invariance* of the attention oper-

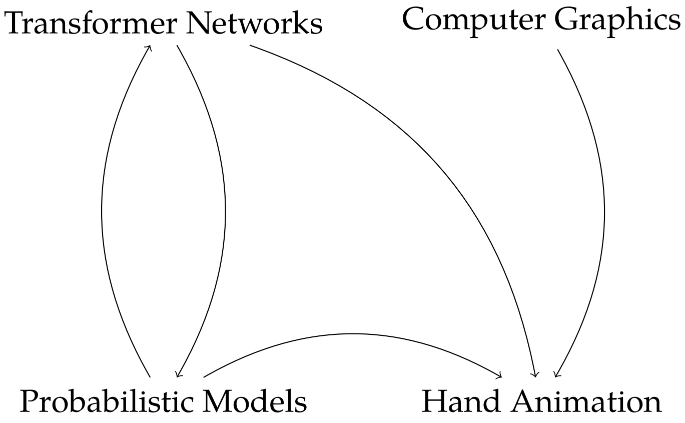

**Figure 1.1:** Which learnings and experiments contributed to each other in this thesis.

ation in transformer models.

2. I present a proof-of-concept transformer-based generative model for hand motion prediction, which can be used to predict hand motion at arbitrary target frames, and to predict the joints of a hand in any order within that frame.

In the rest of this chapter, I will discuss the motivation for this thesis, and previous work that relates to it.

## 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. 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.

In order to fully animate a character, an animator needs to appropriately animate all moving parts of that character. We can divide these into three groups, each involving a similar amount of work:

- Facial animation – animating the muscles of the face when a character talks or otherwise makes facial expressions.

- Hand animation – animating the fingers and wrists when a character makes gestures or manipulates 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 walks, dances, etc.

Of these, the face and body are often the most noticeable, and most of the time recieve the most attention from the audience. However, the hands are also very important, and are often the most difficult to animate.

todo: Examples of problematic hand animation.

### 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 [**alexnet**] in 2012, many problems are increasingly being solved best by Artificial Neural Networks (NN). Any particular NN is not designed but discovered by gradient descent, where the parameters of the NN are iteratively improved with respect to a loss function, and a dataset.

Many variants exist, and in this section I firstly introduce and clarify my notation, and then give an overview of the different aspects of neural networks to contextualize my work.

### 2.1.1 Notation

Since we are often using tensors of rank 3 or higher in neural network implementations, it is useful to have some notation that clarifies how functions are applied across these dimensions.

First, a simple example using activation functions. The ReLU activation function is defined as:

$$\begin{aligned} \phi_{\text{relu}} &: \mathbb{R} \to \mathbb{R} \\ \phi_{\text{relu}}(x) &\overset{\text{def}}{=} \max(0, x) \end{aligned} \tag{2.1}$$

It is customary to use the symbol $\phi$ for activation functions. Activation functions are typicall scalar functions, but are applied independently across all components of a tensor. To represent such, I will use the following notation, for example, applying the ReLU activation function to a vector $\boldsymbol{x}$:

$$\boldsymbol{x}'_i = \phi_{\text{relu}}(\boldsymbol{x}_i)$$

In the above equation, the subscript $i$ shows that the function is applied *independently* to each component of the vector $\boldsymbol{x}$.

This notation also works for multiple dimensions, including when an operation is not applied independently across some dimension. For example, the following is how I will show the *softmax* function, which is a vector valued function, applied independently across the rows of a $B \times N$ matrix $\boldsymbol{X}$ (which is used in the definition of the attention operation in Chapter 3).

The softmax function is defined as:

$$\begin{aligned} \sigma &: \mathbb{R}^N \to \mathbb{R}^N \\ \sigma(\boldsymbol{x}_n) &\overset{\text{def}}{=} \frac{e^{\boldsymbol{x}_n}}{\sum_{n'} e^{\boldsymbol{x}_{n'}}} \end{aligned} \tag{2.2}$$

We can apply the the above function to a matrix $\boldsymbol{X} \in \mathbb{R}^{B \times N}$ independently over $B$ as follows:

$$\begin{aligned} \boldsymbol{X}'_{b,n} &= \sigma(\boldsymbol{X}_b)_n \\ &= \frac{e^{\boldsymbol{X}_{b,n}}}{\sum_{n'} e^{\boldsymbol{X}_{b,n'}}} \end{aligned}$$

The order of the subscript indices $b$ and $n$ is ignored – the indices index their respectively-named dimensions. However, I will generally use the convention that the first subscript is the batch index, the second subscript is the sequence index, and the third subscript is the feature index.

I will now define some simple neural networks as examples for clarifying any later notation.

I will typically use $N$ for the input dimensionality of a network, and $D$ for the *embedding* (or *hidden* / *latent*) dimensionality. Let $\boldsymbol{x} \in \mathbb{R}^N$ be some input data embedded into an $N$-dimensional vector space. Let $W \in \mathbb{R}^{N \times D}$ be a matrix of learned weights, and let $\phi\colon \mathbb{R} \to \mathbb{R}$ be some non-linear function. Then, the computation done by one layer of a simple fully-connected neural network is represented as follows.

$$\begin{aligned} &f_{\text{mlp}}\colon \mathbb{R}^N \to \mathbb{R}^D \\ &f_{\text{mlp}}(\boldsymbol{x}) \overset{\text{def}}{=} \phi(W\boldsymbol{x}) + b \end{aligned} \tag{2.3}$$

$$W \in \mathbb{R}^{N \times D}, \quad \boldsymbol{b} \in \mathbb{R}^D$$

$W$ is the weight matrix, and $\boldsymbol{b}$ is the bias vector, which together are the parameter set for this simple model. The output of the neural network is a $D$-dimensional vector.

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

$$
\begin{aligned}
& f_0 \colon \mathbb{R}^N \to \mathbb{R}^D \\
& f_0(\boldsymbol{x}) = \phi(W_0 \boldsymbol{x}) + \boldsymbol{b} && W_0 \in \mathbb{R}^{N \times D} && \boldsymbol{b}_0 \in \mathbb{R}^D \\
\\
& f_\ell \colon \mathbb{R}^D \to \mathbb{R}^D && \forall \ell \in 1, \dots, L \\
& f_\ell(\boldsymbol{x}) = \phi(W_\ell f_{\ell-1}(\boldsymbol{x})) + \boldsymbol{b}_\ell && W_\ell \in \mathbb{R}^{D \times D} && \boldsymbol{b}_\ell \in \mathbb{R}^D \\
\\
& f_L \colon \mathbb{R}^D \to \mathbb{R}^C \\
& f_L(\boldsymbol{x}) = \sigma(W_L f_{L-1}(\boldsymbol{x}) + \boldsymbol{b}_L) && W_L \in \mathbb{R}^{D \times C} && \boldsymbol{b}_L \in \mathbb{R}^C
\end{aligned}
\tag{2.4}
$$

$$
\begin{aligned}
& f_{\text{classifier}} \colon \mathbb{R}^N \to \mathbb{R}^C \\
& f_{\text{classifier}} = f_L \circ f_{L-1} \circ \cdots \circ f_0 && \theta = \{W_0, \cdots, W_L, \boldsymbol{b}_0, \cdots, \boldsymbol{b}_L\}
\end{aligned}
$$

The parameters of the network are $\theta = \{$W0, ..., WL, vb0, ..., vbL-1$\}$. The output of the network is a $C$-dimensional vector, where each component is the probability that the input belongs to that class. This model would be trained with a categorical cross-entropy loss function – which I will discuss in the next section.

### 2.1.2 Tasks

Neural networks are applied to a wide variety of tasks, which lead to a number of different choices for the architecture and loss function. In this section, I will help to contextualize my later work by giving a brief overview of the ways that different neural networks and training setups differ.

On the following pages in Figures 2.1, 2.2 and 2.3, we can see some simple ontologies of the different considerations that combine to define a particular task and architecture, in particular:

1. Training objective / loss function
2. Data dimensionality / length
3. Dataset type

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*:

$$\begin{aligned} L_{\text{MSE}} &: \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ L_{\text{MSE}}(y, \hat{y})_{ni} &\overset{\text{def}}{=} \frac{1}{N} \sum_{n} \left[ \sum_{i} (y_{ni} - \hat{y}_{ni})^2 \right] \end{aligned} \tag{2.5}$$

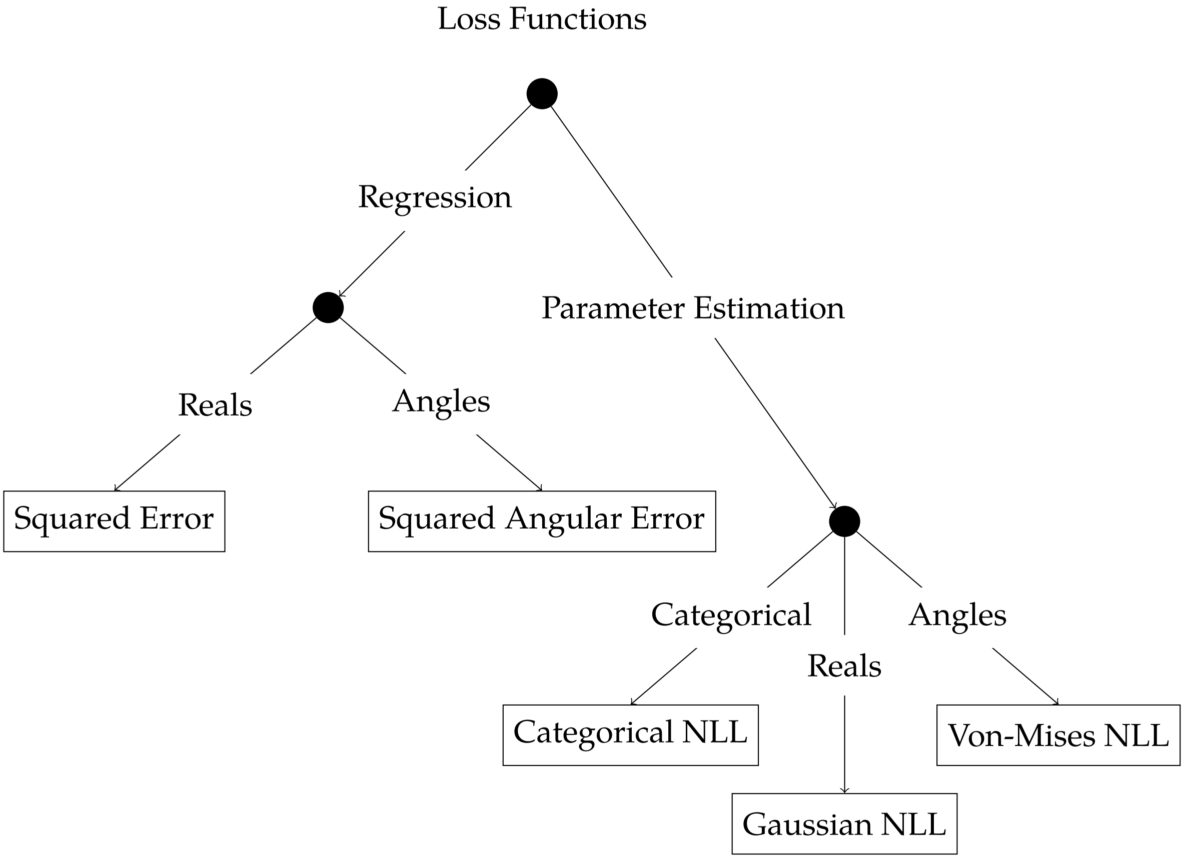

**Figure 2.1:** We can split loss functions into two categories.

In the former, the loss function has the form of an error function. When minimizing this function, the model learns to output the expected value of the posterior $\mathsf{E}\left[p(y \mid x)\right]$ of the output $y$ given the input $x$. This is called a regression or maximum-a-posteriori (MAP).

In the latter, the loss function has the form of a negative-log-likelihood (NLL) function. The model outputs the parameters of a probability distribution, and the loss function is the negative log-likelihood of the data under that distribution. This includes the case of categorical NLL (also called categorical cross-entropy), where the model outputs a probability distribution over a discrete set of classes.

Models trained with a NLL loss learn to output an explicit posterior distribution $p(y \mid x)$, given a fixed functional form for $p$, such as a Gaussian, mixture of Gaussians, Categorical, Von-Mises, etc. Depending on the task, and output format, different functional forms for $p$ may be appropriate.

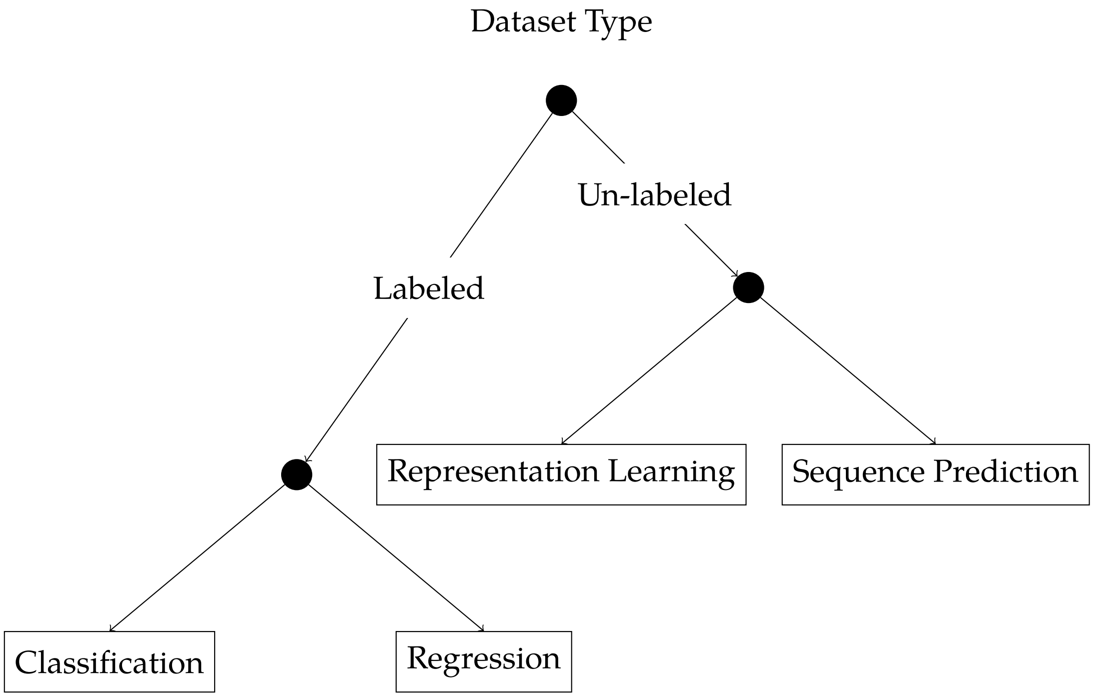

**Figure 2.2:** Basic ontology of dataset types.

When learning on unlabeled data, the goal is to learn a representation of the data that is useful for some downstream task.

When learning on data that is explicitly labeled – the goal is to learn a model that performs well on the task directly.

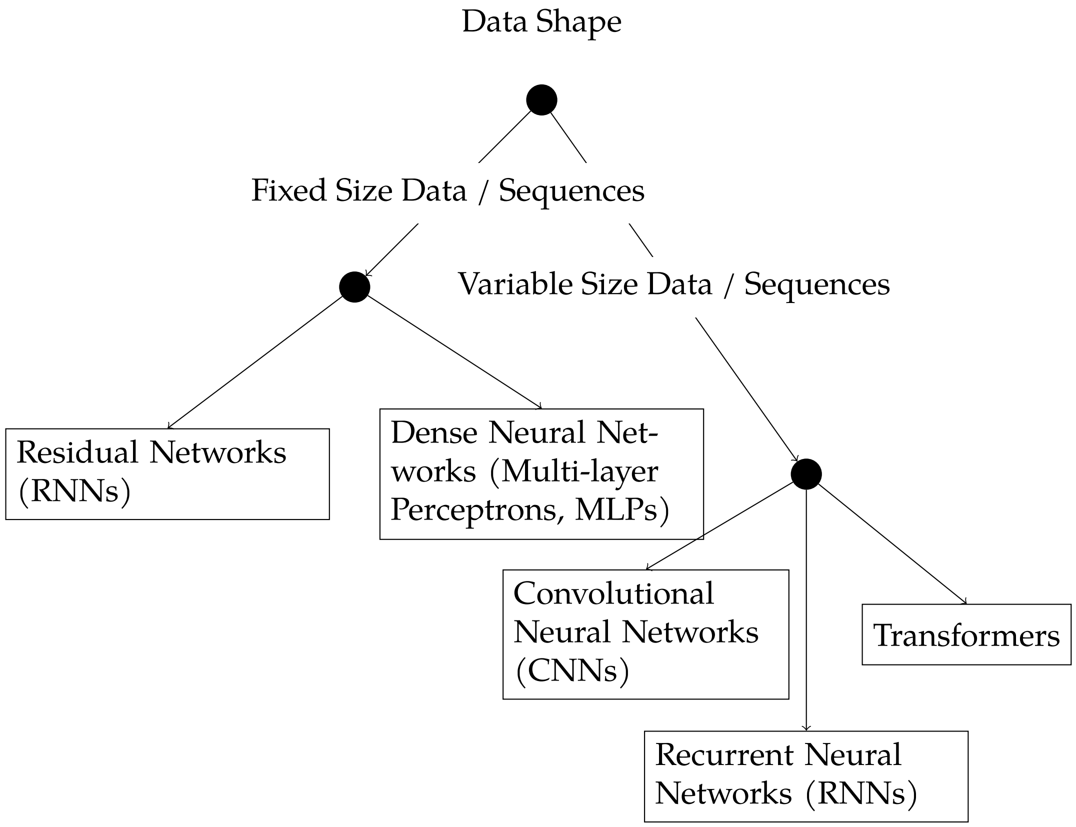
Data Shape
Fixed Size Data / Sequences
Variable Size Data / Sequences
Residual Networks (RNNs)
Dense Neural Networks (Multi-layer Perceptrons, MLPs)
Convolutional Neural Networks (CNNs)
Transformers
Recurrent Neural Networks (RNNs)

**Figure 2.3:** Neural network variants which support variable input shape.

Due to their construction, MLPs and ResNets are restricted to a fixed input shape, and so can only be trained and used on data that is of a fixed size, such as tabular data, or data that has been processed into a fixed size by re-sampling, chunking etc.

RNNs, CNNs and Transformers can accept variable length data, each with their own tradeoffs. They are typically more suitable for data that is naturally of variable size/length, such as text, audio or images.

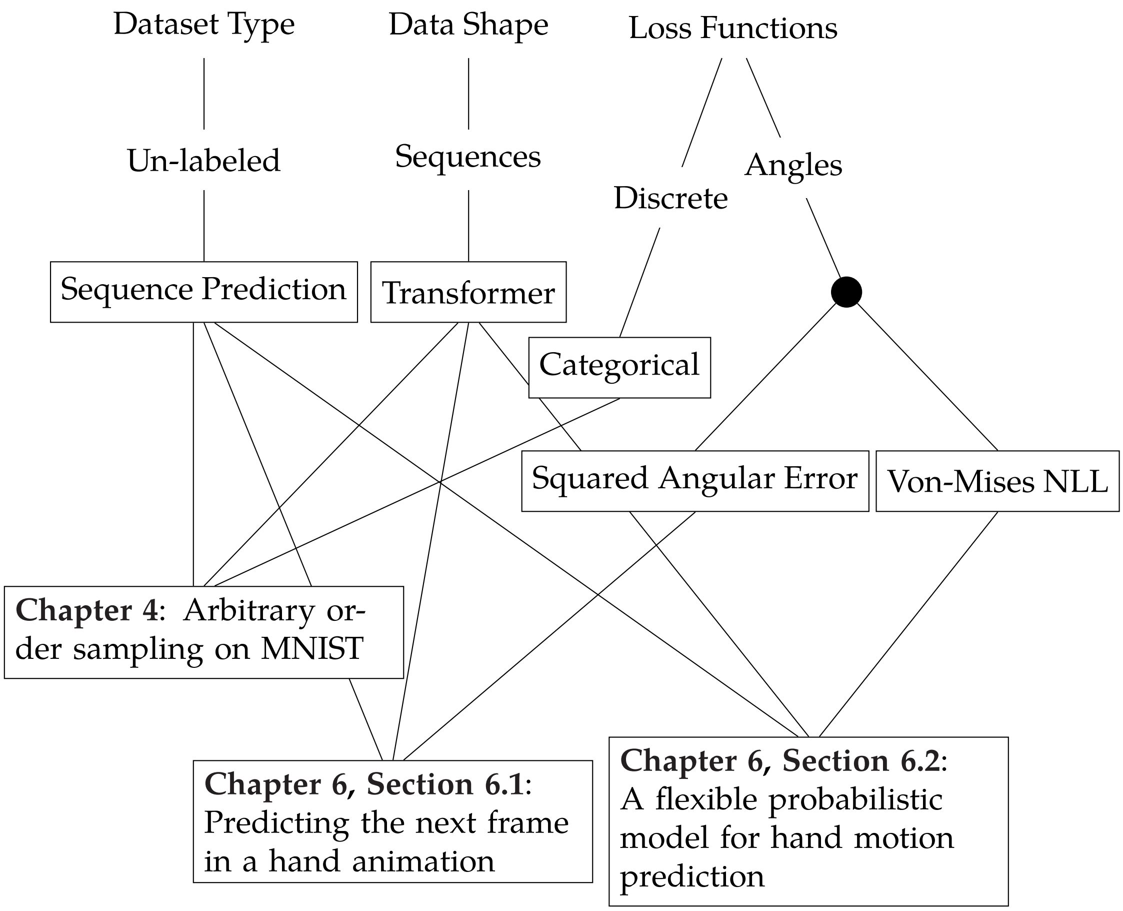
Dataset Type
Data Shape
Loss Functions
Un-labeled
Sequences
Discrete
Angles
Sequence Prediction
Transformer
Categorical
Squared Angular Error
Von-Mises NLL
**Chapter 4**: Arbitrary order sampling on MNIST
**Chapter 6, Section 6.1**: Predicting the next frame in a hand animation
**Chapter 6, Section 6.2**: A flexible probabilistic model for hand motion prediction

**Figure 2.4:** The later work in this thesis sits focuses on learning un-labeled sequence data with transformers, in two different domains.

In Chapter 4, I train a transformer-based probabilistic model which can be used as a gaussian process for predicting pixels on the MNIST dataset.

In Chapter 6, I train a transformer-based model on the ManipNet hand motion dataset. Section 6.1 focuses on a deterministic model, while Section 6.2 focuses on a probabilistic model.

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 $\mathsf{E}[p(y|x)]$.

## 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

[1] Averaging 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 a class of neural networks called *transformers*, which were introduced and named in [1]. In this chapter I seek to understand this class of models at a broader level. I will discuss:

- The unique properties of the transformer architecture, which include working with sequences of any length without changing the weights, being able to be computed in parallel across the sequence during training, and being invariant to the order of the inputs.
- The different variants of transformers, namely encoder-only, decoder-only, and encoder-decoder models.
- The variety of different tasks that this architecture is suited for, which include an extremely generic gaussian-process-like category of tasks.

Firstly, I will discuss the defining feature of a transformer which is including at least one *attention* operation.

## 3.1 The Attention Operation

Attention is a biologically-inspired mechanism that allows a model to recieve inputs from distant parts of the input data, as weighted by the *attention* given to those parts, which is computed from the data itself. This has proven extremely useful for diverse tasks including machine translation, image generation, and more. In this section I will describe the attention operator.

Attention has a number of useful properties which come from its mathematical construction.

### 3.1.1 Mathematical Definition

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

$$f_{\text{attn}} : \mathbb{R}^{M\times D} \times \mathbb{R}^{N\times D} \times \mathbb{R}^{N\times V} \to \mathbb{R}^{M\times V}$$

$$f_{\text{attn}}(Q, K, V)_{mv} \overset{\text{def}}{=} \sum_n \left[ \sigma\Big( \sum_d Q_{md} K_{nd} \Big)_{mn} V_{nv} \right] \tag{3.1}$$

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

$$M, N, D, V \in \mathbb{N}$$

The innermost multiplication of $Q$ and $K$ is simply the inner product (dot product) between vectors $Q_m$ and $K_n$. This however is not inherent. Instead of the inner product, we can substitute any kernel function. (Although this is not usually done because the inner product is the most natural choice, and is efficient to compute)

For clarity, the expanded form of this multiplication, resulting in the unnormalized attention weights **A**, for an arbitrary kernel function $k$, is as follows:

$$\mathbf{A} = QK^T = \begin{bmatrix} Q_{1,1} & Q_{1,2} & \cdots & Q_D \\ Q_{2,1} & Q_{2,2} & \cdots & Q_D \\ \vdots & \vdots & \ddots & \vdots \\ Q_{M,1} & Q_{M,2} & \cdots & Q_D \end{bmatrix} \begin{bmatrix} K_{1,1} & K_{1,2} & \cdots & K_D \\ K_{2,1} & K_{2,2} & \cdots & K_D \\ \vdots & \vdots & \ddots & \vdots \\ K_{N,1} & K_{N,2} & \cdots & K_D \end{bmatrix}^T$$

$$= \begin{bmatrix} k(Q_1, K_1) & k(Q_1, K_2) & \cdots & k(Q_1, K_N) \\ k(Q_2, K_1) & k(Q_2, K_2) & \cdots & k(Q_2, K_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(Q_M, K_1) & k(Q_M, K_2) & \cdots & k(Q_M, K_N) \end{bmatrix}$$

where typically $k = \cdot$ (the dot-product operator).

### 3.1.2 Attention operates on sets

When processing inputs, an interesting property to have that is useful for some tasks is the ability to operate on sets of inputs. For example, in the case of graphs, or sets of heterogeneous values, there may not be a natural ordering in which to process the inputs. In this case, the attention operator can be used to operate on the entire set of inputs at once. (However, when *sampling* outputs, we typically still need to decide on some order. I will discuss this in more detail in Chapter 4).

Due to the construction of the attention operator, we can see that the output $O_i$ corresponding to a query vector $Q_i$ is independent of the order of the key and value vectors $K_i$ and $V_i$. This is because the computation only involves

- pairwise-dot products between $Q_i$ and $K_j$
- the softmax operation, which is *equivariant* to the order of its inputs
- addition of the corresponding $V_j$ vectors, which is *commutative*

The computation of a query vector $Q$ is *invariant* to permutations of the key and value vectors. This is because the softmax operator is *invariant* to permutations of its inputs.

The computations of the query vectors are also independent of each other, which makes the attention operator also *equivariant* to permutations of the query vectors.

This stands in contrast to the convolution operator, which is equivariant to the order of its inputs, but not invariant to permutations of its inputs.

It also stands in contrast to recurrent neural networks, which are neither equivariant nor invariant to permutations of their inputs.

This is a very important property of attention, and I claim it is as yet underappreciated.

todo: Make sure permutation-equivariance is the right terminology

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

### 3.1.3 Attention variants

Attention is computed from the three matrices (or sequences of vectors) $Q$, $K$ and $V$. In a neural network, these are each typically derived in some fashion from the inputs to the network. The most common way to do this is to use a learned linear transformation, which is simply a matrix multiplication followed by a bias term, for example

$$Q = W_Q\boldsymbol{X} + \boldsymbol{b}_Q \tag{3.2}$$

$$K = W_K\boldsymbol{X} + \boldsymbol{b}_K \tag{3.3}$$

$$V = W_V\boldsymbol{X} + \boldsymbol{b}_V \tag{3.4}$$

If we derive all three matrices from the same input $\boldsymbol{X}$, then $M = N$ and the attention operation is called a *self-attention* operation. A diagram of this is shown in Figure 3.1. The blue shaded areas show the receptive field used when computing each output vector $x'_i$.

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

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 $D_{\text{head}} = \frac{D}{n_{\text{heads}}}$.

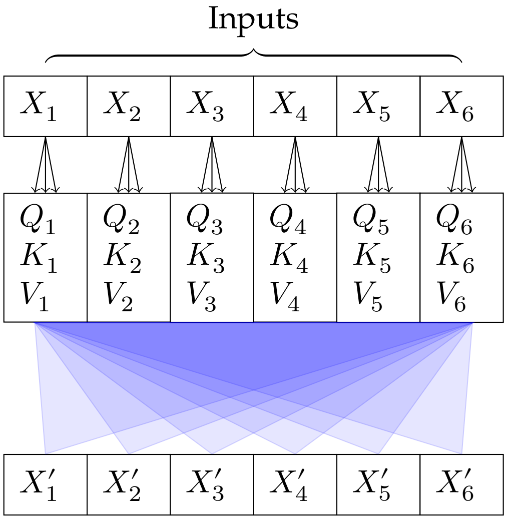

**Figure 3.1:** Full self-attention, as used in transformer encoders.

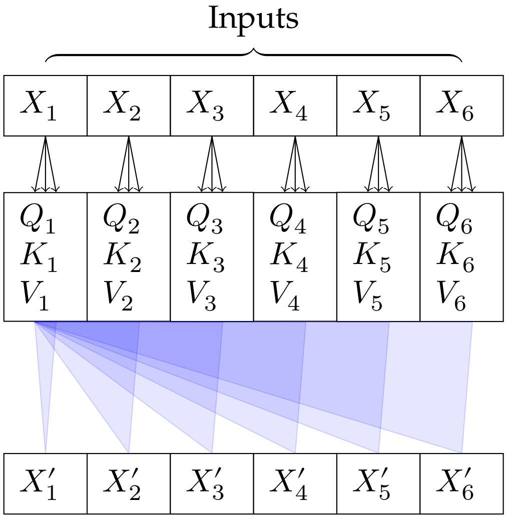

**Figure 3.2:** Self-attention with causal masking, as used in transformer decoders during training.

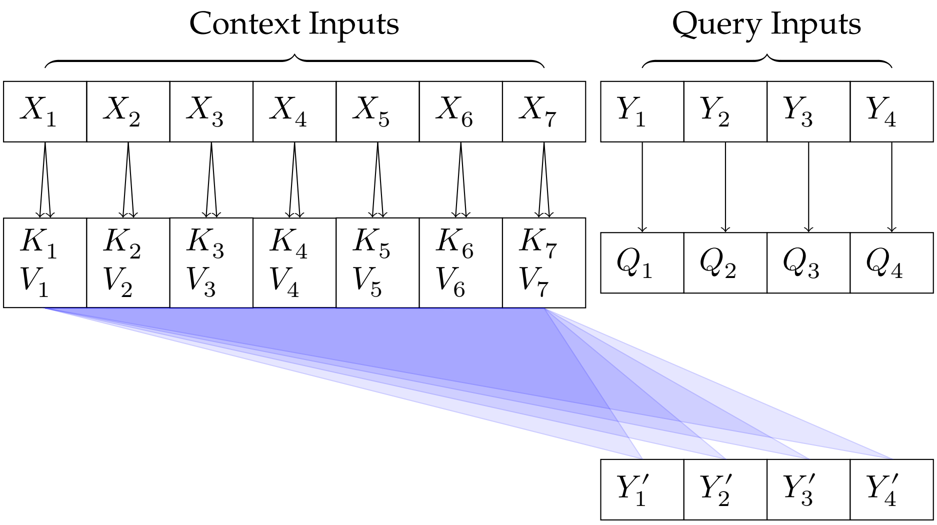
Context Inputs
Query Inputs
$X_1$ $X_2$ $X_3$ $X_4$ $X_5$ $X_6$ $X_7$
$Y_1$ $Y_2$ $Y_3$ $Y_4$
$K_1$ $V_1$ $K_2$ $V_2$ $K_3$ $V_3$ $K_4$ $V_4$ $K_5$ $V_5$ $K_6$ $V_6$ $K_7$ $V_7$
$Q_1$ $Q_2$ $Q_3$ $Q_4$
$Y'_1$ $Y'_2$ $Y'_3$ $Y'_4$

**Figure 3.3:** Cross-attention, as used during during transformer inference, and during training for some variants.

## 3.2 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.

### 3.2.1 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 [3] image model.

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

### 3.2.2 Causal Masking: Decoder-only models

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

Currently the most common transformer architecture is the decoder-only architecture, a diagram of which I show below in

todo: Figure

. These models are used for sequence prediction/generation, and trained via self-supervised learning. The distinguishing feature of a decoder-only model is that its only attention layers are self-attention layers which have a causal mask applied during training.

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**]

### 3.2.3 Encoders vs. Decoders

As we saw above the encoder differs from the decoder in that the decoder is trained with a causal mask, and the encoder is trained with some other variation such as random masking. When run in inference, they can both be used auto-regressively, but a decoder-only model is more effective at this task, because this is all it was trained to do. The training data for the decoder does not have any other examples in it such as "fill-in-the-blank" tasks.

### 3.2.4 Explicit Decoder Targeting

While it is typical for a decoder to predict the tokens of a sequence in some particular order, we are not limited to this. We can also use the decoder to predict the tokens of a sequence in any order.

This is the approach taken in, for example, “Order Matters: Sequence to sequence for sets” [2].

# 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 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:

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

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:

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

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:

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

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 $90^\circ$ about the $x$ axis, and then by $90^\circ$ about the $y$ axis, we will get the same rotation as if we had rotated by $90^\circ$ about the $y$ axis, and then by $90^\circ$ 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.

$$\begin{aligned} &L_{\theta\text{-MSE}} \colon \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ &L_{\theta\text{-MSE}}(y, \hat{y})_{ni} \overset{\text{def}}{=} \frac{1}{N} \sum_{n} \left[ \sum_{i} (\sin y_{ni} - \sin \hat{y}_{ni})^2 + (\cos y_{ni} - \cos \hat{y}_{ni})^2 \right] \end{aligned} \tag{5.1}$$

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 $\theta$-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 |

## 6.1 Predicting the next frame

## 6.2 Learning a probabilistic model

# Chapter 7

# Conclusions

# Bibliography

[1] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017). arXiv: 1706.03762. URL: http://arxiv.org/abs/1706.03762.

[2] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. *Order Matters: Sequence to sequence for sets*. 2015. DOI: 10.48550/ARXIV.1511.06391. URL: https://arxiv.org/abs/1511.06391.

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