Arbitrary-order Sampling and Hand Motion Modeling with Transformers

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

Transformer models are quickly taking over the field of deep learning, due to their versatility and high performance on many tasks. This thesis provides an introduction to transformer models, presents experiments with new ways of sampling data with them, and then applies them to the domain of hand motion modeling.

Firstly, a comprehensive introduction to transformer models is given, including the attention operation, masking, architecture variants, and different pre-training tasks.

Secondly, an experiment is presented using a transformer model on the MNIST dataset, which was trained so that it is capable of *arbitrary-order* sampling. The experiment compares different sampling orders, including some *dynamic* sampling order heuristics based on the entropy. The experiments find that such sampling orders introduce a statistical bias into the samples.

Lastly, the problem domain of hand motion modeling is introduced, and two transformer models are trained to generate hand-motion sequences, via self-supervised learning on a motion-capture dataset. The first model can generate realistic-looking hand motions, but cannot be directed to generate specific motions. The second model performs poorly.

Acknowledgments

I would like to thank my supervisor, Prof. Bastiaan Kleijn, for his guidance and support throughout this project, and my co-supervisor Dr. JP. Lewis for providing me with the hand motion research direction. I would like to thank the effective altruism community for motivating me to pursue deep learning – there may not be a more pressing problem today than that of AI alignment. I hope I can use what I have learned from this Master's to contribute to ensuring future AI systems do not cause the great harms they will likely be capable of. Lastly, I would like to thank my friends and family for their support and encouragement, and especially Kibra for her patience and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transformer models are quickly taking over the field of deep learning, due to their flexibility and proven usefulness. Code completion, image classification, generative art, and many other tasks are being sucessfully tackled with transformer models.

This thesis provides an introduction to transformer models, presents experiments with new ways of sampling data with them, and then applies them to the domain of hand motion modeling.

1.1 Contributions

Although much of the work done is summarizing others' research and presenting learnings, there are two main novel contributions:

1. Chapter 4 presents experiments with *dynamically-ordered* auto-regressive sampling, utilising the permutation-invariance property of the attention operation in transformer models. The comparison of different dynamic sampling orders is to my knowledge the first of its kind.

2. Chapter 6 presents proof-of-concept transformer-based generative models for hand motion prediction, which can be used to predict sequences of hand motion conditional on arbitrary target frames. This is a novel application of auto-regressive transformer models, and to my knowledge is the first time that a transformer model has been used on this particular problem of hand joint prediction.

These contributions involved training neural networks (see Figure 1.1), in particular transformers, on two datasets - MNIST, and the ManipNet motion capture dataset [33]. The results of these experiments are presented in Chapter 4 and Chapter 6 respectively.

1.2 Structure

The structure of the remainder of the thesis is as follows:

1. Chapter 2 Chapter introduces notation and concepts for neural networks which will be used throughout.

2. Chapter 3 provides an introduction and literature review of a class of neural network models called *transformers*, which are a class of models that have become very broadly used in the past few years.

3. Chapter 4 presents a novel method for sampling sequence data in an arbitrary order, including a pretraining task variant, and experiments with this method with transformer models on the MNIST dataset.

4. Chapter 5 introduces notation and concepts for the problem domain

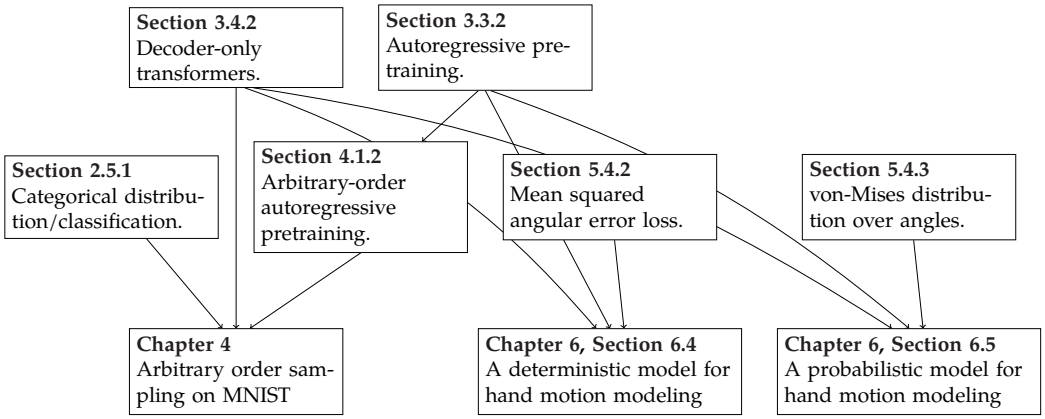

Figure 1.1: How the various concepts in this work are connected to the experiments.

Chapter 4 discusses experiments with different sampling orders for an auto-regressive model trained on the MNIST dataset.

Chapter 6 discusses the development of a model for hand motion modeling using the ManipNet [33] hand motion dataset. Section 6.4 focuses on a deterministic model, while Section 6.5 focuses on a probabilistic model.

of *hand motion modeling*, including methods for representing and working with joint angles and rotations, and character/hand pose data.

5. Chapter 6 then presents the development of a transformer model for predicting hand pose data and generating animations of hands.

6. Lastly, Chapter 7 summarizes the conclusions of the experiments, and discusses future work and reflections.

Chapter 2

Neural Networks and Deep Learning

This chapter introduces neural network notation and concepts which will be used in later chapters.

2.1 Introduction

Since Krizhevsky et al.'s AlexNet [16] in 2012, an incredible number of problems are now being solved with the help Artificial Neural Networks (NN), including image classification, translation of natural language, speech encoding, autonomous driving, computer-generated art and more. 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, which together define the training objective that the NN will learn to achieve.

In this chapter, some notation will first be introduced and clarified so

that it can be used later, and then an overview of the different aspects of neural networks will be given, such as different kinds of training objective.

2.2 Notation

Many of the formalisms in deep learning rely on linear algebra. It is common to see tensors of rank 3 or higher in neural network implementations, along with scalars, vectors and matrices (ranks 0, 1 and 2 respectively). As a result, it is useful to have notation that clearly shows how functions may be applied across the various dimensions of these tensors.

First, a simple example using activation functions.

2.2.1 Tensor index notation

The ReLU activation function is defined as:

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

It is customary to use the symbol ϕ for activation functions. Activation functions are typically scalar functions, but are applied independently across all components of a tensor. This can be represented (and will be throughout) by 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 the *softmax* function can be written, 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:

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

The above function can be applied 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 term $\boldsymbol{X}_b$ refers to the b-th row of X as a vector, as is common in tensor algebra software such as NumPy [12] or TensorFlow [19]. Here however the order of the subscript indices b and n is ignored – the indices index their respectively-named dimensions. **This means that we abandon the distinction between row- and column-vectors**, and must therefore be explicit when representing matrix and vector products.

2.2.2 Neural networks

Some examples of basic neural network operations are given below, to clarify the notation further.

Let N be the input dimensionality of a network, and D 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.

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

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

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:

$$\begin{aligned} &f_0 \colon \mathbb{R}^N \to \mathbb{R}^D \\ &f_0(\boldsymbol{x}) = \phi(W_0\boldsymbol{x}) + \boldsymbol{b} \qquad W_0 \in \mathbb{R}^{N \times D} \qquad \boldsymbol{b}_0 \in \mathbb{R}^D \end{aligned}$$

$$\begin{aligned} &f_\ell \colon \mathbb{R}^D \to \mathbb{R}^D \qquad \forall \ell \in 1, \dots, L \\ &f_\ell(\boldsymbol{x}) = \phi(W_\ell f_{\ell-1}(\boldsymbol{x})) + \boldsymbol{b}_\ell \qquad W_\ell \in \mathbb{R}^{D \times D} \qquad \boldsymbol{b}_\ell \in \mathbb{R}^D \end{aligned}$$

$$\begin{aligned} &f_L \colon \mathbb{R}^D \to \mathbb{R}^C \\ &f_L(\boldsymbol{x}) = \sigma(W_L f_{L-1}(\boldsymbol{x}) + \boldsymbol{b}_L) \qquad W_L \in \mathbb{R}^{D \times C} \qquad \boldsymbol{b}_L \in \mathbb{R}^C \end{aligned}$$

$$
\begin{aligned}
&f_{\text{classifier}} \colon \mathbb{R}^N \to \mathbb{R}^C \\
&f_{\text{classifier}} = f_L \circ f_{L-1} \circ \cdots \circ f_0 \qquad\qquad \theta = \{W_0, \cdots, W_L, \boldsymbol{b}_0, \cdots, \boldsymbol{b}_L\}
\end{aligned}
\tag{2.4}
$$

The parameters of the network are $\theta = \{W_0, ..., W_L, \boldsymbol{b}_0, ..., \boldsymbol{b}_L\}$. 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 will be discussed in the next section.

2.3 Tasks

Neural networks are applied to a wide variety of tasks, such as classification of images, regression of time series data, prediction of tokenized language, and many more. The task that a neural network is applied to determines the loss function that is used to train the network, and the output format of the network. The input format / size / shape of the task also informs the choice of architecture of the neural network – for example, a network that is trained on images will have a 2D input shape (which may be for fixed image sizes or variable image sizes), a network that is trained on tabular data will typically have a fixed-size input shape, and a time series model or language model will have a 1D variable-size input shape.

Different tasks lead to a number of different choices for the architecture and loss function. This section will contextualize the later work by giving a brief overview of the ways that different neural networks and training setups differ.

The following pages show some simple ontologies of the different considerations that combine to define a particular task and architecture, in particular:

1. Training objective / loss function (Figure 2.1)
2. Data dimensionality / length (Figure 2.3)
3. Dataset format (Figure 2.2)

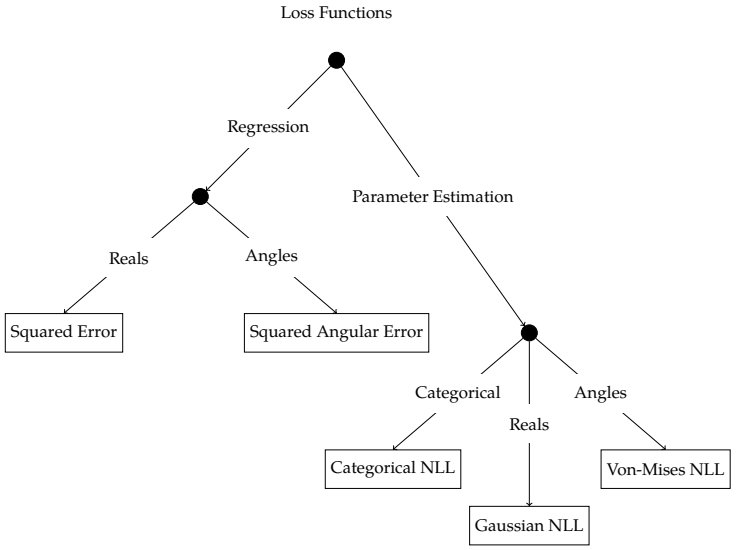

Figure 2.1: We can split loss functions into two categories – regression losses and parameter estimation losses.

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) task.

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, and many more. Depending on the task, and output format, different functional forms for p may be appropriate.

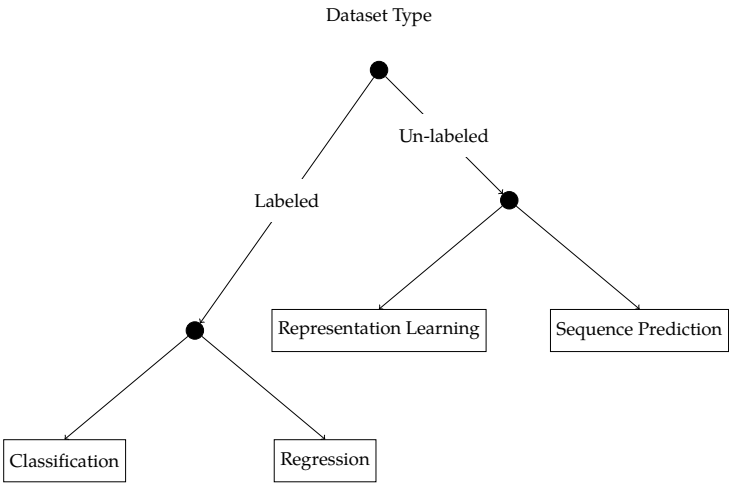

Figure 2.2: Basic ontology of dataset types.

When data is explicitly labeled a model can be trained on a task directly. However, the labeling process is often expensive, and in many cases, the data is unlabeled.

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

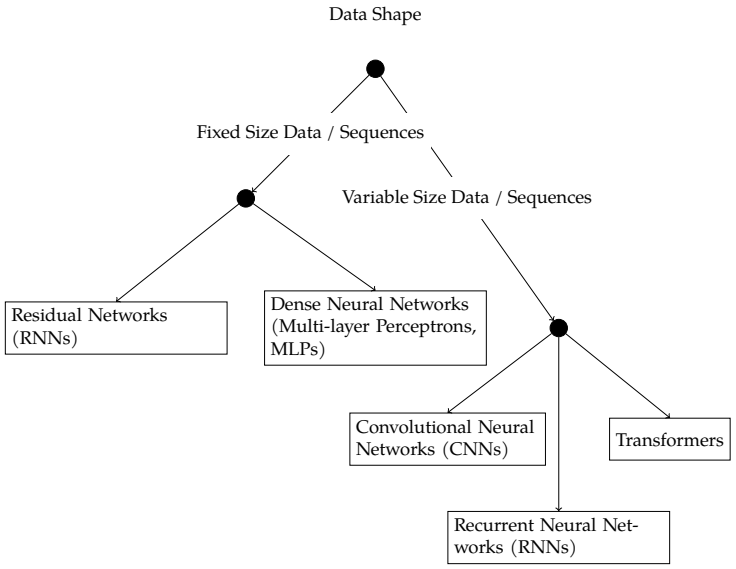

Figure 2.3: Neural network variants which support variable input shape.

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.

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 structure of the data affects the type of model that can be used, and the format of the dataset affects what tasks we can learn from it.

2.4 Deterministic models

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. Given input x, the model output $y = f(x)$ can be interpreted as $\mathsf{E}[p(y|x)]$. Regression is characterized by using an error function as the loss, for example *mean-squared-error*.

2.4.1 Mean squared error

The squared error between two vectors y and $\hat{y}$ is defined as

$$\begin{aligned} &\mathcal{L}_{\text{SE}} \colon \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R} \\ &\mathcal{L}_{\text{SE}}(y, \hat{y})_i \stackrel{\text{def}}{=} \sum_i (y_i - \hat{y}_i)^2 \end{aligned} \tag{2.5}$$

where D is the dimensionality of the output vectors (which can simply be 1). The error is the square of the euclidean distance between the two vectors.

More commonly in neural networks, the mean squared error is used, which is the average squared error over two batches/sequences of vectors

Y and $\hat{Y}$:

$$\begin{aligned} &\mathcal{L}_{\text{MSE}} \colon \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ &\mathcal{L}_{\text{MSE}}(Y, \hat{Y}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n} \left[\sum_{i} (Y_{ni} - \hat{Y}_{ni})^2 \right] \end{aligned} \tag{2.6}$$

where N is the number of samples in the batch.

This function sums the error over the *feature* dimension D and averages the error over the *batch* dimension N. 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.

2.5 Probabilistic models

When modeling data, it is often useful to have a *probabilitic* model of the data, rather than a deterministic model. This allows the model to quantify uncertainty about the model outputs, produce multiple different samples, and avoid problems when the output distribution is multi-modal.

To make a model probabilistic, we train it on a parameter-estimation task. This means the model now outputs the parameters of a probability distribution, and the loss function used corresponds to the functional form of that distribution. This is straightforward for scalar outputs, but when the data being modeled is high-dimensional, probabilistic models often become computationally expensive or intractable, and some different techniques or assumptions must be made.

There are a few common distributions used when training probabilistic

models, such as the Gaussian, Categorical, and Logistic distributions.

2.5.1 Categorical distribution

The most common probability distribution used in neural networks is the categorical distribution, which is used when a task involves predicting discrete variables, commonly classification. The distribution and corresponding loss function are defined as follows

$$\begin{aligned} &p_{\text{Cat}} \colon \mathbb{R}^C \to \mathbb{R} \\ &p_{\text{Cat}}(x = i \mid \theta) \stackrel{\text{def}}{=} \frac{e^{\theta_i}}{\sum_j \exp(\theta_j)} \end{aligned} \tag{2.7}$$

$$\begin{aligned} &L_{\text{Cat}} \colon \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ &L_{\text{Cat}}(y, \hat{\theta}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_n - \log p_{\text{Cat}}(y_n \mid \hat{\theta}_n) \end{aligned} \tag{2.8}$$

where C is the number of classes, and θ are the parameters, and N is the batch size. The equations are written in terms of unnormalized θ (called *logits*), because it is common to implement it this way for better numerical stability with floating point numbers. The categorical distribution is also called the discrete distribution, and the loss function above is equivalent to the "categorical cross-entropy" loss.

2.5.2 Gaussian distribution

Another common distribution to use in parameter estimation is the Gaussian distribution, which is used for continuous variables with unbounded domain (eg. [8]). The distribution and corresponding loss function are

defined as follows

$$\begin{aligned} &p_{\text{Gauss}} \colon \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R} \\ &p_{\text{Gauss}}(x \mid \mu, \sigma) \stackrel{\text{def}}{=} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \end{aligned} \tag{2.9}$$

$$\begin{aligned} &L_{\text{Gauss}} \colon \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\ &L_{\text{Gauss}}(y, \hat{\mu}, \hat{\sigma}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_n -\log p_{\text{Gauss}}(y \mid \hat{\mu}_n, \hat{\sigma}_n) \end{aligned} \tag{2.10}$$

where μ and σ are mean and variance parameters, and N is the batch size. The Gaussian distribution is a continuous distribution, and so the loss function is the mean-squared-error loss.

2.6 Probabilistic models over high-dimensional data

Let us imagine we are modeling a sequence of N observations $\boldsymbol{x}_i \in \mathbb{R}^D$, $1 \leq i \leq N$ (each observation is a D-dimensional vector). To represent the joint distribution over this data, even with a simple gaussian distribution requires $DN + (DN)^2$ parameters (DN means, plus a $DN \times DN$ covariance matrix). If N is large, this is a very large number of parameters, but still manageable.

A gaussian however is limited in its ability to model the data. For example, it cannot model multi-modal distributions. To model multi-modal distributions, we could use a mixture of gaussians, but this increases the number of parameters even further, to $(DN + (DN)^2)K + K$, where K is

the number of mixture components, making sampling and inference much more expensive.

In general, the more expressive the family of distributions we use to model the data, the more parameters we need to represent the distribution and the more expensive it is to sample from the distribution.

There are a few main approaches to address this problem:

- Discretization
- Independence assumptions
- Auto-regressive factorization

2.6.1 Discretization

One approach to managing the intractability of high-dimensional data is to discretize the data, and then model it with a categorical distribution. For example, we can use a clustering algorithm to learn a series of K points within our DN-dimensional space, and then form a discrete distribution over these points. A discrete distribution over K points has $K - 1$ free parameters, so this approach reduces the number of parameters from $(DN)^2$ to $K - 1$, and makes sampling and inference more efficient. However the discretization changes the domain of the data, which may not always be useful. Additionally, the larger the domain, the more cluster points we need to use, and we might not be able to find a good clustering of the data.

2.6.2 Independence assumptions

Independence assumptions are another way to reduce the number of parameters. For example, we could assume that the observations $\boldsymbol{x}_i$ are independent, and then model the sequence as a product of N independent D-dimensional distributions. For a Gaussian, this means fixing parts of the co-variance matrix, and we often reduce to having a single variance parameter. A single variance parameter reduces the number of parameters from $(DN)^2$ to DN, but it also makes the model less expressive. For sequence data, this assumption is almost never valid along the sequence dimension, so this approach is not very useful.

2.6.3 Auto-regressive factorization

Auto-regressive factorization is a third approach to reducing the number of parameters. In this approach, we break down the joint distribution over the sequence into a product of conditional distributions, where each conditional distribution depends only on the previous observations. We then typically model all of these conditional distributions with the same model.

$$\begin{aligned} p(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N) &= \prod_i^N p(\boldsymbol{x}_i | \boldsymbol{x}_1, \dots, \boldsymbol{x}_{i-1}) \\ &= p(\boldsymbol{x}_1) p(\boldsymbol{x}_2|\boldsymbol{x}_1) p(\boldsymbol{x}_3|\boldsymbol{x}_1, \boldsymbol{x}_2) \dots p(\boldsymbol{x}_N|\boldsymbol{x}_1, \dots, \boldsymbol{x}_{N-1}) \end{aligned} \tag{2.11}$$

When we use an auto-regressive model to predict sequences, we usually choose some fixed order for this decomposition. For data with a temporal dimension, this is usually first-to-last, which is usually natural because the

real process that generated the data had a causal structure in the temporal dimension.

However, it is valid to perform the decomposition in any order, for example choosing a random permutation of the sequence(s):

$$J = 5, 3, 9, 1, ..., N$$

$$p(\boldsymbol{x}_1, \boldsymbol{x}_2, \dots, \boldsymbol{x}_N) = p(\boldsymbol{x}_1)p(\boldsymbol{x}_2|\boldsymbol{x}_1)p(\boldsymbol{x}_3|\boldsymbol{x}_1, \boldsymbol{x}_2) \dots p(\boldsymbol{x}_N|\boldsymbol{x}_1, \dots, \boldsymbol{x}_{N-1}) \tag{2.12}$$

In the case of data that does not have a temporal dimension, such as pixels in an image, or joint angles of a hand (within one frame), simple ordering may not always be the best. Also, some data may have very long sequences and latency requirements (such as character animation data), where it is expensive to generate the data in order, and we would instead like to generate only particular parts of the sequence.

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 "Attention Is All You Need", Vaswani et al. 2017 [28]. The distinguishing feature of these models is using one or more *attention* layers to enable information propagation between elements in sequence data. Since 2017, a large number of transformer variants have been developed.

This chapter seeks to understand this class of models at a broader level, including:

- The unique properties of the attention operation, 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 these model are trained on and used

for, building up to the next chapter which uses transformers in their most flexible capacity.

The first and most imporant concept to understand is the operation that under-pins transformers – attention.

3.1 The Attention Operation

Attention is a biologically-inspired mechanism that allows a model to receive inputs from distant parts of the input data, weighted by the *attention weight* given to those inputs, which is typically computed from the data itself. This has proven extremely useful for diverse tasks including machine translation, image generation, and more.

Attention has a number of useful properties which come from its mathematical construction, such as permutation-invariance in the inputs.

3.1.1 Mathematical Definition

An attention operation is of the following form, using short summation notation, where σ is the *softmax* operator (see 2.2)

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

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

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

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

The innermost multiplication of Q and K in the above equation is simply the expanded form of inner product (dot product) between vectors Q_m and K_n. This operation however is not inherent. Instead of the inner product, we can substitute any kernel function $k(q, k)$

$$f_{\text{attn}}(Q, K, V)_{mv} \stackrel{\text{def}}{=} \sum_n [\sigma(k(Q_m, K_n))_{mn} V_{nv}] \tag{3.2}$$

However, substituting a different kernel function is not usually done because the inner product is the most natural choice, and is efficient to compute.

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

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

or, when $k(a, b)$ is chosen to be $a \cdot b$ (the inner product):

$$\begin{aligned} A_{mn} = k(Q_m, K_n) = Q_m \cdot K_n &= \begin{bmatrix} Q_{1,1} & Q_{1,2} & \cdots & Q_D \\ Q_{2,1} & Q_{2,2} & \cdots & Q_D \\ \vdots & \vdots & \ddots & \vdots \\ Q_{M,1} & Q_{M,2} & \cdots & Q_D \end{bmatrix} \begin{bmatrix} K_{1,1} & K_{1,2} & \cdots & K_D \\ K_{2,1} & K_{2,2} & \cdots & K_D \\ \vdots & \vdots & \ddots & \vdots \\ K_{N,1} & K_{N,2} & \cdots & K_D \end{bmatrix}^T \\ &= QK^T. \end{aligned}$$

We can see that the attention weights A have shape $M \times N$. This is the

primary drawback of the attention operation. M and N are typically both large, and the attention weights take $O(MND)$ time to compute (assuming dot product attention), and $O(MN)$ space to store. Despite this drawback, the attention operation has proven extremely useful in a variety of tasks. There are many different ways to address this but they are out of scope here.

The query, key and value vectors Q, K and V are typically computed via three learned projections:

$$\begin{aligned} Q &= W_Q \boldsymbol{X} + \boldsymbol{b}_Q \\ K &= W_K \boldsymbol{X} + \boldsymbol{b}_K \\ V &= W_V \boldsymbol{X} + \boldsymbol{b}_V \end{aligned} \tag{3.3}$$

where $\boldsymbol{X}$ is a sequence of latent vectors – either the embedded inputs to the network, or the outputs of a previous layer.

3.1.2 Permutation-invariance with respect to K and V

The first interesting property of attention is that it is permutation-invariant with respect to the key and value inputs. This property is more or less useful depending on the task. 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, we do not have to introduce any artificial ordering. (However, when *sampling* outputs, we typically still need to decide on some order. This is discussed in more detail in Chapter 4).

This property is due to the construction of the attention operator. We can see that the output O_m corresponding to a query vector Q_m is inde-

pendent of the order of the key and value vectors K_n and V_n, because the summation across n is commutative:

$$O_m = \sum_n V_n \sigma(A_m)_n \tag{3.4}$$

3.1.3 Permutation-equivariance with respect to Q

Relatedly, attention is also permutation-*equivariant* with respect to the query inputs. Equivariance means that the value of the output O_m is dependent on the value of the query vector Q_m, but independent of the order of all *other* query vectors $Q_{m'}$, $m' \neq m$.

For example, let us imagine we have a sequence of four key & value inputs $[K_1, K_2, K_3, K_4]$, and $[V_1, V_2, V_3, V_4]$, and a sequence of three query inputs $[Q_A, Q_B, Q_C]$ which when passed into an attention operation, produce three outputs $[O_A, O_B, O_C]$. First, due to the first property (invariance in K and V), if we swap for example the inputs K_1 & V_1 with K_3 and V_3, the output sequence will remain $[O_A, O_B, O_C]$. Second, if we swap the inputs Q_A and Q_C, the second property (equivariance in Q) means that the output sequence is correspondingly transformed to $[O_C, O_B, O_A]$.

This property is due to the fact that softmax operation is equivariant to the order of its inputs, which we can see from the construction in Equation (2.2).

This property of attention stands in contrast to the two main other methods used to process sequence data, convolution (CNNs) and recurrence (RNNs). Neither of these operations are permutation invariant or equivariant with respect to their inputs.

3.1.4 Dynamic length inputs

The second (and most useful) property of attention is that it can be used to process inputs of dynamic length. We can again see why this is the case from Equation (3.4). The softmax operation normalizes the attention weights, which causes the resulting summation of vectors V_n to be a convex combination. The resulting output O_m will therefore sit within the convex hull of the vectors V_n. This means that the output O_m will be a "valid" output regardless of the length of the input sequence K_v.

3.1.5 Parallel computation

The third property of attention is that it can be computed in parallel across the inputs sequence during training. At all steps of the attention computation except the softmax operation, there are no dependencies between neighbouring elements of the tensors. This stands in contrast to RNNs, where the outputs for one position in the sequence depend on the previous outputs.

The fact that attention can be computed in parallel is a very useful property, since while the operation requires $O(MN)$ in time and space, if we can utilize bigger GPU hardware the real-time cost reduces to $O(1)$ (assuming sufficient memory capacity, compute capability, and also GPU memory bandwidth, which is often the limiting factor [25].) The attention logits A_{mn} can be computed entirely in parallel. The softmax operation depends on all previous attention logits across the key-value dimension N, which requires cross-talk between GPU units but does not prevent parallel computation. The final computation for the outputs O_m can also be computed

in parallel.

So attention is a operation with a number of useful properties. Now we will see how it is used to build a variety of models capable of solving a variety of tasks with sequence data.

3.2 Basic components of transformers.

An attention operation model does not itself make a neural network. It is simply a building block that can be used to construct a neural network.

3.2.1 Residual Blocks

A transformer model puts attention operations together with MLP blocks similarly to in residual networks (ResNets [13]). Without MLP blocks as non-linearities, the outputs of an attention operation are linear functions of the inputs, since the softmax is only used to compute coefficients when summing the values. Because the weights are convex (sum to 1), the output vectors of an attention operation are convex combinations of the value vectors, and will always sit within the convex hull of the value vectors. (However, note that the outputs are non-linear functions of the inputs, due to the fact that the weights are also computed from the inputs). Attention-only we would like to be able to apply non-linear transformations to the value vectors. MLP blocks are used to introduce additional non-linearities into the model.

Figure 3.1 shows a diagram of a typical residual block in a transformer, with an attention operation, an MLP, and a residual connection.

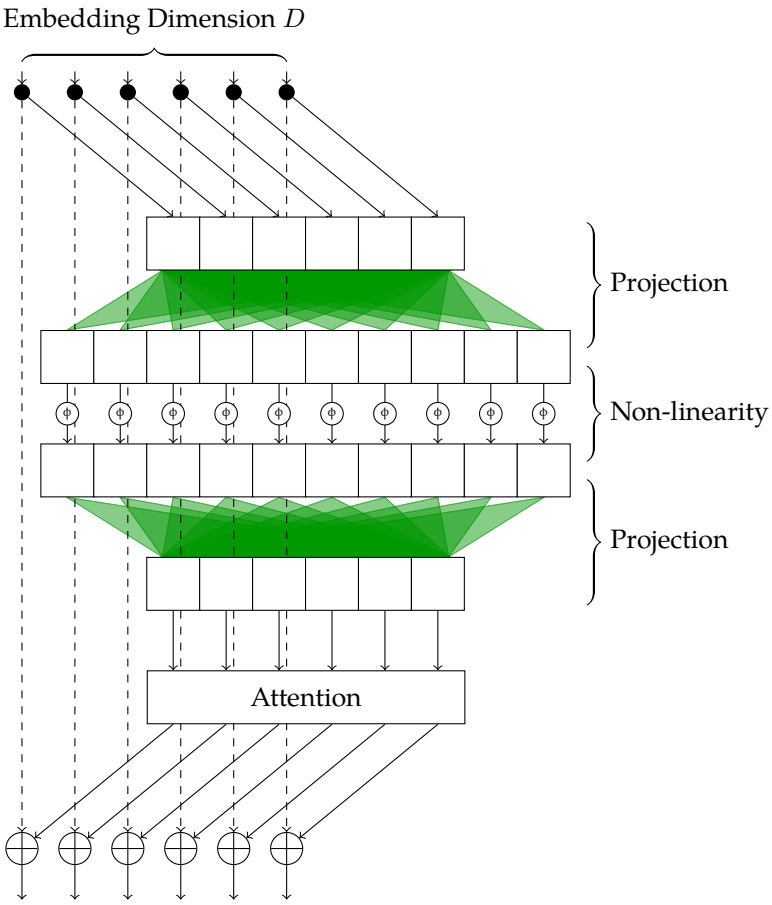

Figure 3.1: Typical residual block in transformer (with the D dimension expanded, rather than the N or M dimension as usual.).

In the MLP block, the sequence of residual latents are (independently) projected into a higher-dimensional space, where a non-linearity is applied, and then projected back down to the original dimensionality. Projecting to a higher dimension allows the block to represent more complex functions, and projecting back down is mostly for computational efficiency. The output of the MLP block is then projected into Q, K, and V spaces, and the attention operation is applied. Finally, the results are added to the residual stream.

3.2.2 Multi-head attention

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

Multi-head attention splits the key, query, and value matrices into n_{heads} smaller matrices, each with dimensionality D_{head}, and computes n_{heads} separate attention matrices. The results are then concatenated together and projected back down to the original dimensionality D:

$$\begin{aligned}
Q &= W_Q \boldsymbol{X} + \boldsymbol{b}_Q \\
K &= W_K \boldsymbol{X} + \boldsymbol{b}_K \\
V &= W_V \boldsymbol{X} + \boldsymbol{b}_V \\
Q_h &= \text{split}(Q, n_{\text{heads}}) \\
K_h &= \text{split}(K, n_{\text{heads}}) \\
V_h &= \text{split}(V, n_{\text{heads}}) \\
A_h &= \text{softmax}(\frac{Q_h K_h^T}{\sqrt{D_{\text{head}}}}) \\
O_h &= A_h V_h \\
O &= \text{concat}(O_h) \\
O &= W_O O + \boldsymbol{b}_O
\end{aligned} \tag{3.5}$$

where split and concat are operations which split and concatenate the tensor along the D dimension.

3.2.3 Input embedding and positional encoding

Attention operates a sequence of latent vectors $\boldsymbol{X}$, but the input to a transformer model will not necessarily be in this format. Mapping the input into this format is the job of the embedding layer. The form of the embedding layer depends on the task being solved. For example, in language modelling, the input is provided as discrete tokens. The embedding layer is then a lookup table (also called a codebook), which maps each token to a learned vector. In timeseries data, the input is provided as a sequence of real-valued vectors. The embedding layer is then a linear transformation, which maps each vector to a learned vector.

Recall that attention is order independent. However, information about the order of inputs is often neccessary to perform a task. When order is important, the embedding layer will include an additional learned vector, which is added to the input vectors. This vector is called a *positional encoding*. The two most common choices for positional encoding are sinusoidal and codebook encodings. A codebook encoding is simply a lookup table, which maps each position in the sequence to a learned vector. Sinusoidal positional encodings were introduced in the original transformer paper [28] and are vectors of (sin, cos) pairs of different frequencies.

To form the input embedding vectors $\boldsymbol{X}$, the embedded 'content' vectors are then typically added to the embedded 'position' vectors.

The next sections will introduce *masking* and *pretraining tasks*, which are the primary ways that transformer architecture variants differ.

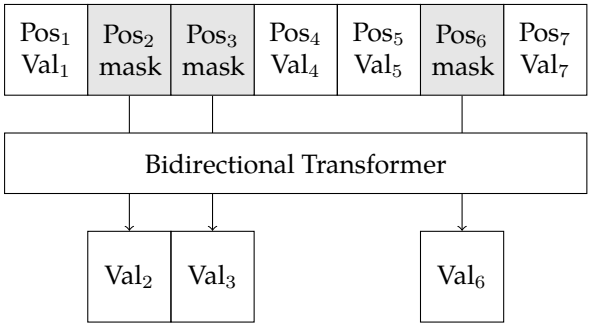

Figure 3.2: Masked sequence modeling pretraining task. The model is trained to reconstruct masked-out elements of the original sequence.

3.3 Masking and Pretraining

When transformer models are trained, they are often trained on large datasets of un-labeled data. The resulting models are then often further trained on smaller labeled datasets, which is called transfer-learning or fine-tuning.

In order to train a model on large unlabeled data, it needs to be formatted into a self-supervised learning task, (also called a pre-training task). This is done via masking out certain inputs and/or connections within the model itself. There are two types of masking, input masking and attention masking, which distinguish the two main pretraining tasks are *masked sequence modeling* and *auto-regressive pretraining*.

3.3.1 Masked sequence modeling

Masked sequence modeling is a pretraining task which uses the first kind of masking – input masking – to define random tasks. In this task, a random subset of sequence elements is masked out, and the model is trained to reconstruct the masked inputs. Input masking means that the 'content'

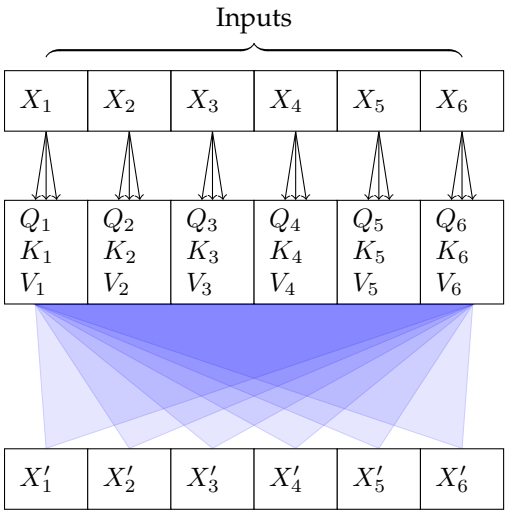

Figure 3.3: Full self-attention (bi-directional attention), as used in transformer encoders. The blue shaded regions show which inputs are used to compute each output. In full self-attention, all inputs are used to compute all outputs.

embedding is replaced with a 'mask' embedding (which is a learned vector). The positional encoding is still provided. A diagram of this is shown in Figure 3.2.

3.3.2 Causal Masking & Auto-regressive pretraining

The second type of masking is *attention masking*. This masking is applied to the attention operation itself. Specifically, a large constant is subtracted from the attention logits for all inputs which are to be masked out. A diagram of the different types of masking is shown in Figure 3.6.

The most common use for attention masking is *causal* attention masking. This masking scheme is used to define a "forward-only" prediction task (also called an auto-regressive task). The masking limits informa-

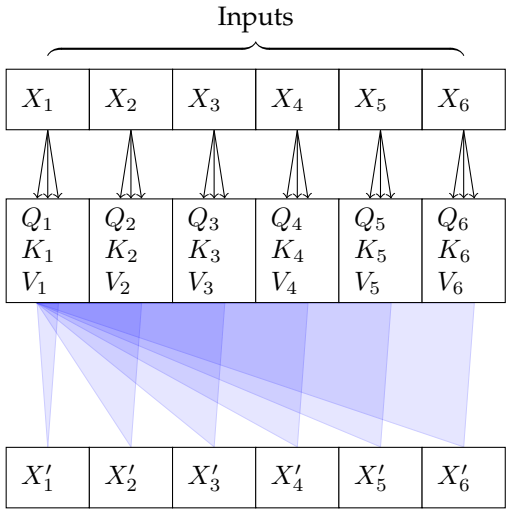

Figure 3.4: Self-attention with causal masking, (uni-directional attention) as used in transformer decoders during training. The blue shaded regions show which inputs are used to compute each output. In causal masking, only inputs to the left of the current output are used to compute the current output.

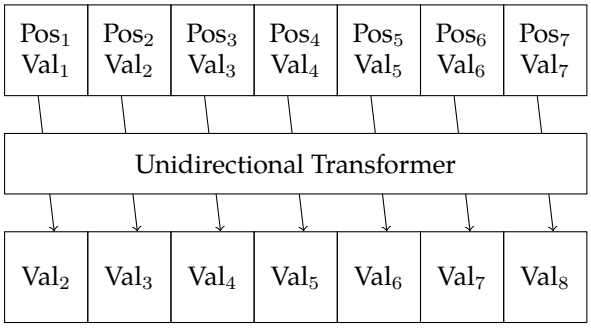

Figure 3.5: Autoregressive pretraining task. The model is trained to predict the next element in the sequence.

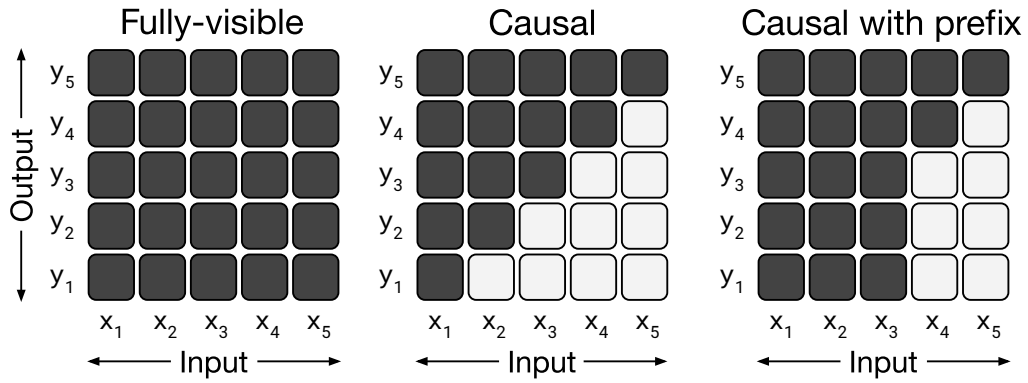

Figure 3.6: Masking in unified attention layers [5, 24]. The bi- and uni-directional attention can be performed within a single attention layer.

tion to only flow in one direction through the model with respect to the sequence dimension. A diagram of causal masking is shown in Figure 3.4.

The inputs and outputs are offset, so that the model is trained to predict the next token in the sequence. This is shown in Figure 3.5.

Because causal masking means information only flows from left to right along the sequence, models which have only been trained on this task are called *unidirectional*. In contrast, models trained on masked sequence modeling use no attention mask (which is sometimes called *full* self-attention, shown in Figure 3.3). This allows information to flow in both directions along the sequence and so models trained with this pretraining task are called *bidirectional* models.

The following chapter experiments with a new pre-training task, which is a variation on auto-regressive sequence modeling.

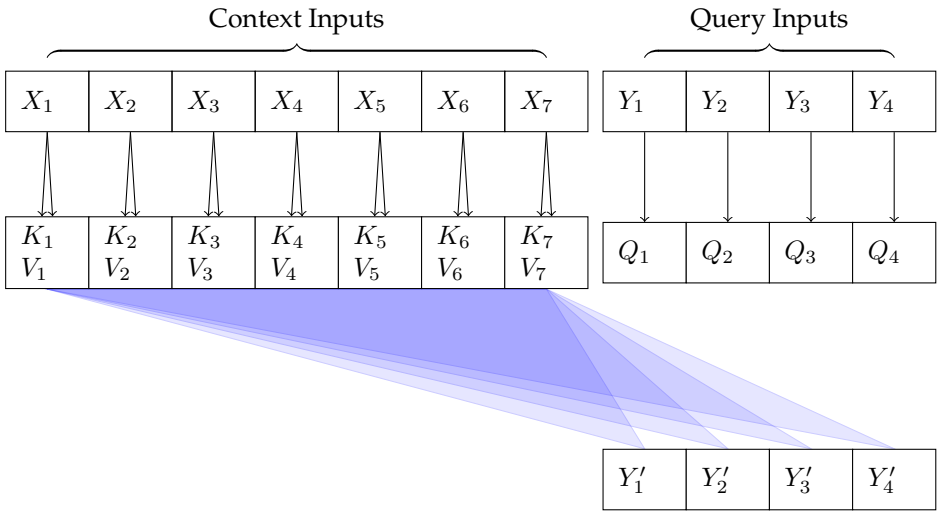

Figure 3.7: Cross-attention, as used in encoder-decoder models.

3.3.3 Unified pretraining

Input masking and attention masking can be combined arbitrarily into a single pretraining task, as we see in Figure 3.6. This is called *unified* pretraining / masking eg. [5, 24].

A model trained with unified pretraining can be used as either a bidirectional or unidirectional model. However, the training is costly, because the attention matrix is $O((A+B)^2)$, where A is the sequence of inputs that will be encoded with bidirectional attention, and B is the sequence of inputs that will be encoded with unidirectional attention.

3.3.4 Cross attention

Instead of using a unified self-attention layer, it is possible to separate the dense and causal parts of the sequence into two separate layers, and connect the two with *cross attention* layers, as shown in Figure 3.7. This reduces

the time and memory complexity of the attention matrix to $O(A^2 + B^2 + AB)$.

3.4 Transformer Architectures

When assembled into a complete model, different combinations of pre-training tasks / masking schemes / attention layers define the different transformer architectures. The next sections will introduce the four most common transformer architectures.

3.4.1 Encoder-only models

Arguably the simplest attention-based model architecture is encoder-only transformers, which are trained on a masked sequence modeling task. Masked sequence modeling is also called Masked Language Modeling (MLM), Masked Image Modeling (MIM), etc. depending on the domain. When used in natural language processing (NLP) they are known as bi-directional language models, because they allow information to flow in both directions. Examples are the BERT [4] language model family, Wav2Vec [2] for speech, and SimMIM [29] image model.

These models are typically used for sequence understanding tasks and classification tasks, however they *can* also be used for generating sequences. The limitation of these kinds of models is that their pretraining task is not efficient for training these models to generate sequences. For generating sequences, a *decoder* model is used, either in conjunction with an encoder, or simply as a decoder-only model.

3.4.2 Decoder-only models

The distinguishing feature of a *decoder* as opposed to an encoder is that during training, all attention layers have a causal mask applied. These models are used for and are trained via auto-regressive pre-training. A diagram of the decoder-only architecture is shown below in Figure 3.8.

This architecture has limited capabilities and so is less common. Some examples of where we see this architecture in use are:

- OpenAI's GPT-series [23, 3] language models.
- Latent code prediction (the "prior") in VQ-GAN [7]

The benefit of this architecture over the encoder-only architecture is that auto-regressive pretraining makes more efficient use of the data than masked sequence modeling. This is because the model is trained to predict the next token in the sequence, rather than predicting a masked token. The training task involves predicting every single sequence element, rather than only the masked out subset. However, the resulting model will only perform well at generating sequences, and cannot be used for eg. sequence understanding tasks.

3.4.3 Encoder-decoder models

When the transformer was introduced in [28], the first architecture proposed was an encoder-decoder architecture. This is a model which has both an encoder and a decoder. The encoder is used to encode a sequence of *conditioning* or *context* inputs, and the decoder is used to generate the output sequence. The encoder and decoder are connected by cross-attention

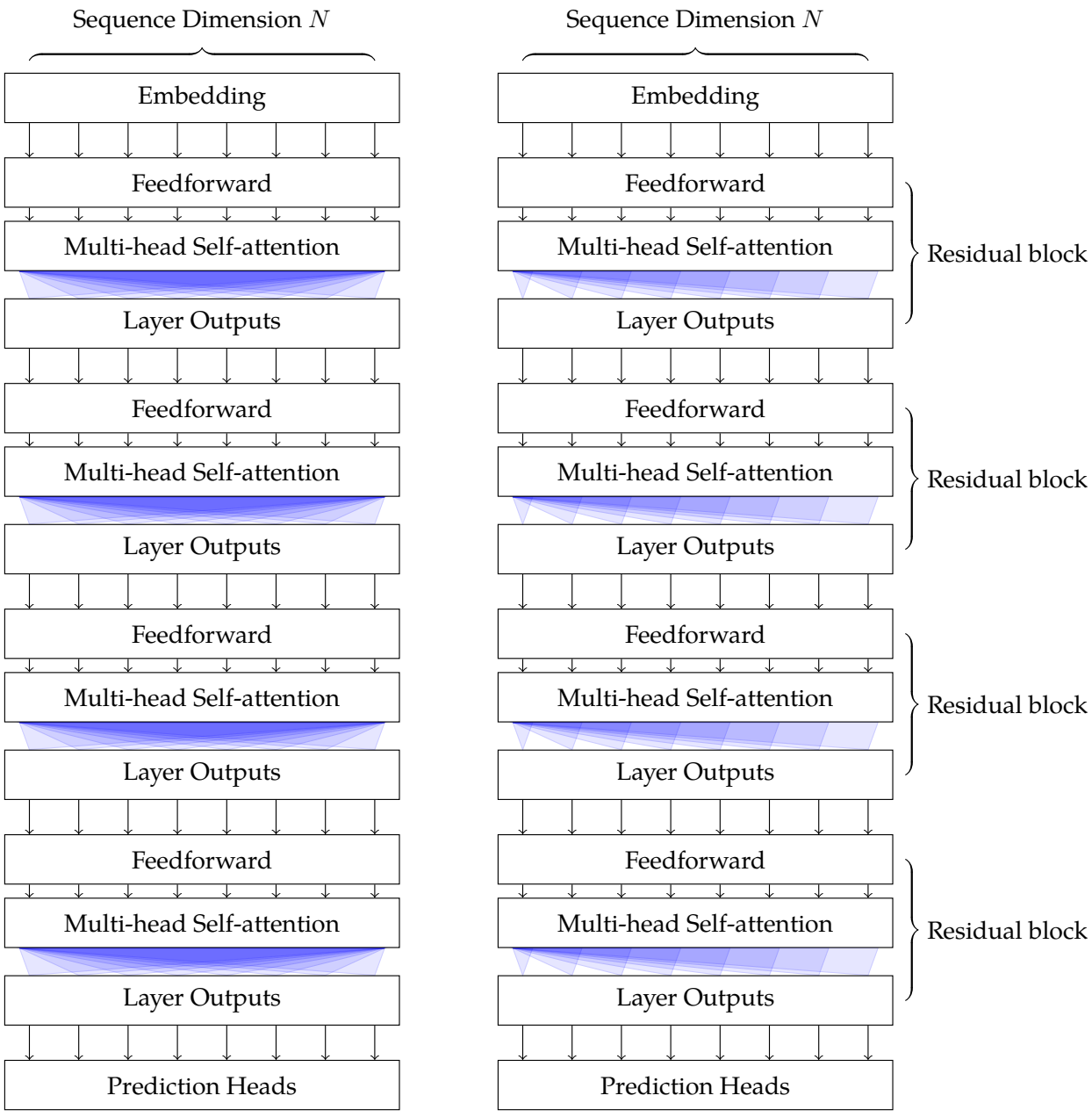

Figure 3.8: Left: Encoder-only model. Right: Decoder-only model. Residual connections have been omitted for brevity.

layers (see Figure 3.7), which allow the decoder to attend to the encoded context sequence.

Encoder-decoder models are more flexible than either previous class of model, because they allow predicting some outputs, or sequences of outputs, *given* some inputs.

Examples of this are the original transformer architecture [28], the BART [17] model, and more recently Google's Parti multi-modal text-to-image model [32].

In [28], they train an encoder-decoder architecture for text translation. Their architecture takes one sequence of text in language A as conditioning input, then auto-regressively samples a sequence of text in a second language B. The two languages can have very different word orderings or numbers of words to each other, and using the cross-attention operation to connect the two sequences introduces no bias towards aligned word orderings or even word counts.

3.4.4 T5 models

The final commonly seen transformer architecture are T5 models [24], which uses unified attention. As we saw before in Section 3.3.3, we can use the same attention matrix for both the encoded sequence and the decoded sequence. However, as discussed, this is less computationally efficient than using cross attention. So, why would this be done?

The answer is that because a transformer shares weights across the sequence dimension, the resulting model will be capable of being used as either a bidirectional or unidirectional model, depending on the masking

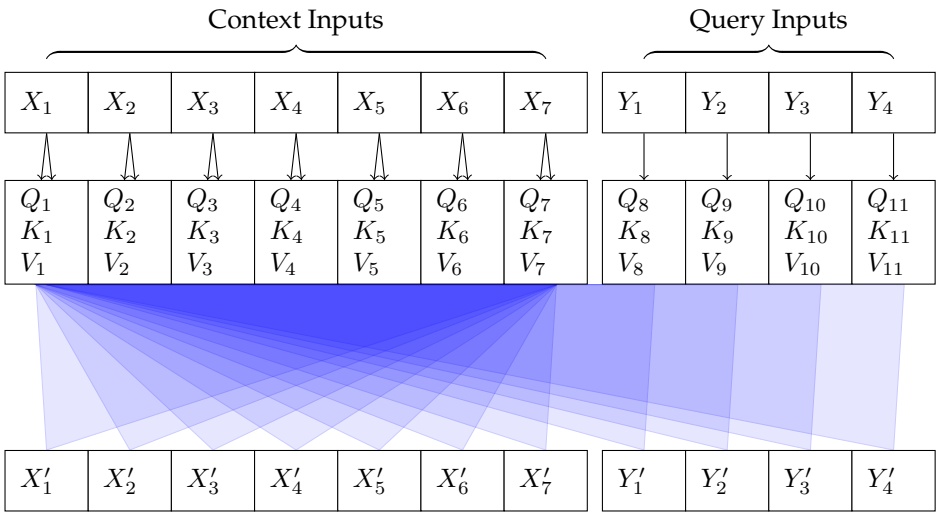

Figure 3.9: Unified self- and cross-attention, from [5].

Bi- and uni-directional attention can be performed with the same attention layers with careful masking, allowing the same model to be trained on any mixture of pre-training tasks.

The "encoder" outputs X' are computed with full self-attention, and the "decoder" outputs Y' are computed with full attention with respect to X and causal attention with respect to Y.

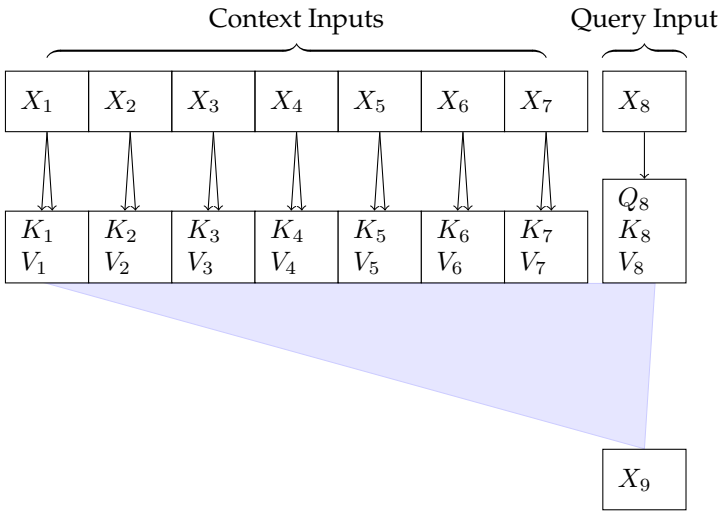

Figure 3.10: Partial self-attention, as used during incremental inference (fast decoding).

scheme used during inference. This is useful for *transfer learning*, where we retrain (fine-tune) a model on a new, more specific task. Having a base model that has been trained with unified attention allows us to fine-tune the same model for both bidirectional and unidirectional tasks.

These models are known as T5 models: **T**ext-**T**o-**T**ext **T**ransfer **T**ransformers [24]. Training a large T5 model is very computationally expensive, but the resulting models are extremely useful for transfer learning. Currently, many state of the art results are being achieved by fine-tuning these models.

3.5 Fast decoding

When training transformer models, an entire sequence of inputs is provided at once. However, at inference time not all the computations need to

be performed to generate sequences. The computed key and value matrices can be reused for multiple predictions, and the attention matrices can be computed incrementally. When this is done attention operation looks as shown in Figure 3.10.

Chapter 4

Sampling Sequences In Any Order

The unique properties of transformer models allows great flexibility in the kind of tasks that they can be trained to perform. This chapter shows that with minor changes to the model architecture and training procedure, a transformer is able to generate data in any order, including dynamically choosing the order of the data while sampling. The resulting model is a stochastic process on the dataset.

4.1 Introduction

What is the best order to sample pixels in an image, to maximize the quality of the sample as a whole? Is it unimportant, in which case any order will do? Or does the optimal sampling order depend on the data itself?

To answer this question, a transformer-based neural network model is trained so that it can be used auto-regressively sample sequences of pixels

in any order.

The model is then used to compare the following sampling orders:

- Left-to-right, top-to-bottom
- Random
- Highest-entropy-first
- Lowest-entropy-first

Contrary to my hypothesis that a lowest-entropy-first sampling order would result in the best samples, such sampled images are biased towards images with large amounts of empty background, such as images of 1s. Correspondingly, images sampled with highest-entropy-first sampling order are biased towards images of 8s and 9s. In the discussion, I hypothesize this occurs because the criterion we are selecting for introduces a bias, and I characterise this bias.

4.1.1 Dynamically-ordered auto-regressive sampling

If we have an auto-regressive model of the appropriate form that has been appropriately trained, we can dynamically choose the order that we sample a sequence. The form of the model must be such the data $\boldsymbol{x}_i$ can be split into two components 'position' and 'content' which we will represent with x_i and y_i respectively.

Given some seed sequence length i we have input data $\boldsymbol{x}_{<i} = \{(x_n, y_n) \mid n < i\}$. For each remaining position x_n, $n \geq i$, we can compute the conditional distribution $p(y_n \mid x_n, \boldsymbol{x}_{<i})$. This gives us $N - i$ conditionally-independent distributions.

Given that we have a number of conditionally-independent distributions $p(y_n \mid x_n, \boldsymbol{x}_{<i})$, and we can sample from any of them, which one *should* we sample from? This is the question which the experiment in this chapter tries to answer. We can choose any statistic of these distributions as a heuristic to decide the sampling order. For example, we could choose the distribution with the highest entropy, or the distribution with the lowest entropy, or the distribution with the highest mean, or the distribution with the lowest mean, etc. We can then sample from the chosen distribution to get the next y_n.

For the experiments we will compare the following sampling orders:

- Fixed order (Left-to-right, top-to-bottom)
- Random
- Highest-entropy-first
- Lowest-entropy-first

Because the data are being sampled, and the auto-regressive factorization of a sequence is in principle independent of the order of the factorization, we might expect that the order has no effect – however in practice, the distribution of results may be affected because the model is imperfect, and the sampling process may introduce statistical bias when choosing the next location to sample. This will be discussed more in Section 4.7.

4.1.2 Arbitrary order auto-regressive pre-training

Arbitrary order auto-regressive sequence modeling is a variant of auto-regressive pretraining developed specifically for this project. Let us re-

call causal/autoregressive pretraining from the previous chapter. (See Figure 3.5) Recall that these predict the next input from the previous input, conditioned on the rest of the sequence via their attention layers.

The input sequence can be represented as:

$$\boldsymbol{x} = (y_{<i}, x_{<i}) = (\{y_i, y_{i-1}, ..., y_1\}, \{x_i, x_{i-1}, ..., x_1\})$$

where i represents the position of a token, and x represents the value of a token. When predicting the next input from the previous input, the model typically infers the next position from the previous position during the process of predicting.

However, if the data is not in order, then the the model is not able to infer the position to predict. We instead construct an input sequence in the following way, providing the model with the *target* position information explicitly:

$$\boldsymbol{x} = (x_{<i+1}, y_{<i}, x_{<i}) = (\{x_{i+1}, x_i, ..., x_2\}, \{y_i, y_{i-1}, ..., y_1\}, \{x_i, x_{i-1}, ..., x_1\}).$$

By providing the input as (target position, input position, input value] triples instead of [input position, input value] pairs (in which the target is implicit), we allow the model to predict the next input value without inferring the next position.

If our sequence was presented in contiguous forward-only ordering, the target position of x_i would always be the same, and so the addition of x_{i+1} would not introduce any new information. However, in this pretraining task we randomly shuffle the order of the tokens, so the model learns to utilize this information.

At inference time we can choose x_{i+1} to be any position that we want the

model to predict next, by constructing the following triple (x_{i+1}, y_i, x_i) and appending it to the rest of the previous tokens. This enables performing the experiments with dynamic sampling orders.

4.2 Previous work

A stochastic process is a sequence of random variables, where each random variable depends on the previous random variables. Equivalently, a stochastic process is a function drawn from a probability distribution over functions. Stochastic processes are extremely useful objects. Their most interesting use is for Bayesian Optimization (which we will not discuss here), but they are also used in many other areas, such as in the study of dynamical systems, and in the study of random walks.

This section discusses some previous work relating neural networks and stochastic processes, and how it relates to the work in this chapter.

There are a number of previous works that have explored the use of neural networks as stochastic processes:

- Neural Processes [10]
- Attentive Neural Processes [15]
- Transformer Neural Processes [22]
- Transformers Can Do Bayesian Inference [20]

What these works have in common is that create some kind of learned model which fulfills the properties of a stochastic proceess. A key aspect that defines a stochastic process is that the distribution over the function

space is invariant to permutations of the input sequence, and that the inputs sequence can be of any length. 'Neural Processes' use a mean pooling operation to achieve this, but the subsequent works use the attention operation to achieve this.

The models can be used draw samples from any point in a function space, then can be conditioned on the newly sampled points to form a new learned distribution over the same function space.

The main difference between current approaches to neural stochastic processes is which family of distributions the learned model uses. In 'Neural Processes' and 'Attentive Neural Processes', the model learns a distribution over latent variables, samples from which are then fed into a decoder to determine a function. In other words, the model is a distribution can be used to sample a function. In 'Transformer Neural Processes', and 'Transformers Can Do Bayesian Inference', the model outputs the parameters for a distribution over functions directly. In other words, the model is a function which outputs distributions.

The work in this chapter takes the second approach – learning a function which outputs distributions. Because this approach gives access to closed-form distributions, it is possible to compute the entropy, probability density, and other quantities of interest. This is not possible with the other approaches, which only enable sampling from the distribution. However, the other approaches are able to learn more complex distributions.

The above works [22] and [20] are the most similar to the work in this chapter. The model trained in this chapter is exactly a 'Transformer Neural Process', except that the model in this chapter outputs discrete distributions rather than Gaussians.

The work in this chapter investigates the effect of choosing a dynamic sampling order for image completion with a transformer neural process based on the entropy of the distribution. To my knowledge none of the previous works have investigated this.

4.3 Hypothesis

By training a transformer model with the pretraining method described in Section 4.1.2, we get a model which can be used to sample any position in the image. We can then use this model to sample the image in different orders, and compare the results.

Using the model 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.

My hypothesis was 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

To investigate this hypothesis, a transformer neural process was trained with the pretraining method described in Section 4.1.2, on the MNIST dataset. The model was then used to sample images in different orders, and the results were compared.

4.4.1 Data

This series of experiments uses the MNIST dataset, which is a set of 28x28 grayscale images of handwritten digits. The dataset was split into a training set of 60,000 examples, and a test set of 10,000 examples. Each image is labeled with the digit it represents, from 0 to 9, but the labels were not used in the experiments. Each pixel is represented as a value between 0 and 255, where 0 is black and 255 is white.

Instead of representing full 256 colors, a 2-bit representation was used, where each pixel is represented as a value between 0 and 3. The 256 colors were discretized further into 4 colors using a learned k-means clustering over the all pixel values in the MNIST training set. This was done to simplify the form of the distribution that the model would output, and remove any complextity here as an additional variable to debug. 4 colors was the smallest representation that gave good visual quality when the images were reconstructed.

The MNIST images were then formatted into a sequence prediction task using the following procedure. First, each image was treated as a sequence of $28 \times 28 = 784$ triplets, where each triplets contains three integers: the row index of the pixel and the column index of the pixel (both in the range $[0, 28)$), and the pixel value (in the range $[0, 4)$ – 0 = black, 1 = dark gray, 2 = light gray, and 3 = white).

The sequence of triplets was then shuffled to form a random permutation of the pixels. The row and column indices of each pixel were then appended to the previous triplet, forming a 5-tuple. This was done to specify the target position that the model should predict, which it could no longer

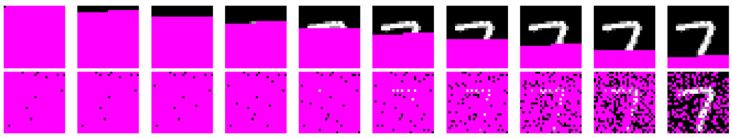

Figure 4.1: Examples of two tasks, showing input sequences with progressively more pixels filled in. (Pink means the pixel was not provided).

Top: Pixels presented in a contiguous, forward-only order.

Bottom: Pixels are presented in a non-contiguous, random order. This is the task that the model in this experiment was trained on.

---

infer from the ordering of the sequence since the sequence was shuffled.

Lastly, all of the integers each 5-tuple were embedded using learned codebooks, and the resulting sequence of embedding vectors was used as the input to the model.

Some examples of this task are shown in Figure 4.1.

4.5 Training

A transformer model was then trained on the described task. A table of training details is shown in Table 4.1.

4.6 Results

Some predicted images using this model are shown in Figure 4.2. For comparison, a deterministic model trained on a standard forward-only autoregressive pretraining task is shown in Figure 4.3. The deterministic model was trained for the same number of steps, and with the same hyperparameters as the probabilistic model, but was trained on a different task and

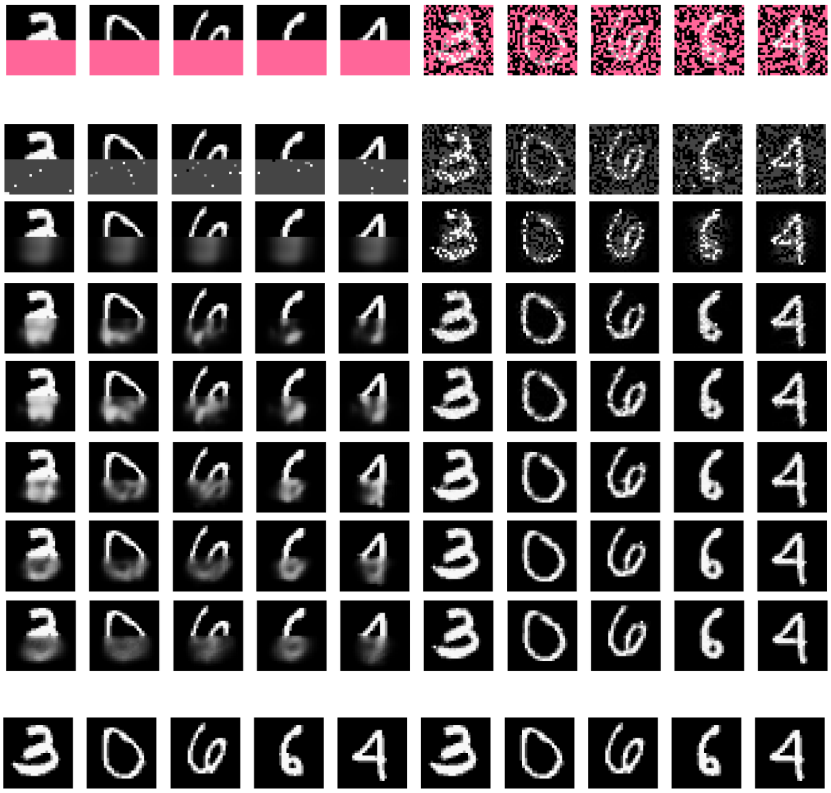

Figure 4.2: The above figure shows the probabilistic 4-color model converging as it is trained. This was the model used for the sampling order experiments.

Top: Seed inputs provided to the model. Pink pixels represent inputs not provided.

Middle, descending: The expected value of the model's distribution over the unknown values as the training process progresses.

Bottom: Ground truth images.

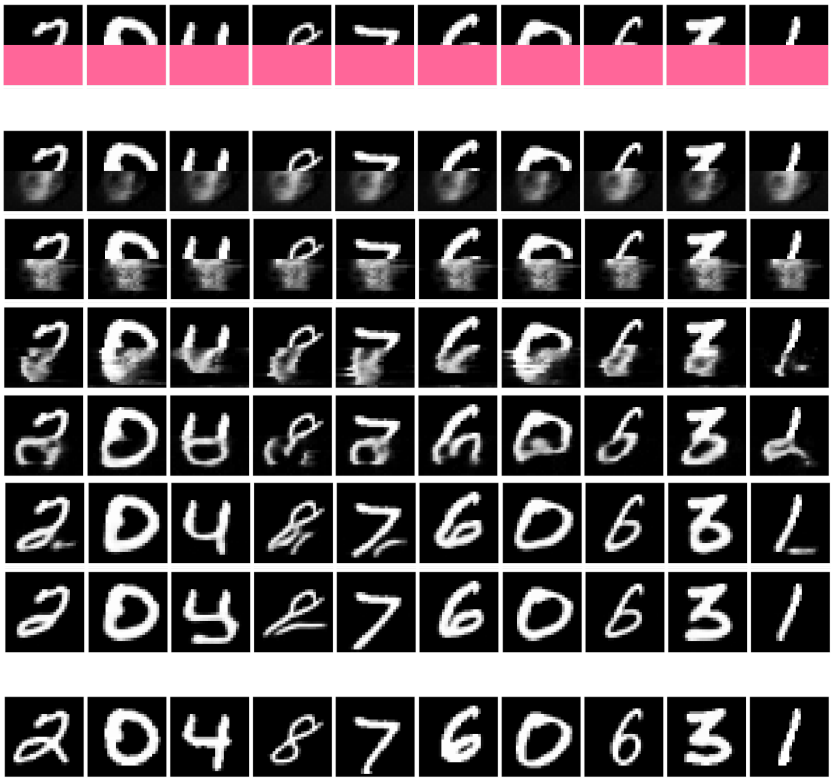

Figure 4.3: The above figure shows a deterministic model converging as it is trained. This model predicts the pixel values as a floating point value directly, rather than as a probability distribution.

Top: Seed inputs provided to the model. Pink pixels represent inputs not provided.

Middle, descending: The model's prediction over the unknown values as the training process progresses.

Bottom: Ground truth images.

N° embedding dims	256
N° layers	5
N° hidden dims	515
N° attention heads	12
Total N° parameters	17,495,809
Batch size	32
Sequence length	784
Training time	1h 27m 4s
Training steps	20,000

Table 4.1: Training details for the probabililistic 4-color model.

with a different loss function.

The results of using the model to sample images using different sampling orders are shown in Figure 4.4.

4.7 Discussion

As we can see in Figure 4.4, the "lowest-entropy-first" ordering produces distinct images from the "highest-entropy-first" ordering. However, neither are as good as the "random" ordering. This clearly disproves the hypothesis that the "lowest-entropy-first" ordering would produce the best results.

Why do the "lowest-entropy-first" and "highest-entropy-first" orderings produce such different results? Why should they be different from the "random" ordering?

If the model has perfectly learned the true distribution of the data, then all orderings should produce the same results. However, the model is not

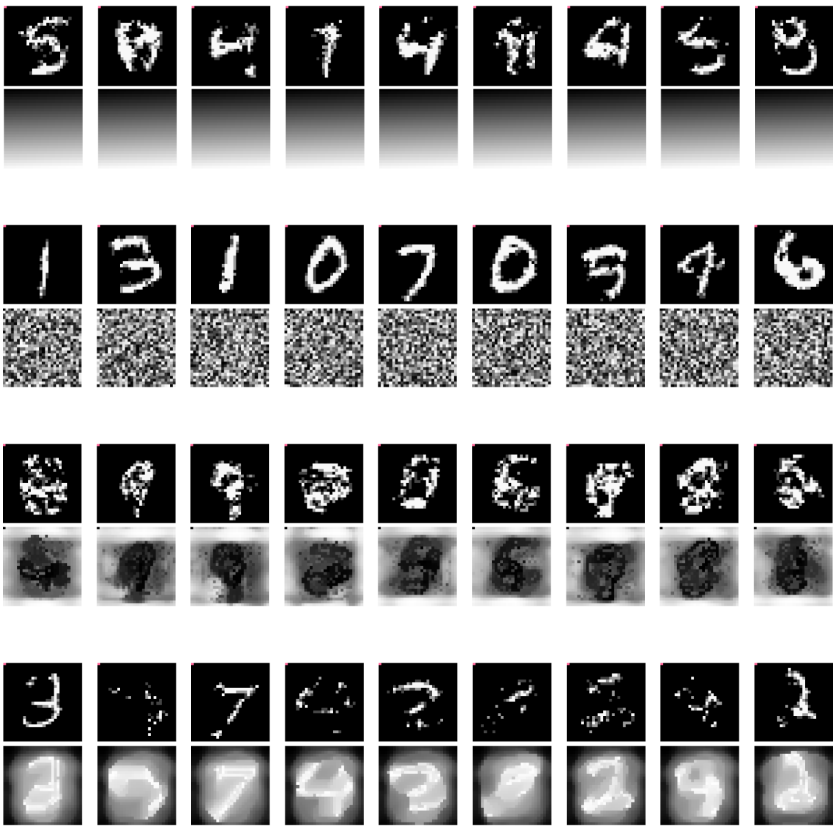

Figure 4.4: The above figure shows the effect of the sampling order on the visual quality of the predicted images, using the probabilistic 4-color model. Each pair of rows shows predicted images and a visualization of the sampling order used to generate it, respectively. The bottom row of each pair is a visualization of the sampling order, where the color of each pixel indicates the order in which it was sampled – black pixels were sampled first, and white pixels were sampled last.

Top: Sequential sampling order, where pixels are sampled in the order they appear in the image.

Top-middle: Random sampling order, where pixels are sampled in a random order.

Bottom-middle: Highest-entropy-first sampling order, where pixels are sampled in order of decreasing entropy.

Bottom: Lowest-entropy-first sampling order, where pixels are sampled in order of increasing entropy.

perfect, and the "random" ordering is the only one that is not biased by the model's imperfections.

When we select a dynamic ordering based on the model's predictions, we are introducing a bias into the model's predictions. Let us examine this bias in more detail.

Let us imagine that the model outputs a gaussian =- more specifically it outputs estimates of the parameters μ and σ of the true conditional distribution $p(y_i \mid x_i, y_{<i}, x_{<i})$. Then, also assume we can approximate the the fact that the model is imperfect by adding gaussian noise to the model's output, $\mu + \epsilon_\mu$ and $\sigma + \epsilon_\sigma$, where $\epsilon_\mu \sim \mathbb{N}(0, v_\mu)$ and $\epsilon_\sigma \sim \mathbb{N}(0, v_\sigma)$, for some small v_μ and v_σ. Let the model's output distribution be $q(i) = \mathbb{N}(y_i | x_i, y_{<i}, x_{<i}, \mu + \epsilon_\mu, \sigma + \epsilon_\sigma)$.

If we select the next position i randomly, then when we sample $y_i \sim q(i)$, since the means of the error terms ϵ_μ and ϵ_σ are both 0, the expected value of y_i remains μ.

However, when we select the next location i to sample based on the entropy of $q(i)$, we select the location among many which has the highest (or lowest) variance $\sigma + \epsilon_\sigma$. On average, we will select a position with both high contribution from σ, **and** high contribution from ϵ_σ. Because of the high ϵ_σ term, this selection biases us towards sampling from distributions where the model is more uncertain than in the true distribution. We will therefore draw samples that are on average **less likely** in the true distribution. i.e. $\mathsf{E}[p(y_i \mid x_i, y_{<i}, x_{<i})] < \mathsf{E}[q(i)]$. To summarize, when we select an i because the corresponding $q(i)$ has high entropy (variance), and then sample from this distribution, we will produce a pixel with a value that has $p(y) < q(y)$. The reverse is true for the "lowest-entropy-first" ordering.

This is the bias that we are introducing into a *single* prediction from the model.

I claim this same reasoning applies for the discrete case which I actually used in the experiment – we can add an ϵ term to the logits, which when selecting for high entropy, pushes them towards the uniform distribution, and when selecting for low entropy, pushes them away. It so happens that on MNIST, this typically means the pixel will be brighter.

As we repeat this process, we will produce some pixels that are on average brighter than the true sequence. When the model is conditioned on these, it will generally infer that the remaining pixels should be brighter as well. This is why the "highest-entropy-first" ordering produces images that are as a whole brighter than the "lowest-entropy-first" ordering, and why both are shifted away from the "random" ordering.

4.8 Conclusions and Future Work

In this chapter, we have shown that the order in which we sample from a model can in practice have an impact on the quality of the samples produced, despite different sampling orders being equivalent in principle. However we also showed that sampling with "lowest-entropy-first" and "highest-entropy-first" produce a directional bias in the samples produced. This bias is due to the fact that the model is imperfect, and selecting locations based on the entropy has the effect of compounding these imperfections. With some assumptions about the nature of these imperfections, we showed that sampling with a "highest-entropy-first" ordering shifts the samples towards the uniform distribution, and sampling with a "lowest-

entropy-first" ordering shifts the samples away from the uniform distribution.

Future work could explore the effect of this bias under different classes of probability distributions, and investigate other sampling orders such as a learned order.

Chapter 5

Angles, Joints and Hands

The next chapter discusses the development of a model to predict motion of a human hand. To understand the model and some of the implementation choices, it is necessary to understand the anatomy of the hand, and how the many degrees of freedom of the hand can be represented (paramaterized). This chapter discusses the different ways to parameterize angles, 3D rotations, and collections of many joints. It also introduces some loss functions for training neural networks where the input and output are angles.

5.1 Representing Angles

In toplogical terms, an angle is a point on the unit circle, aka the 1-sphere S^1. In the context of 3D rotations, an angle is a rotation around a single axis. We typically use either of two representations for angles: A single number or a 2D unit vector.

Representing an angle as a single number (in either radians or degrees) is the most common representation. It is simple and addition between an-

gles represented in this way is well defined.

The other representation is the unit vector representation. In this representation, an angle a is represented by a 2D unit vector v

$$v = \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \cos a \\ \sin a \end{bmatrix} \tag{5.1}$$

. This representation is useful for neural network implementations for two reasons:

1. The dot product between two unit vectors is equal to the cosine of the angle between them. This means that the dot product between two unit vectors can be used as a measure of the similarity between two angles.

2. Linear interpolation between two unit vectors gives a vector which has the same angle as the linear interpolation between the two angles. This means that if we re-normalize the vector, we get a unit vector which represents the same angle as the linear interpolation between the two angles.

Using the unit vector representation also makes it simpler to define the *circular mean* operation (which will be used later in Chapter 6)

5.1.1 Circular mean

Take for example two angles in radians, $a = 0$ and $b = 2\pi - 1 \approx 5.28$. They represent angles 1 radian apart, but naively computing the midpoint between them would give $\pi - 0.5 \approx 2.64$, which is not correct. The correct answer should be $2\pi - 0.5 \approx 5.78$.

Performing this operation is more natural if we use the vector represenation v_a and v_b. Using this representation we can define the *circular mean* of the two angles v_a and v_b as

$$v_{\text{mean}} = \frac{v_a + v_b}{\|v_a + v_b\|} \tag{5.2}$$

where $\|v\|$ is the magnitude of a vector v. Note that this is not defined when $v_a + v_b = 0$, ie. when the two angles are opposite each other.

5.2 Representing Rotations

Joints in a human body are connected by bones. The bones are connected by joints, which allow the bones to rotate relative to each other in 3 dimensions. In order to represent a hand pose we must therefore represent rotations in 3 dimensions. There are a few different ways to parameterize rotations: Euler Angles, Axis-Angle and Quaternions.

5.2.1 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.

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.

5.2.2 Axis-Angle

An alternative parameterization for the rotation of a 3D object or joint is to use an axis-angle representation. In this representation, each joint is assigned a 3-component unit vector and an angle. The unit vector specifies an axis of rotation, and the angle specifies an amount of rotation about that

axis. It might be surprising that any rotation in 3D can be represented this way.

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, there are fewer of them, and they are easier to avoid. An example of non-uniqueness is when the angle of rotation is zero, this corresponds to no rotation at all regardless of the axis of rotation.

This parameterization is also easier to interpolate between. 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.

However, it is not widely used in practice. The reason is that it is not as easy to understand and implement as the Euler angle parameterization. The main issue is that it is not very intuitive to choose an axis of rotation. When the axis is a coordinate axis it is straightforward, a rotation is to be performed which would involve multiple coordinate axis in an Euler angle representation, instead a single axis must accurately be chosen, which is not very intuitive.

5.2.3 Quaternions

The most natural representation of rotations in 3D is the quaternion. Quaternions get their name from quaternion algebra, which is a generalization of complex numbers (2-component numbers) to 4-component numbers as

follows:

$$\mathbf{q} = a + bi + cj + dk = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} \begin{bmatrix} 1 \\ i \\ j \\ k \end{bmatrix} \tag{5.3}$$

where the multiplication of additional axes/constants j and k satisfies the following rules:

$$\begin{aligned} i^2 = j^2 = k^2 = ijk = -1, \\ ij = k, \\ jk = i, \\ ki = j. \end{aligned} \tag{5.4}$$

A rotation is represented by a quaternion of norm 1, and in computer graphics, the term "quaternion" typically refers to such quaternions – the more precise terminology is "unit quaternion" or "versor". Because in this thesis quaternions are only used for rotations, the term "quaternion" is used to refer to unit quaternions/ve from here on.

What separates quaternions from regular 4-vectors is the quaternion product, which is also known as the Hamilton product. It is defined as follows:

$$
\begin{aligned}
q_1 \cdot q_2 &= (a_1 + b_1 i + c_1 j + d_1 k)(a_2 + b_2 i + c_2 j + d_2 k) \\
&= a_1 a_2 + a_1 b_2 i + a_1 c_2 j + a_1 d_2 k \\
&\quad + b_1 a_2 i + b_1 b_2 i^2 + b_1 c_2 ij + b_1 d_2 ik \\
&\quad + c_1 a_2 j + c_1 b_2 ji + c_1 c_2 j^2 + c_1 d_2 jk \\
&\quad + d_1 a_2 k + d_1 b_2 ki + d_1 c_2 kj + d_1 d_2 k^2 \\
&= (a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2) \\
&\quad + (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2) i \\
&\quad + (a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2) j \\
&\quad + (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2) k.
\end{aligned} \tag{5.5}
$$

To use a quaternion q to rotate a 3D vector v, we first represent the vector as a quaternion p with real part 0:

$$
p = \begin{bmatrix} 0 \\ v_x i \\ v_y j \\ v_z k \end{bmatrix}. \tag{5.6}
$$

We then rotate the vector by the quaternion q by multiplying q by p and then by the inverse of q:

$$
q \cdot p \cdot q^{-1} = \begin{bmatrix} 0 \\ v'_x i \\ v'_y j \\ v'_z k \end{bmatrix}, \tag{5.7}
$$

where v'_x, v'_y, and v'_z are the components of the rotated vector. The inverse

(or reciprocal) of a quaternion is defined as follows:

$$q^{-1} = \frac{q^*}{||q||^2}, \tag{5.8}$$

where q^* is the conjugate of q:

$$q^* = \begin{bmatrix} a \\ -bi \\ -cj \\ -dk \end{bmatrix}. \tag{5.9}$$

Any two rotations q_1 followed by q_2 can be composed into a single rotation q_3 as follows:

$$q_3 = q_2 \cdot q_1. \tag{5.10}$$

Note that the quaternion product is not commutative, because the order of the rotations matters. If we want to rotate a vector by q_1 and then by q_2, we $q_2 \cdot q_1$ in that order. However, if we want to rotate a vector by q_2 and then by q_1, we must do $q_1 \cdot q_2$ in that order.

The space of quaternions (as commonly referred to in computer graphics) has the topology S^3, which is the same as the space of rotations $SO(3)$, so it is a perfect representation. There are no singularities. Interpolation is also straightforward, as we can simply interpolate between the two quaternions in a great circle on the 3-sphere (in 4D space).

Quaternions are widely used as the back-end representation when performing operations on rotations. However, they are not widely used as the front-end representation, for example in 3D modeling software. The reason is that they are not very intuitive to use - the user would have to have an intuitive grasp of the 3-sphere. Instead, it is common for software to

use the Euler angle parameterization, and then convert to quaternions for internal use when interpolation or composition is required.

5.3 Parameterizing Hand Poses

To a first approximation, the human hand has 23 degrees of freedom, which are shown in Figure 5.1.

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.

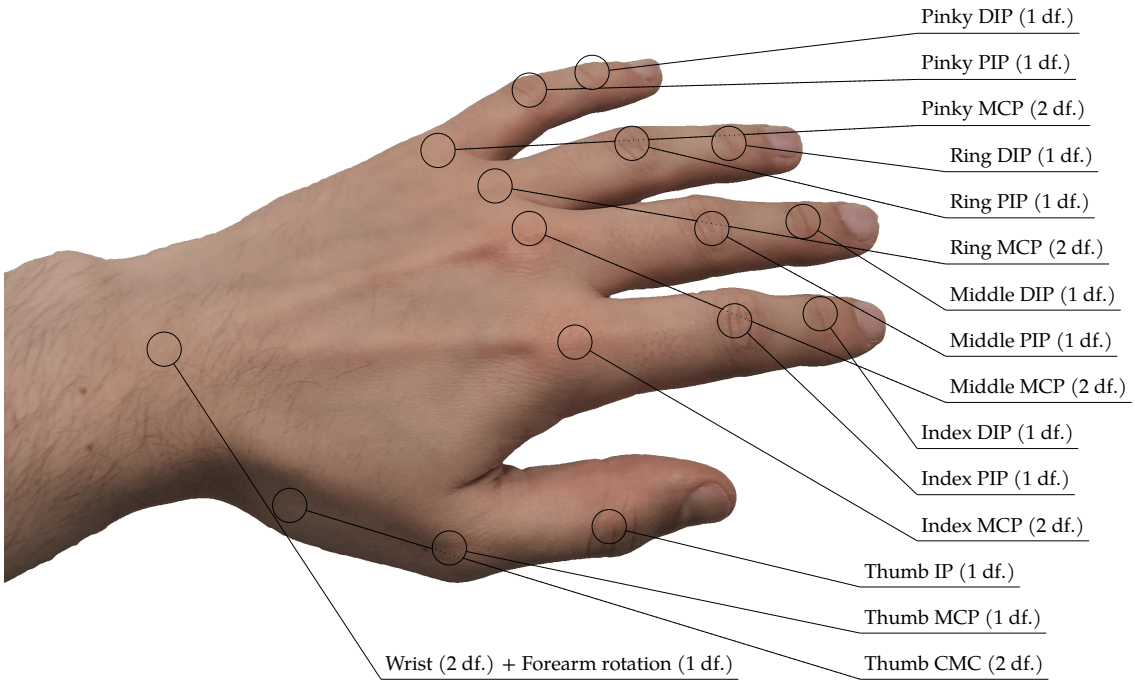

Figure 5.1: Each hand has 16 joints - three per digit, plus the wrist.

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.

23 degrees of freedom is only a first approximation. A fully-realistic hand model needs to account for more minor degrees of freedom, such as movement of the metacarpal bones, rotation of the digits around the longitudinal axis, and movement of the skin and muscles.

5.3.1 Heirarchy of Rotations

The most common parameterization of the hand is to use a heirarchy of rotations. Each joint is represented as a rotation (using one of the representations discussed in the previous section), along with an position offset, both relative to the previous joint. The magnitude of the offset typically remains fixed, because it represents the length of the bone. The wrist rotation is a rotation relative to the reference frame of the forearm (or the global reference frame, if only the hands are being modeled). The pinky MCP rotation is relative to the reference frame of the wrist, the PIP is relative to the MCP and so on. This means that rotating the wrist will change the position of all the joints in the hand, and rotating the MCP will change the position of all the joints below it, etc.

The benefit of this parameterization are that it matches the real kinematics of the hand – rotating the wrist does in fact change the position of all the joints in the hand. For this reason this parameterization is used for the experiments in Chapter 6.

5.3.2 Constraints

Rather than representing each joint with a full rotation, we can place constraints on the range of values that the rotation can take. For example, the

following constraints are used by [18]:

$$\begin{aligned} 0° \leq \theta_{\text{MCP-F}} \leq 90° \\ -15° \leq \theta_{\text{MCP-AA}} \leq 15° \\ 0° \leq \theta_{\text{PIP}} \leq 110° \\ 0° \leq \theta_{\text{DIP}} \leq 90° \end{aligned} \tag{5.11}$$

Whether or not to place constraints on the motion depends on the domain. For example, if we are modeling the hand for a virtual reality application, we might want to constrain the motion to be realistic. However, if we are training a neural network, then we cannot place non-differentiable constraints on the data, and it is more common to simply allow the network learn the constraints from the data.

5.3.3 Point Cloud Representation

For different tasks, different paramterizations might be appropriate. Take for example *pose estimation*, where the task is to find the position and rotation of the joints from an image or video. In this case, a point cloud representation is often used, where each joint is represented by a point in 3D space. The benefit of this representation is that it is very easy to estimate from an image or video, and it is also easy to interpolate between two configurations. To then use this representation for animation, it is then converted into the hierarchical representation.

5.4 Loss functions for learning angles

Chapter 2 introduced some loss functions such as the Mean Squared Error (MSE) (Section 2.4.1), which trains the model to output the posterior expectation $\mathbb{E}(y|x)$, and the Gaussian negative-log-likelihood which trains the model to output the posterior distribution $N(y|\mu, \sigma)$. This section will discuss some related loss functions that are used when data sits on a manifold, such as angles on the unit circle S^1, or rotations on S^3.

5.4.1 Arc distance loss

The most obvious loss function that we might use when training a model to output angles is the arc distance between two angles. This is the distance between the two angles on the unit circle, and is given by

$$\mathcal{L}_{\text{arc}}(y, \hat{y}) = \cos^{-1}(\cos(y - \hat{y})) \tag{5.12}$$

A graph of this loss function is shown in Figure 5.2. The loss is zero when the two angles are equal, and increases linearly as the angles move further apart. This loss function is not differentiable at $y = \hat{y} + \pi$, but this is not a problem because the model will never output this value.

5.4.2 Angular mean squared error loss

The analogous loss functions to the mean squared error for data that sits on the unit circle is the *angular* mean-squared-error, which will be used later

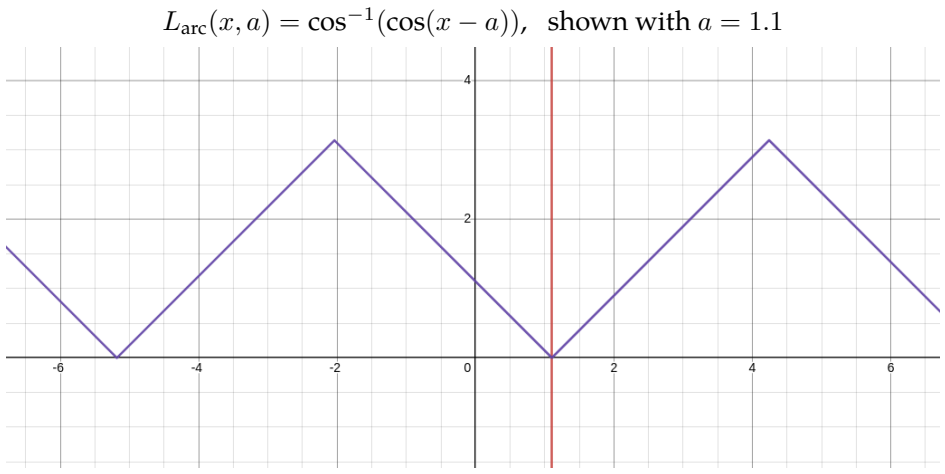

Figure 5.2: The arc distance loss function is the distance between two angles on the unit circle, analogous to the absolute distance for unbounded variables. It is zero when the two angles are equal, and increases linearly as the angles move further apart. The function is not differentiable at $y = \hat{y}$ and $y = \hat{y} + \pi$.

---

on in Chapter 6. It is defined as follows:

$$\begin{aligned} &\mathcal{L}_{\theta\text{-MSE}} \colon \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ &\mathcal{L}_{\theta\text{-MSE}}(Y, \hat{Y})_{ni} \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n,i} (\sin Y_{ni} - \sin \hat{Y}_{ni})^2 + (\cos Y_{ni} - \cos \hat{Y}_{ni})^2 \end{aligned} \tag{5.13}$$

where Y and $\hat{Y}$ are sequences of vectors of angles, with batch shape N and vector length D. We can see that the definition is similar to that of the circular mean in Section 5.1.1. A graph of the angular mean squared error loss is shown in Figure 5.3.

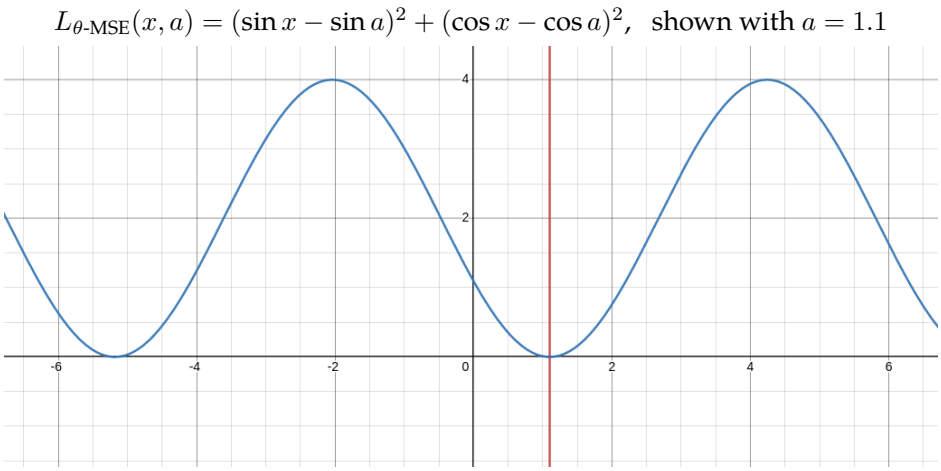

Figure 5.3: The angular mean squared error loss function is analogous to the squared error for unbounded variables. It is zero when the two angles are equal, and increases quadratically as the angles move further apart, reaching a maximum when the two angles are opposite. Unlike the arc distance loss, the angular mean squared error loss is differentiable everywhere.

5.4.3 von-Mises distribution

The von-Mises distribution (eg. [9]) is a distribution over angles on the unit circle. It is defined as follows:

$$\begin{aligned} &\mathrm{vM}\colon \mathbb{R}\times\mathbb{R}\to\mathbb{R} \\ &\mathrm{vM}(\mu,\kappa) = \frac{1}{2\pi I_0(\kappa)}\exp(\kappa\cos(\theta-\mu)) \end{aligned} \tag{5.14}$$

where μ is the modal angle and the circular mean of the distribution, and κ is a concentration parameter. The distribution is the uniform distribution on the circle when $\kappa = 0$, and becomes more concentrated as κ increases.

I_0 is the modified Bessel function of the first kind of order zero:

$$\begin{aligned} &I_0\colon \mathbb{R}\to\mathbb{R} \\ &I_0(x) \stackrel{\text{def}}{=} \int_{-\pi}^{\pi}\exp(x\cos t)\,dt \end{aligned} \tag{5.15}$$

5.4.4 Motivation for the Angular Mean Squared Error

The angular mean squared error defined above in Section 5.4.2 is not arbitrarily chosen. The motivation for it is described in this subsection. First the derivation of the mean-squared-error from the likelihood function of the Gaussian distribution is given, and then the derivation of the angular mean-squared-error from the likelihood function of the von-Mises distribution.

The Gaussian distribution and the von-Mises distribution are the *maximum-entropy* distributions of their respective domains. This means among all distributions over the real line $x \in (-\infty, \infty)$ with finite mean μ and variance σ^2, the Gaussian distribution $N(x \mid \mu, \sigma)$ is the one with the largest

entropy. Similarly, among all distributions over the unit circle $x \in (-\pi, pi]$ with circular mean μ and circular variance κ, the von-Mises distribution is the one with the largest entropy.

As a result of this fact, training a model which outputs only a mean estimate with a mean-squared-error loss is equivalent to maximizing the likelihood of the data under the Gaussian distribution.

The likelihood function of a Gaussian distribution is as follows, where the mean μ is the output of the model $f(x, \theta)$.

$$\begin{aligned} &\mathbb{L}_G : \mathbb{R}^P \to \mathbb{R} \\ &\mathbb{L}_G(\theta) \stackrel{\text{def}}{=} \prod_n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - f(x_n, \theta))^2\right) \end{aligned} \tag{5.16}$$

where x and y are the input and target data respectively, N is the number of data points, and P is the number of parameters. For simplicity this is defined for scalar data, but the definition can be extended to vector data.

The optimization of the parameters θ is represented with the arg max operator, which returns the value of θ that maximizes the likelihood of the data. Let x and y be the input and target data respectively, N be the number of data points, and P be the number of parameters.

The derivation is as follows:

$$
\begin{aligned}
&\arg\max_\theta \mathbb{L}_G(\theta) \\
&= \arg\max_\theta \prod_n \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2}(y_n - f(x_n,\theta))^2\right) \\
&= \arg\max_\theta \prod_n \exp\left(-\frac{1}{2\sigma^2}(y_n - f(x_n,\theta))^2\right) && \text{by monotonicity} \\
&= \arg\max_\theta \prod_n \exp\left(-(y_n - f(x_n,\theta))^2\right) && \text{by monotonicity} \\
&= \arg\max_\theta \sum_n -(y_n - f(x_n,\theta))^2 && \text{by monotonicity of ln} \\
&= \arg\min_\theta \sum_n (y_n - f(x_n,\theta))^2 \\
&= \arg\min_\theta \frac{1}{N}\sum_n (y_n - f(x_n,\theta))^2 && \text{by monotonicity} \\
&= \arg\min_\theta \mathcal{L}_{\text{MSE}}(y, f(x,\theta))
\end{aligned}
\tag{5.17}
$$

Similarly to the relationship between $\mathcal{L}_{\text{MSE}}$ and the Gaussian, minimizing the angular mean-squared-error loss is equivalent to maximizing the likelihood of a von-Mises distribution. The likelihood function of a von-Mises distribution is as follows, where the circular mean μ is the output of the model $f(x,\theta)$.

$$
\begin{aligned}
&\mathbb{L}_{\text{vM}} \colon \mathbb{R}^P \to \mathbb{R} \\
&\mathbb{L}_{\text{vM}}(\theta) \stackrel{\text{def}}{=} \prod_n \frac{1}{2\pi I_0(\kappa)} \exp\left(\kappa \cos(y_n - f(x_n,\theta))\right)
\end{aligned}
\tag{5.18}
$$

where x and y are the input and target data respectively, N is the number of data points, and P is the number of parameters. Again, for simplicity this is defined for scalar data, but the definition can be extended to vectors

of angles.

Maximizing the above likelihood function is equivalent to minimizing the angular mean-squared-error loss. The derivation is as follows:

$$
\begin{aligned}
&\arg\max_\theta \mathbb{L}_{\text{vM}}(\theta) \\
&= \arg\max_\theta \prod_n \frac{1}{2\pi I_0(\kappa)} \exp\left(\kappa \cos(y_n - f(x_n, \theta))\right) \\
&= \arg\max_\theta \prod_n \exp\left(\kappa \cos(y_n - f(x_n, \theta))\right) && \text{by monotonicity} \\
&= \arg\max_\theta \prod_n \exp\left(\cos(y_n - f(x_n, \theta))\right) && \text{by monotonicity} \\
&= \arg\max_\theta \sum_n \cos(y_n - f(x_n, \theta)) && \text{by monotonicity of ln} \\
&= \arg\min_\theta \sum_n -\cos(y_n - f(x_n, \theta)) \\
&= \arg\min_\theta \sum_n -\cos(y_n - f(x_n, \theta)) + 1 + 1 \\
&= \arg\min_\theta \sum_n -\cos(y_n - f(x_n, \theta)) + (\sin^2 y_n + \cos^2 y_n) \\
&\quad + (\sin^2 f(x_n, \theta) + \cos^2 f(x_n, \theta)) && \text{by } \sin^2 a + \cos^2 a = 1 \\
&= \arg\min_\theta \sum_n -(\sin y_n \sin f(x_n, \theta) + \cos y_n \cos f(x_n, \theta)) \\
&\quad + (\sin^2 y_n + \cos^2 y_n) + (\sin^2 f(x_n, \theta) + \cos^2 f(x_n, \theta)) && \begin{array}{l}\text{by } \cos(a \pm b) \\ \quad = \cos a \cos b \mp \sin a \sin b\end{array} \\
&= \arg\min_\theta \sum_n (\sin^2 y_n - \sin y_n \sin f(x_n, \theta) + \sin^2 f(x_n, \theta)) \\
&\quad + (\cos^2 y_n - \cos y_n \cos f(x_n, \theta) + \cos^2 f(x_n, \theta)) \\
&= \arg\min_\theta \sum_n (\sin^2 y_n + 2 \sin y_n \sin f(x_n, \theta) + \sin^2 f(x_n, \theta)) \\
&\quad + (\cos^2 y_n + 2 \cos y_n \cos f(x_n, \theta) + 2 \cos^2 f(x_n, \theta)) && \text{by monotonicity}
\end{aligned}
$$

$$
\begin{aligned}
&= \arg\min_{\theta} \sum_{n} (\sin^2 y_n - 2 \sin y_n \sin f(x_n, \theta) + \sin^2 f(x_n, \theta)) \\
&\quad + (\cos^2 y_n - 2 \cos y_n \cos f(x_n, \theta) + \cos^2 f(x_n, \theta)) \\
&= \arg\min_{\theta} \sum_{n} (\sin y_n - \sin f(x_n, \theta))^2 \\
&\quad + (\cos y_n - \cos f(x_n, \theta))^2 \\
&= \arg\min_{\theta} \mathcal{L}_{\theta\text{-MSE}}(y_n, \sin f(x_n, \theta))
\end{aligned} \tag{5.19}
$$

Chapter 6

Hand Motion Models

This chapter puts together the learnings from the previous chapters to develop neural network-based models for hand motion modeling.

6.1 Introduction

Hand motion modeling is a task which involves predicting hand poses (ie. configurations of joints, as described in Section 5.3), across many timesteps in an animation. This is a challenging task because of the large number of degrees of freedom in the hand, the large number of timesteps in an animation, and the fact that the hand is a complex structure where small errors are easily noticable.

This chapter presents two models which are able to predict hand poses. They are both trained on a dataset of 3D hand animations produced via motion capture, and are able to predict hand poses in a variety of different poses. The first model is trained with a regression objective, and is able to predict whole hand poses. The second model is trained with a parameter

estimation objective, and is able to predict the parameters of a von-Mises distribution over individual joints in the hand. This model is then auto-regressively called to predict the whole hand pose.

6.2 Previous Work

To my knowledge this work is the first to target this exact task, but there are two closely related areas of previous work which are relevant to this chapter. The first is the use of neural networks for pose estimation, and the second is the use of neural networks for human motion modeling.

6.2.1 Pose Estimation

Pose estimation (eg. [26]) is the task of estimating the pose/configuration of a character or hand from image or video data. Almost all the work in the literature related to hand motion is some variant of hand pose estimation task – this is because it is directly applicable to virtual reality (VR) applications. The task of hand motion modeling is often downstream of hand pose estimation – the output of a hand pose estimation model is often used as input to a hand motion model. This is the case for the ManipNet dataset [33], which is used in this thesis.

An example of hand pose estimation is shown in Figure 6.1. The input to the model is a single image of a hand, and the output is a 3D pose of the hand. The model is able to predict the pose of the hand in a variety of different poses, including poses which are not in the training data.

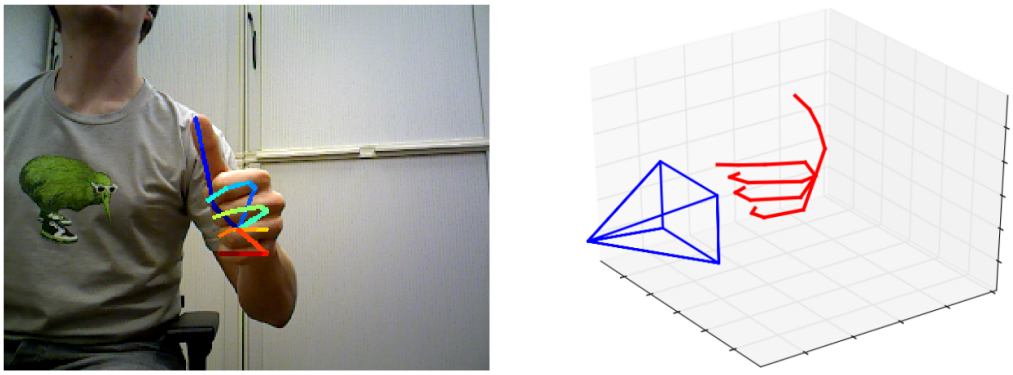

Figure 6.1: An example of the hand pose estimation task, from [6]. Left: The input to the model is an image or video of a hand. Right: The output is a 3D pose of a hand.

6.2.2 Human Motion Modeling

Human motion modeling is the task of predicting human motion from past motion data. We can divide this into three rough categories, as per [30]:

1. Human motion prediction, which is the task of predicting future human motion given an observed sequence of human pose.

2. Human motion control, which is the task of producing a sequence of realistic human motion given some goal, eg. a target pose, or a target trajectory.

3. Cross-modal motion synthesis, which is the task of producing a sequence of human motion given some other input, eg. a sequence of images, or a sequence of audio. This overlaps with pose estimation, because the input to the model is often video data.

Most of the literature in this area is focused on character motion, ie. predicting the pose of the body and limbs, rather than the hands or facial

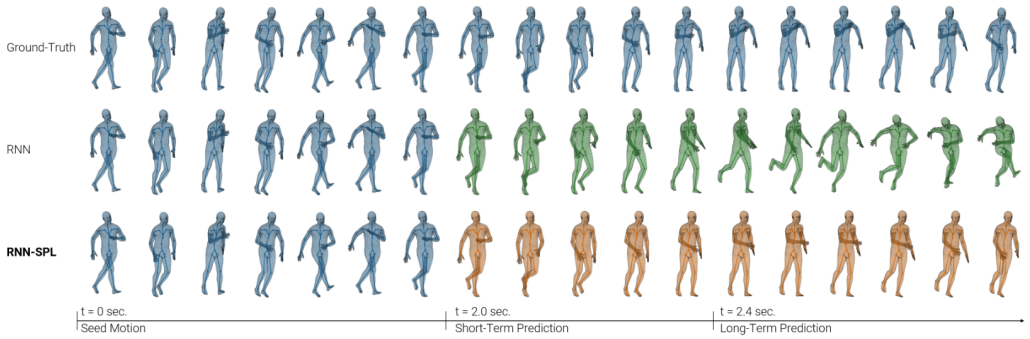

Figure 6.2: An example of the character motion modeling task, from [1]. Top: Ground truth motion sequence, typically recorded via motion capture. Middle and bottom: Two different completions of the motion sequence, produced by a model.

expression. For example, there are a number of recent papers on modeling choreographed dancing, which treats character motion as a sequence-to-sequence task, generating a sequence of poses conditional on a music track [31]. Recent works here use encoder-decoder transformers for this purpose, for example [27].

Of the many related works, the most relevant ones are the following:

- ManipNet [33], whose high-fidelity motion capture dataset is used in this chapter, although their task is very different. Their work is unique and focuses on predicting the pose of the fingers based on the pose of only the wrist and the pose and geometry of a held object.

- [1], who focus on predicting character motion. The task in this chapter is identical to their character motion modeling task shown in Figure 6.2, except that of course they work with character poses rather than hand poses.

- SignBERT [14], which focuses on training a "language model" for sign language, using hand pose sequences extracted from video datasets of sign language using hand pose estimation. The work in this chapter is very similar to this, except that SignBERT uses an encoder-only transformer and a masked sequence modeling pretraining task, whereas the work in this chapter uses a decoder-only model and an autoregressive sequence modeling pretraining task. Additionally, the dataset used in this chapter is captured with a high-fidelity motion capture system, rather than derived from video data.

6.3 Dataset

The dataset used for the experiments in this chapter is the ManipNet [33] dataset. This dataset was captured using a *deep label* system [11], and an *Optitrack* [21] motion capture system with 16 cameras at 60 frames per second, and gloves with 19 markers per hand for finger tracking.

The dataset contains 62 animation sequences, each of which involves a participant (whose hand and joint position is being recorded via a motion capture setup) manipulating a rigid object such as a block, teapot or mug. Each animation sequence contains somewhere between 4000 frames (2m 20s at 30 frames/s) and 8000 frames (4m 40s) of data.

The data was split into a training set, validation set, and test set. The lengths of these splits along with some other statistics about the dataset can be seen in Table 6.1. The test set was used for producing visualizations and for the final evaluation of the models, and the validation set was used to tune hyperparameters and for determining early stopping point when

	Total	Train	Val	Test
N° frames	467372	375372	44000	48000
N° examples	62	50	6	6
Length	4h 19m	3h 28m	24m	26m
Mean $\mathcal{L}_{\theta\text{-MSE}}$		0.0737	0.0555	0.0818

Table 6.1: Summary statistics for the ManipNet [33] dataset used in the experiments. The $\mathcal{L}_{\theta\text{-MSE}}$ statistic shows the error on the respective dataset if a model simply predicts the *circular mean* (Equation (5.2)) of all the frames in the training set. This represents an upper bound (and sanity check) on the error for the experiments.

training.

The data is represented in the BVH (BioVision Hierarchy) format [**bvh**]. A BVH file is a text file which contains the definition of a hierarchy of joints, including the offset and rest pose, followed by data for each frame of the animation. Each ManipNet BVH file contains 51 floating point values per frame (14 joints $\times$ 3 values per joint). The two hands are specified in two different files.

6.4 Training the deterministic models

The first models are deterministic models that were trained with a regression loss, to predict a whole hand pose at each timestep.

6.4.1 Tasks

Two models were trained with identical hyperparameters on two tasks. Both models were trained to minimize the angular mean squared error

$\mathcal{L}_{\theta\text{-MSE}}$ (Equation (5.13)) between the predicted pose and the ground truth pose. As a result of using this training objective, the outputs of the models estimate the expectation over the posterior distribution of the hand pose at the next frame, given the previous frames.

The first task was a plain auto-regressive pre-training task as in Section 3.3.2. Specifically, the task was to predict the pose of the hand at the next frame, given some number of previous frames.

The second task was as above but with the addition of a goal condition. Specifically, the task was to predict the pose of the hand at the next frame, given some number of previous frames and a goal pose. The goal pose was the pose of the hand at the last frame of the sequence, and was prepended to the input sequence of every training example.

The data was provided to the model as a sequence of 204-dimensional vectors (2 hands $\times$ 17 joints $\times$ 3 unit vectors per joint $\times$ 2 components per unit vector), and the model was trained to predict the outputs of the subsequent timestep (another 204-dimensional vector).

The details of the training and hyperparameters are shown in Table 6.2.

6.4.2 Results

The results are presented in two formats: image visualizations in Figure 6.3, and a video visualization on YouTube (see Figure 6.4). Both visualizations show the model's predictions at each timestep, and the ground truth at each timestep.

N° input dims	204
N° output dims	204
N° embedding dims	256
N° layers	24
N° hidden dims	512
N° attention heads	8
Total N° parameters	56,977,304
Batch size	16
Chunk length	512
GPU	NVIDIA RTX 3080
Training steps	20,000
Task 1	
Training time	≈19m
Training $\mathcal{L}_{\theta\text{-MSE}}$	0.000456
Validation $\mathcal{L}_{\theta\text{-MSE}}$	0.0448
Task 2	
Training time	≈19m
Training $\mathcal{L}_{\theta\text{-MSE}}$	0.000447
Validation $\mathcal{L}_{\theta\text{-MSE}}$	0.0480

Table 6.2: Hyperparameters and training configuration for the deterministic models.

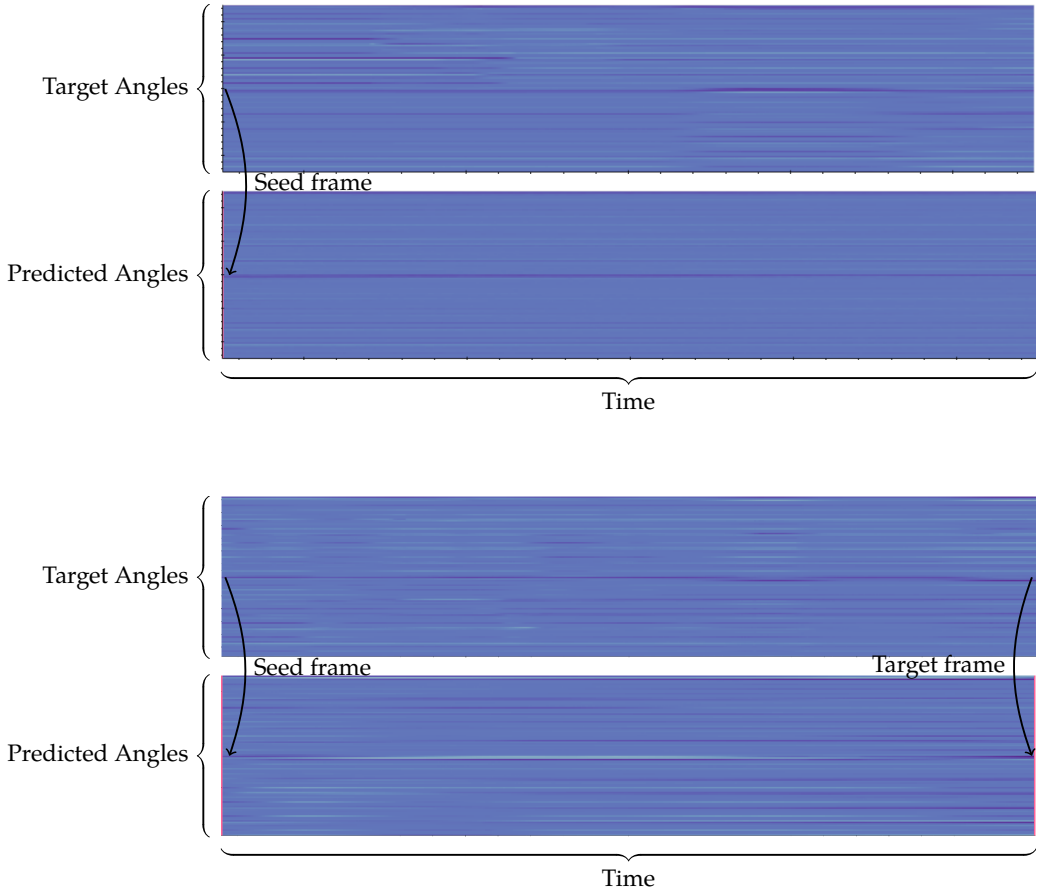

Figure 6.3: Image visualizations of outputs from the deterministic model. Top: Predictions from the untargeted model. Bottom: Predictions from the model trained on the target pose problem.

The above 102px by 512px images show two pairs of predicted animation sequences. The top image of each pair shows the ground truth data, and the bottom image shows the model's predictions.

The predictions were produced by seeding the model with either the first frame of the ground truth data then auto-regressively predicting the hand poses at each timestep in between. In the bottom figure, the model is also provided the final frame of data as a target. We can see that in both cases the model initially correctly predicts poses close to the ground truth, but it then diverges.

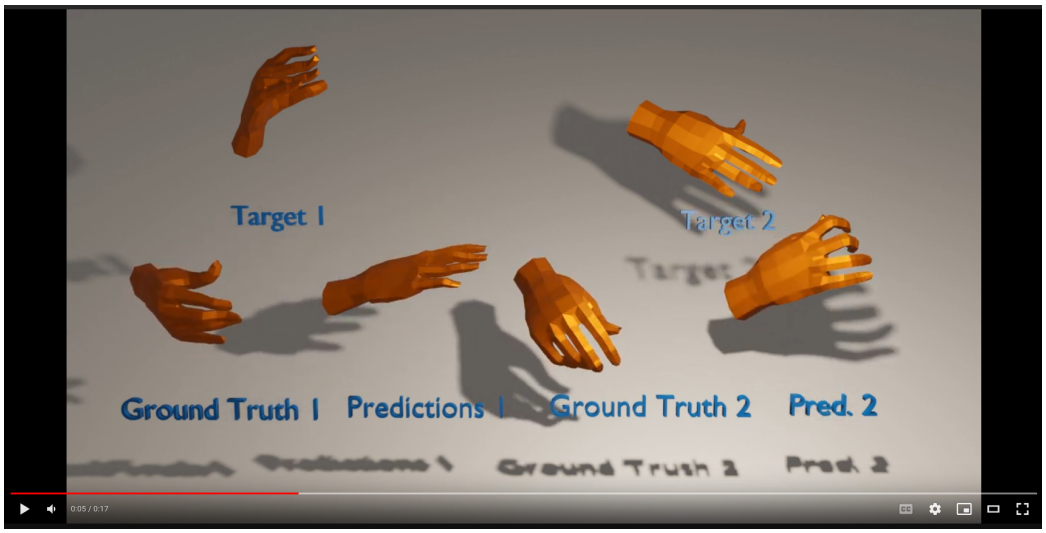

Figure 6.4: Video visualizations of outputs from the deterministic model (On the targeted task only). A video showing the results as rendered animated hands can be found here. The video shows two different ground truth seed sequences, and the model's predictions for each.

As we can see in the figures and video, both models produce relatively smooth and realistic predictions. However, the model trained on the targeted task does not achieve the objective. Despite the target pose being provided during both training and the prediction process, the predictions do not re-converge to the target pose at the end of the predicted sequence. This may be because the model has not learned to use the target pose, which might happen if the model has over-fitted to the training data. Over-fitting is also suggested by the difference between the training and validation $\mathcal{L}_{\theta\text{-MSE}}$ values (Table 6.2). The training loss is significantly lower than the validation loss.

6.5 Training a probabilistic model

The second model was trained with a parameter estimation objective, to predict the parameters of a von-Mises distribution over individual joints in the hand.

6.5.1 Task

Unlike the previous model where the data was provided as sequences of whole frames, for the probabilistic model the data was provided as a sequence of single 2D unit vectors. This is neccessary in order to model the joint probability distribution of a hand pose. Each unit vector represents a single component angle of a single joint in the hand. The model was then trained to predict the parameters of a von-Mises distribution over the next component angle, using the von-Mises negative-log-likelihood as the loss

function. The model is called 51 times to produce all of the component angles for one frame of the animation, then this is repeated for each frame in the sequence.

Because the data was provided as a sequence of single components rather than full pose vectors, the probabilistic model requires a much longer sequence length to condition the model on the same amount of data from the animation. Because the transformer attention matrices scale in memory complexity with the sequence length, training the probabilistic model was limited by the available GPU memory. In order to keep a long context window and still fit into the GPU memory during training, the model is therefore smaller than the deterministic model.

6.5.2 Results

As before, the results are presented as both image visualization in Figure 6.5, and video visualization in Figure 6.6. Both show a comparison between the model predictions and the ground truth data.

We can see from the figures and video that the probabilistic model unfortunately does not perform well. There are two main problems:

1. The sampling process is unstable, and the fingers rapidly converge into an unusual part of the hand pose space.

2. The model predicts distributions with high variance, resulting in flickering / noise when sampling. This is especially noticeable in the wrist, where it causes the whole hand to shift.

N° input dims	2
N° output dims	2
N° embedding dims	510
N° layers	3
N° hidden dims	1048
N° attention heads	8
Total N° parameters	23,710,750
Batch size	16
Chunk length	1530 (30 frames $\times$ 51 components)
Training time	12m 9s
Training steps	10,000

Table 6.3: Hyperparameters and training configuration for the probabilistic model.

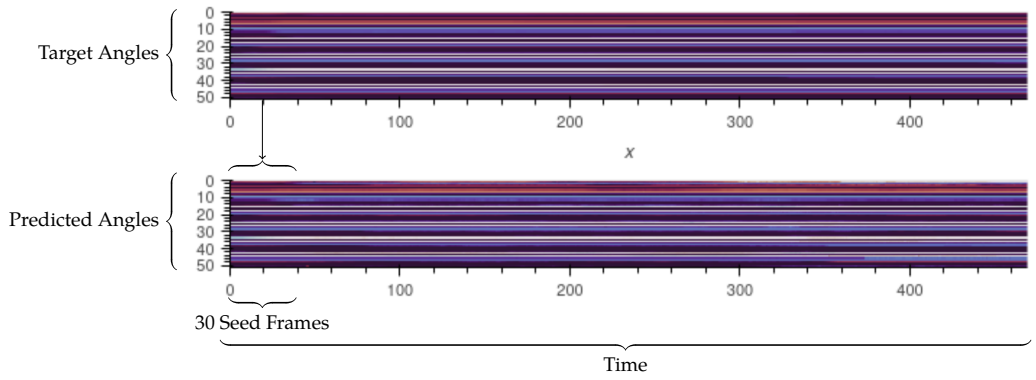

Figure 6.5: Image visualizations of outputs from the probabilistic model. The above 51px by 450px images show two animation sequences. The top image shows ground truth data, and the bottom image shows the model's predictions. The first 30 frames of the predicted sequence are copied from the ground truth as seed data, and the remaining frames are predicted by the model.

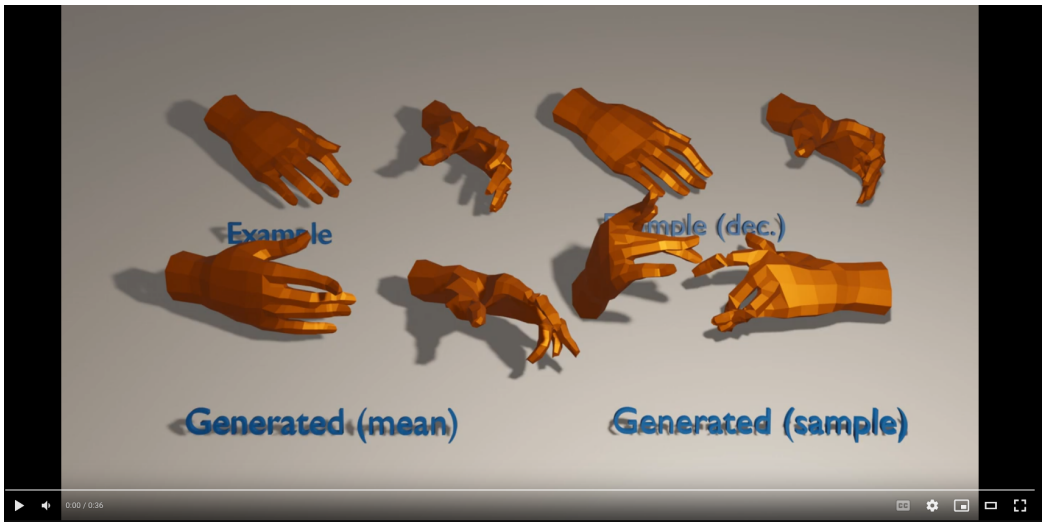

Figure 6.6: Video visualizations of outputs from the probabilistic model. A video showing the results as rendered animated hands can be found here. The video shows two different ground truth seed sequences, and the model's predictions for each.

6.6 Discussion

The deterministic model performs better than the probabilistic model – it is able to produce smooth animations, while the probabilistic model produces flickering/noise and also degrades as the sampling process progresses.

The difference in performance may be due to the difference in difficulty on the respective problems. Because the probabilistic model predicts individual component angles, it effectively has a $100\times$ longer sequence leng. In addition it is trained with a parameter estimation objective, rather than a reconstruction objective, which may be more difficult to optimize - in particular the κ (concentration) paramter of the von-Mises distribution.

To determine whether the difference in performance is due to the model architecture, or the training objective, a third model could be trained with

the $\mathcal{L}_{\theta\text{-MSE}}$ loss, but on the sequence-of-single-components task. However, due to time constraints this was not done.

Additionally, it was only recently that the smooth results of the deterministic model were achieved, and due to time constraints the probabilistic model was not retrained to match the deterministic model's hyperparameters. As a result, it is possible that the probabilistic model could be yet improved to be competitive by retraining it with similar hyperparameters to the deterministic model.

6.7 Conclusions and Future Work

This chapter has presented two classes of transformer models on hand motion modeling tasks.

The first models are trained with a regression objective, and are able to produce smooth animations, but are not able to be conditioned on a target pose, possibly due to overfitting. Future work would be to investigate training smaller models and regularization techniques to prevent overfitting (and so determine whether this is the issue). If so, then additional datasets (such as the sign language dataset from [14]) could be used to increase the model's generalization ability.

The second model is trained to predict the parameters of a von-Mises distribution over individual component angles of a hand pose. Unfortunately, the model performs poorly and the conclusion is inconclusive. Future work here would include:

1. Further hyperparameter tuning.

2. Training a deterministic model on the sequence-of-single-components task, to determine whether the difference in performance is due to the different task, or the different training objective.

Chapter 7

Conclusions

To conclude, the main conclusions from the experiments in Chapter 4 and Chapter 6 are summarized, along with some reflections on the project as a whole.

7.1 Conclusions

Chapter 4 trained a probabilistic model on sequences of pixels in MNIST, and compared the effect of sampling pixels in a variety of orders. The results of this experiment was the discovery that using "highest-entropy-first" and "lowest-entropy-first" as heuristics for determining the sampling orders resulted in biased samples, of worse visual quality than using a "random" sampling order. This is most likely due to imperfections in the model being exacerbated during the sampling process by the entropy hueristic, but not by the random sampling order.

Chapter 6 trained both a deterministic model and a probabilistic model on sequences of hand poses from the ManipNet dataset. The determinis-

tic model was able to generate sequences of hand poses that were visually similar to the training data, but the probabilistic model performed very poorly. The conclusion as to why is unclear however, since due to time constraints the experiments needed to investigate why have not been performed. Further investigation is required.

7.2 Reflections

If I had known what I know now at the start of my project, what would I have done differently?

First, training neural networks can be very finnicky. Instead of trying new architectures and model types, I later found that hyperparameters such as the weight initialization, learning rate, and regularization loss terms have a much bigger impact, including on whether the network learns anything reasonable at all. I would have spent more time tuning these hyperparameters, and less time tuning the number of layers, number of neurons, activation functions, and other architecture choices. Relatedly, during the middle of my project, I was working with a custom implementation of a transformer, which was learning poorly. If I was experimenting primarily with these different kinds of hyperparameters, I would have kept using the standard transformer implementation, which would have saved me significant implemenation and debugging time.

Second, training with Masked Sequence Modeling is sufficient to represent the task I was trying to achieve in Chapter 4, but I didn't know this at the outset. I could have used a mostly-pre-implemented data preprocessing pipeline, again saving me significant implementation and de-

bugging time.

Third, running comparison experiments was forever a weak point for me. Doing this project again, I would have spent the additional effort to keep every version of my experiments working in the same codebase simultaneously, so that I could easily switch between them and compare results. Rather than modifying the implementation to run a new experiment (*even if* the current model & hyperparameters were currently broken), I would have added additional flags and configurations for every experiment, and added some unit tests to make sure that the code still works when these flags are changed. This would have meant I could have produced more meaningful results in Chapter 6.

7.3 Final words

I have enjoyed working on this thesis, more than I expected at the outset. I have found a passion for learning about deep learning. The project has been challenging but has given me the opportunity to learn an immense amount about machine learning techniques, including modern methods such as transformers. Applying this knowledge to hand motion modeling has also been a challenge of surprising depth. I hope that this thesis has been a useful contribution to the field of hand motion modeling, and that it might be useful to others.

Bibliography

[1] Emre Aksan, Manuel Kaufmann, and Otmar Hilliges. "Structured prediction helps 3d human motion modelling". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision*. 2019, pp. 7144–7153. URL: https://arxiv.org/abs/1910.09070.

[2] Alexei Baevski et al. "wav2vec 2.0: A Framework for Self-Supervised Learning of Speech Representations". In: *CoRR* abs/2006.11477 (2020). arXiv: 2006.11477. URL: https://arxiv.org/abs/2006.11477.

[3] Tom B. Brown et al. "Language Models are Few-Shot Learners". In: *CoRR* abs/2005.14165 (2020). arXiv: 2005.14165. URL: https://arxiv.org/abs/2005.14165.

[4] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: 2018. URL: https://arxiv.org/abs/1810.04805.

[5] Li Dong et al. "Unified language model pre-training for natural language understanding and generation". In: *Advances in Neural Information Processing Systems* 32 (2019). arXiv: 1905.03197. URL: http://arxiv.org/abs/1905.03197.

[6] Bardia Doosti. "Hand pose estimation: A survey". In: *arXiv preprint arXiv:1903.01013* (2019). URL: https://arxiv.org/abs/1903.01013.

[7] Patrick Esser, Robin Rombach, and Bjorn Ommer. "Taming transformers for high-resolution image synthesis". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2021, pp. 12873–12883. URL: http://openaccess.thecvf.com/content/CVPR2021/papers/Esser_Taming_Transformers_for_High-Resolution_Image_Synthesis_CVPR_2021_paper.pdf.

[8] Catherine Forbes et al. "Statistical Distributions, 4th Edition". In: Wiley, 2010, p. 143. URL: http://personalpages.to.infn.it/~zaninett/pdf/statistical-distributions.pdf.

[9] Catherine Forbes et al. "Statistical Distributions, 4th Edition". In: Wiley, 2010, p. 191. URL: http://personalpages.to.infn.it/~zaninett/pdf/statistical-distributions.pdf.

[10] Marta Garnelo et al. "Neural processes". In: *arXiv preprint arXiv:1807.01622* (2018). URL: https://arxiv.org/abs/1807.01622.

[11] Shangchen Han et al. "Online optical marker-based hand tracking with deep labels". In: *ACM Transactions on Graphics (TOG)* 37.4 (2018), pp. 1–10.

[12] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[13] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern*

recognition. 2016, pp. 770–778. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[14] Hezhen Hu et al. "SignBERT: Pre-Training of Hand-Model-Aware Representation for Sign Language Recognition". In: *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*. Oct. 2021, pp. 11087–11096. URL: https://openaccess.thecvf.com/content/ICCV2021/papers/Hu_SignBERT_Pre-Training_of_Hand-Model-Aware_Representation_for_Sign_Language_Recognition_ICCV_2021_paper.pdf.

[15] Hyunjik Kim et al. "Attentive neural processes". In: *arXiv preprint arXiv:1901.05761* (2019). URL: https://arxiv.org/abs/1901.05761.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Communications of the ACM* 60.6 (2017), pp. 84–90.

[17] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 2020, pp. 7871–7880. arXiv: 1910.13461. URL: http://arxiv.org/abs/1910.13461.

[18] John Lin, Ying Wu, and Thomas S Huang. "Modeling the constraints of human hand motion". In: *Proceedings workshop on human motion*. IEEE. 2000, pp. 121–126. URL: http://www.ifp.illinois.edu/~yingwu/papers/Humo00.pdf.

[19] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[20] Samuel Müller et al. "Transformers Can Do Bayesian Inference". In: *arXiv preprint arXiv:2112.10510* (2021).

[21] Inc. NaturalPoint. *OptiTrack*. Website. 2022. URL: https://optitrack.com/.

[22] Tung Nguyen and Aditya Grover. "Transformer neural processes: Uncertainty-aware meta learning via sequence modeling". In: *arXiv preprint arXiv:2207.04179* (2022). URL: https://arxiv.org/abs/2207.04179.

[23] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019). URL: https://github.com/openai/gpt-2.

[24] Colin Raffel et al. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer". In: *Journal of Machine Learning Research* 21.140 (2020), pp. 1–67. URL: http://jmlr.org/papers/v21/20-074.html.

[25] Noam Shazeer. "Fast transformer decoding: One write-head is all you need". In: *arXiv preprint arXiv:1911.02150* (2019). arXiv: 1911.02150. URL: http://arxiv.org/abs/1911.02150.

[26] Anastasia Tkach. *Real-Time Generative Hand Modeling and Tracking*. Tech. rep. EPFL, 2018. URL: https://infoscience.epfl.ch/record/256674/files/EPFL_TH8573.pdf.

[27] Guillermo Valle-Pérez et al. "Transflower: probabilistic autoregressive dance generation with multimodal attention". In: *ACM Transactions on Graphics (TOG)* 40.6 (2021), pp. 1–14. URL: https://dl.acm.org/doi/pdf/10.1145/3478513.3480570.

[28] 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.

[29] 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.

[30] Zijie Ye, Haozhe Wu, and Jia Jia. "Human motion modeling with deep learning: A survey". In: *AI Open* (2021). URL: https://www.sciencedirect.com/science/article/pii/S2666651021000309/pdfft.

[31] Zijie Ye et al. "Choreonet: Towards music to dance synthesis with choreographic action unit". In: *Proceedings of the 28th ACM International Conference on Multimedia*. 2020, pp. 744–752. URL: https://dl.acm.org/doi/pdf/10.1145/3394171.3414005.

[32] Jiahui Yu et al. *Scaling Autoregressive Models for Content-Rich Text-to-Image Generation*. 2022. DOI: 10.48550/ARXIV.2206.10789. URL: https://arxiv.org/abs/2206.10789.

[33] He Zhang et al. "ManipNet: Neural manipulation synthesis with a hand-object spatial representation". In: *ACM Transactions on Graphics (ToG)* 40.4 (2021), pp. 1–14. URL: https://www.ipab.inf.ed.ac.uk/cgvu/zhang2021.pdf.