Conditional Generative Transformers for Hand-Guided Animation Automation

by

Maxwell Clarke

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

Master of Science
in Computer Science.

Victoria University of Wellington

November 2022

Abstract

In this thesis I seek to understand transformer models, explore new ways of using them, and then apply them to the domain of hand motion modeling.

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

Secondly, I explore the use of transformer models for *arbitrary-order* sampling. I show how we can construct and train a model which can be used to predict sparse sequence data, and demonstrate this on the MNIST dataset. I then compare different sampling orders, including some heuristic-based *dynamic* sampling orders. I find that these dynamic sampling orders introduce a statistical bias into the samples.

Thirdly, I introduce the problem domain of hand motion and hand animation, and discuss various ways to parameterize configurations of hands, and probability distributions over those configurations.

Lastly, I develop a predictive model for hand-motion data, via self-supervised learning on a motion-capture dataset, and present the results of using this model to generate hand-motion sequences. I find that the learned model is able to generate realistic-looking hand motions, but is unable to be directed to generate specific motions.

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 the research direction. I would like to thank the effective altruism community for motivating me to pursue deep learning – I hope I can contribute to the project of AI alignment in future. Lastly, I would like to thank my friends and family for their support and encouragement, and especially my girlfriend Kibra for her patience and understanding.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis introduces concepts and experiments at the intersection of two areas: Deep Learning and Computer Graphics. There are three main areas of focus: Comprehensively introducing a class of neural network models called *transformer* networks; an experiment comparing different sampling orders when predicting data using*auto-regressive* transformer models; and the developement of an application of auto-regressive transformer models for *hand pose modeling*.

1.1 Contributions

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

1. I present experiments with *dynamically-ordered* auto-regressive sampling, utilising the *permutation-invariance* property of the attention operation in transformer models.

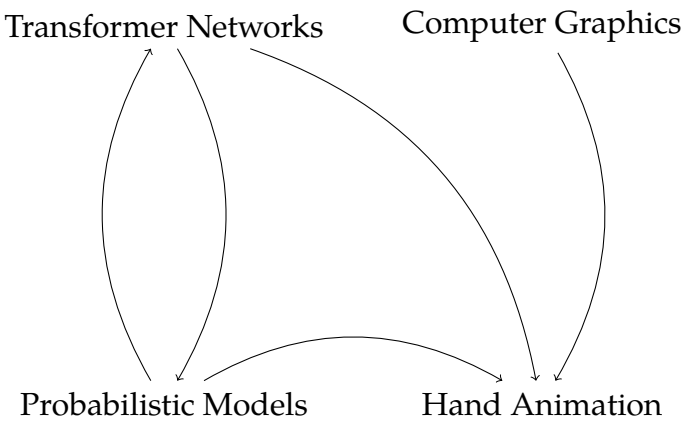

Figure 1.1: How learnings and experiments from different fields contribute to each other in this thesis.

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

These contributions involved training neural networks (see Figure 1.2), in particular transformers, on two datasets - MNIST, and a motion capture dataset of hand motion. The results of these experiments are presented in Chapter 4 and Chapter 6 respectively.

1.2 Structure

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

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

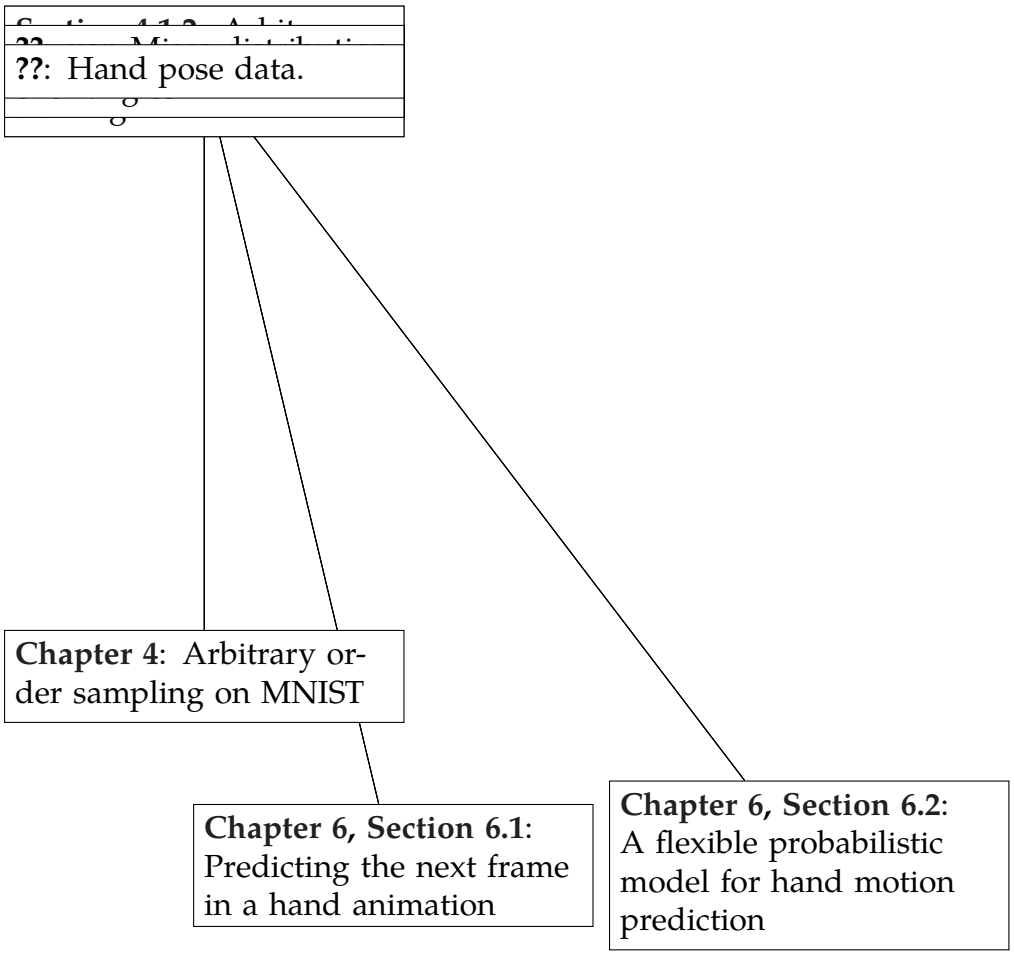

Figure 1.2: The later work in this thesis sits focuses on learning un-labeled sequence data with transformers, in two different domains.

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

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

2. Chapter 3 provides an introduction and literature reivew 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 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 thesis and discusses future work and reflections.

Chapter 2

Neural Networks and Deep Learning

Since Krizhevsky et al.'s AlexNet [**alexnet**] in 2012, many problems are increasingly being solved best by Artificial Neural Networks (NN), including image classifcation, translation of natural language, speech encoding, driving 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.

Many variants of NN exist. In this chapter, the notation used throughout this thesis will first be introduced and clarified, and then an overview of the different aspects of neural networks will be given, to contextualize the work in the following chapters.

2.1 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. The following notation is used throughout this thesis.

First, a simple example using activation functions.

2.1.1 Tensor index notation

The ReLU activation function is defined as:

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

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 wiil be throughout this thesis) 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 exam-

ple, 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:

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

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

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

The 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 [7] or TensorFlow [9]. Here however the order of the subscript indices b and n is ignored – the indices index their respectively-named dimensions. This that we abandon the distinction between row- and column-vectors. I will thus be explicit when applying matrix and vector products.

2.1.2 Neural networks

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

I will typically use N for the input dimensionality of a network, and D for the *embedding* (or *hidden* / *latent*) dimensionality. Let $\boldsymbol{x} \in \mathbb{R}^N$ be some input data embedded into an N-dimensional vector space. Let $W \in$

$\mathbb{R}^{N\times D}$ be a matrix of learned weights, and let $\phi\colon \mathbb{R} \to \mathbb{R}$ be some non-linear function. Then, the computation done by one layer of a simple fully-connected neural network is represented as follows.

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

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

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 dimen-

sional data classified into C classes, with L hidden layers:

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

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 I will discuss in the next section.

2.2 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 out-

put 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 timeseries 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. In this section, I will help to contextualize my later work by giving a brief overview of the ways that different neural networks and training setups differ.

On the following pages, I give 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)

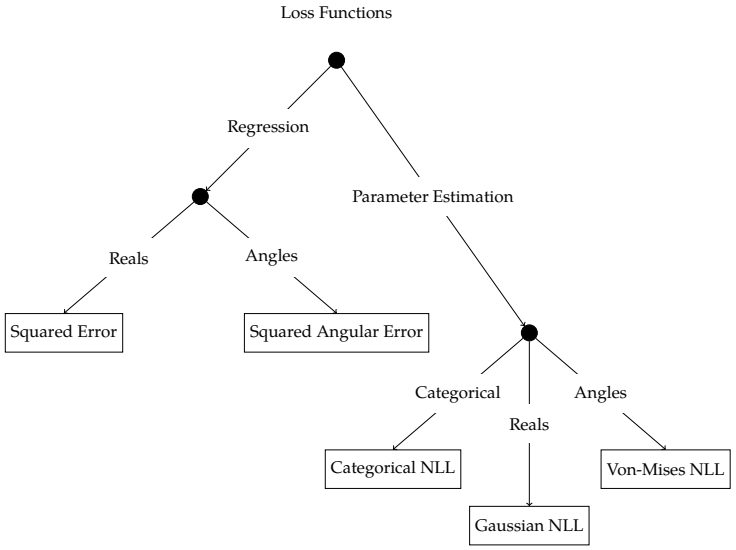

Figure 2.1: We can split loss functions into two categories – regression losses and parameter estimation losses.

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.

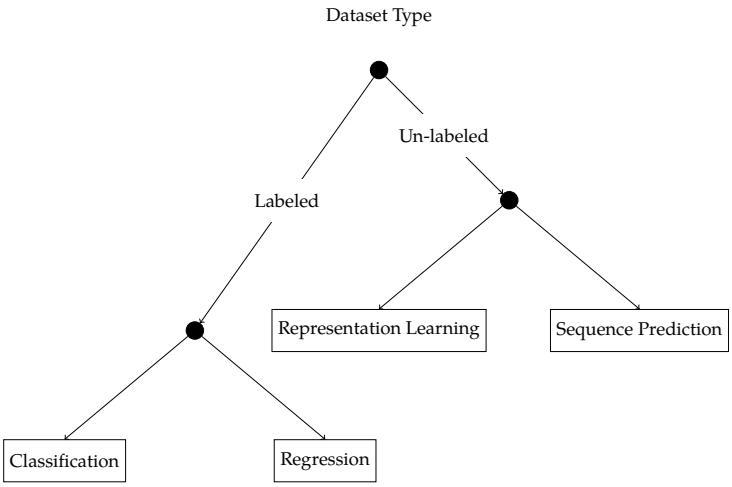

Figure 2.2: Basic ontology of dataset types.

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.

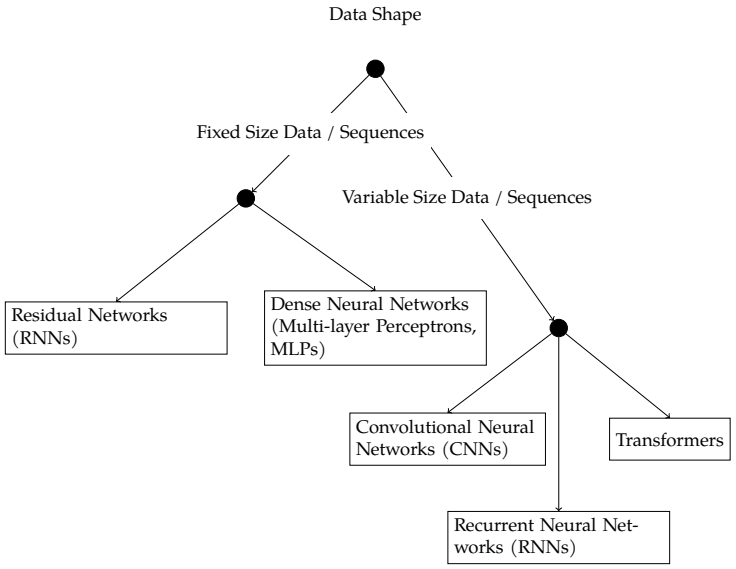

Figure 2.3: Neural network variants which support variable input shape.

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.

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

$$L_{\text{MSE}} : \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \tag{2.5}$$

$$L_{\text{MSE}}(y, \hat{y})_{ni} \stackrel{\text{def}}{=} \frac{1}{N} \sum_{n} \left[\sum_{i} (y_{ni} - \hat{y}_{ni})^2 \right] \tag{2.6}$$

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

2.3 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.

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

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 correspondes 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.

2.3.1 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

$$p_{\text{Cat}} \colon \mathbb{R}^C \to \mathbb{R} \tag{2.7}$$

$$p_{\text{Cat}}(x = i \mid \theta) \stackrel{\text{def}}{=} \frac{e^{\theta_i}}{\sum_j \exp(\theta_j)} \tag{2.8}$$

$$L_{\text{Cat}} \colon \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \tag{2.9}$$

$$L_{\text{Cat}}(y, \hat{\theta}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_n - \log p_{\text{Cat}}(y_n \mid \hat{\theta}_n) \tag{2.10}$$

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.

2.3.2 Gaussian distribution

Another common distribution to use in parameter estimation is the Gaussian distribution, which is used for continuous variables with unbounded domain. The distribution and corresponding loss function are defined as follows

$$p_{\text{Gauss}} \colon \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R} \tag{2.11}$$

$$p_{\text{Gauss}}(x \mid \mu, \sigma) \stackrel{\text{def}}{=} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{2.12}$$

$$L_{\text{Gauss}} \colon \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R} \tag{2.13}$$

$$L_{\text{Gauss}}(y, \hat{\mu}, \hat{\sigma}) \stackrel{\text{def}}{=} \frac{1}{N} \sum_n -\log p_{\text{Gauss}}(y \mid \hat{\mu}_n, \hat{\sigma}_n) \tag{2.14}$$

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.

2.3.3 Distributions over high-dimensional data

For example, 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, which I show below:

- Discretization
- Independence assumptions
- Auto-regressive factorization

Discretization

One approach to managing the intractibility 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.

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.

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.

$$p(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N) = \prod_i^N p(\boldsymbol{x}_i | \boldsymbol{x}_1, ..., \boldsymbol{x}_{i-1})$$
$$= p(\boldsymbol{x}_1)p(\boldsymbol{x}_2|\boldsymbol{x}_1)p(\boldsymbol{x}_3|\boldsymbol{x}_1, \boldsymbol{x}_2) ... p(\boldsymbol{x}_N|\boldsymbol{x}_1, ..., \boldsymbol{x}_{N-1}) \tag{2.15}$$

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):

$$J = 5, 3, 9, 1, ..., N$$
$$p(\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N) = p(\boldsymbol{x}_1)p(\boldsymbol{x}_2|\boldsymbol{x}_1)p(\boldsymbol{x}_3|\boldsymbol{x}_1, \boldsymbol{x}_2) ... p(\boldsymbol{x}_N|\boldsymbol{x}_1, ..., \boldsymbol{x}_{N-1}) \tag{2.16}$$

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 [13]. 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, and in this chapter I seek to understand this class of models at a broader level. I will discuss:

- 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 where I introduce an extremely generic gaussian-process-like task.

Firstly, I will discuss 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. In this section I will describe the attention operator.

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), and A is the pre-softmax attention logits.

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

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

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

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

For clarity, the expanded form of 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 $k(a, b) = a \cdot b$, then

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

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 takes $O(MN)$ space and time to compute. Despite this

drawback, the attention operation has proven extremely useful in a variety of tasks. There are many different ways to address this but I will not discuss them.

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. I will discuss this 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 independent of the order of the key and value vectors K_n and V_n, because the summation across n is commutative:

$$O_m = \sum_n V_n \sigma(A_m)_n \tag{3.2}$$

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 $Q_{m'}$, $m' \neq m$. 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 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 (3.2). 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 [12].) 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 Transformer models

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.

A transformer model puts attention operations together with MLP blocks similarly to in residual networks (ResNets). 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. This means that the outputs of an attention operation are linear combinations of the inputs. This is useful as a building block but we would like to be able to learn non-linear functions of the inputs. MLP blocks are used to introduce non-linearities into the model.

In Figure 3.1 we can see a diagram of a typical residual block in a transformer, combining an attention operation, a feed-forward layer, and a residual connection as in a ResNet.

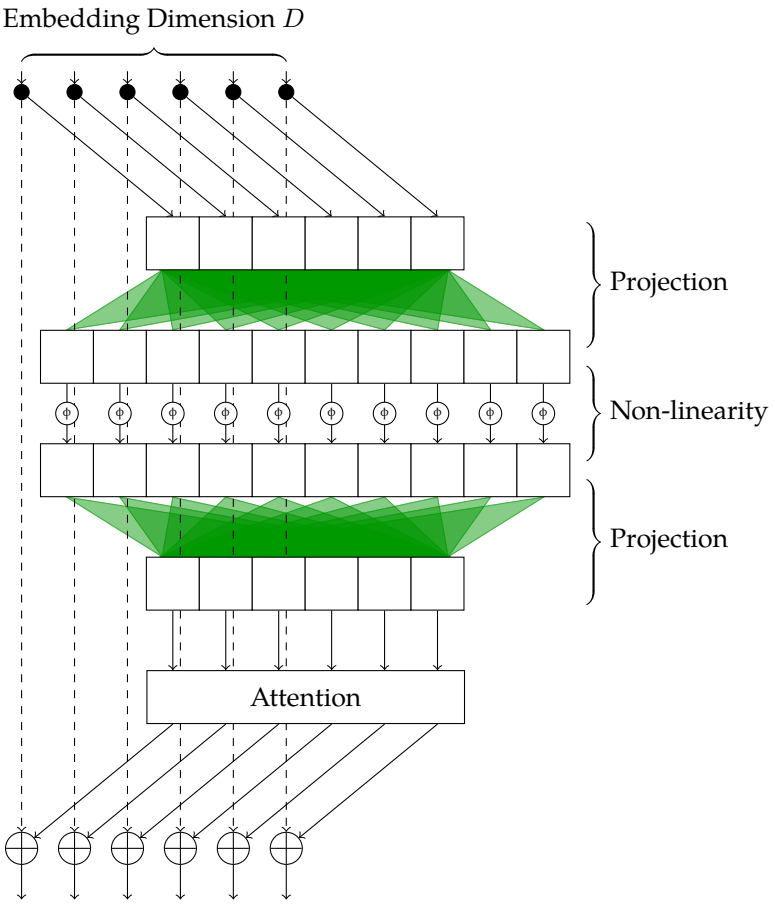

Figure 3.1: Typical residual block in transformer (with the D dimension expanded, rather than the N or M dimension as usual.).

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 dimension. 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.

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

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

If we derive all three matrices from the same input $\boldsymbol{X}$, then $M = N$ and the attention operation is called a *self-attention* operation. A diagram of this is shown in Figure 3.2. The blue shaded areas show the receptive field used when computing each output vector x'_i.

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

Since the introduction of transformers it is common to use *multi-head* attention, which allows for multiple *heads* which each perform an attention operation in parallel with smaller key dimensionality $D_{\text{head}} = \frac{D}{n_{\text{heads}}}$.

The defining feature of a transformer model is that it has "attention" layers. However, there is not just one way to assemble these layers, and there is not just one way to train these models.

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

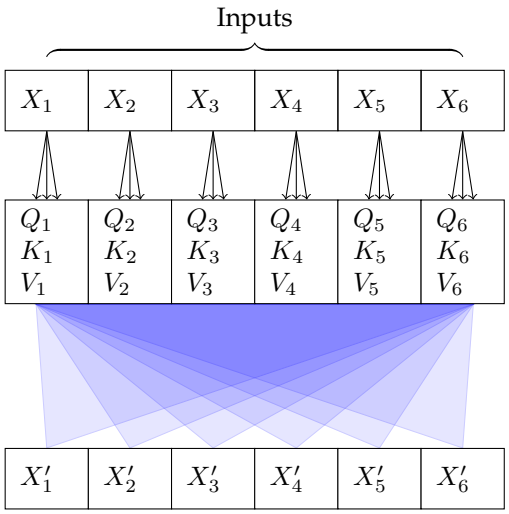

Figure 3.2: Full self-attention (bi-directional attention), as used in transformer encoders. The blue shaded regions show which inputs are used to compute each output. In full self-attention, all inputs are used to compute all outputs.

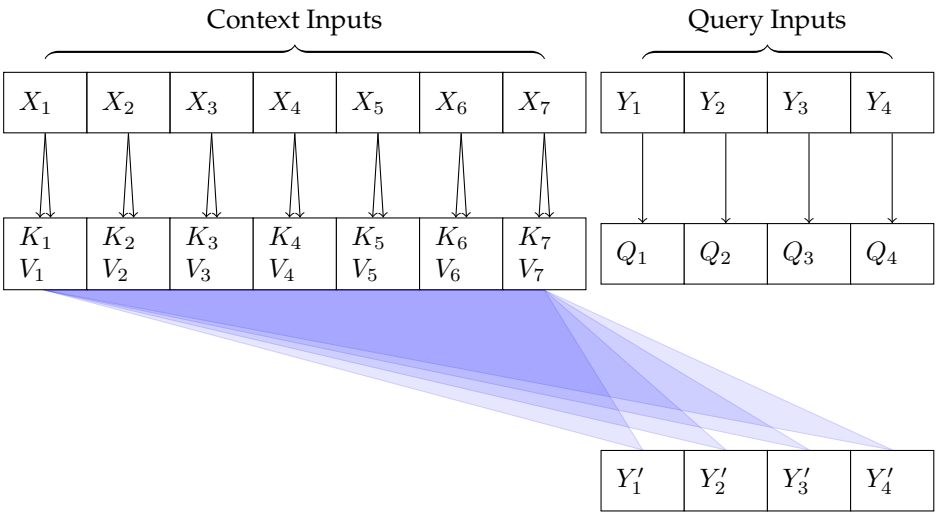

Figure 3.3: Cross-attention, as used in encoder-decoder models.

3.3 Masking and Pretraining

When transformer models are trained, they are often trained on large datasets of un-labeled data, which is called self-supervised learning or pre-training. 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 pre-training task. The two main types of pre-training task are *masked sequence modeling* and *auto-regressive pretraining*.

3.3.1 Masked sequence modeling

3.3.2 Causal Masking & Auto-regressive pretraining

3.3.3 Unified pretraining

3.4 Transformer Architectures

3.4.1 Masked Sequence Modeling: Encoder-only models

Arguably the simplest attention-based model architecture is encoder-only transformers. 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 [1] for speech, and SimMIM [14] image model.

These models are trained on a sequence reconstruction task, called Masked Language Modeling (MLM), Masked Image Modeling (MIM), or more generally Masked Sequence Modeling, which I will discuss more in **??**.

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, an encoder model in conjunction with a *decoder* model (see **??** 3.4.3), or simply a decoder-only model.

3.4.2 Sequence Prediction: Decoder-only models

A diagram of the decoder-only architecture, is shown below in **??**. The distinguishing feature of a *decoder* as opposed to an encoder is that its attention layers are all causally-masked self-attention layers as in Figure 3.4. These models are used for sequence prediction/generation, and trained via self-supervised learning.

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

- OpenAI's GPT-series [10, 2] language models.
- Latent code prediction (the "prior") in VQ-GAN [6]

3.4.3 Encoder-decoder models

When the transformer was introduced in [13], 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

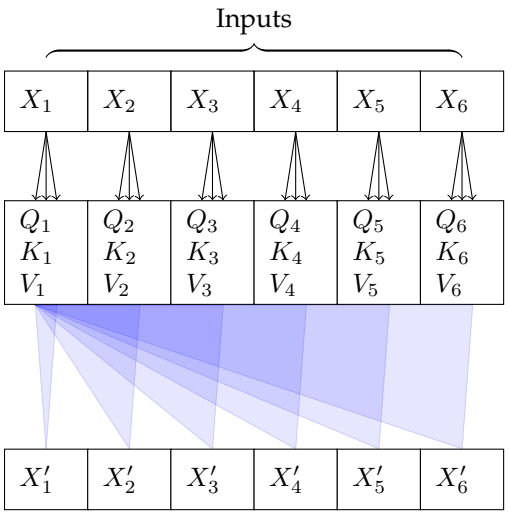

Figure 3.4: Self-attention with causal masking, (uni-directional attention) as used in transformer decoders during training. The blue shaded regions show which inputs are used to compute each output. In causal masking, only inputs to the left of the current output are used to compute the current output.

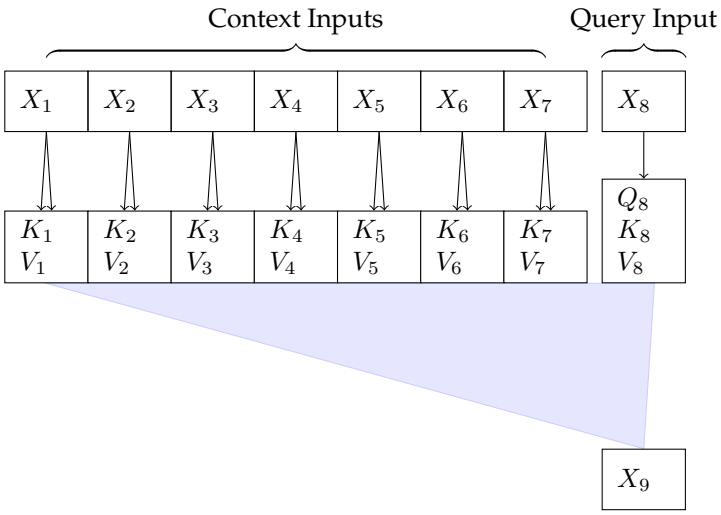

Figure 3.5: Partial self-attention, as used during incremental autoregressive inference (in models with a decoder).

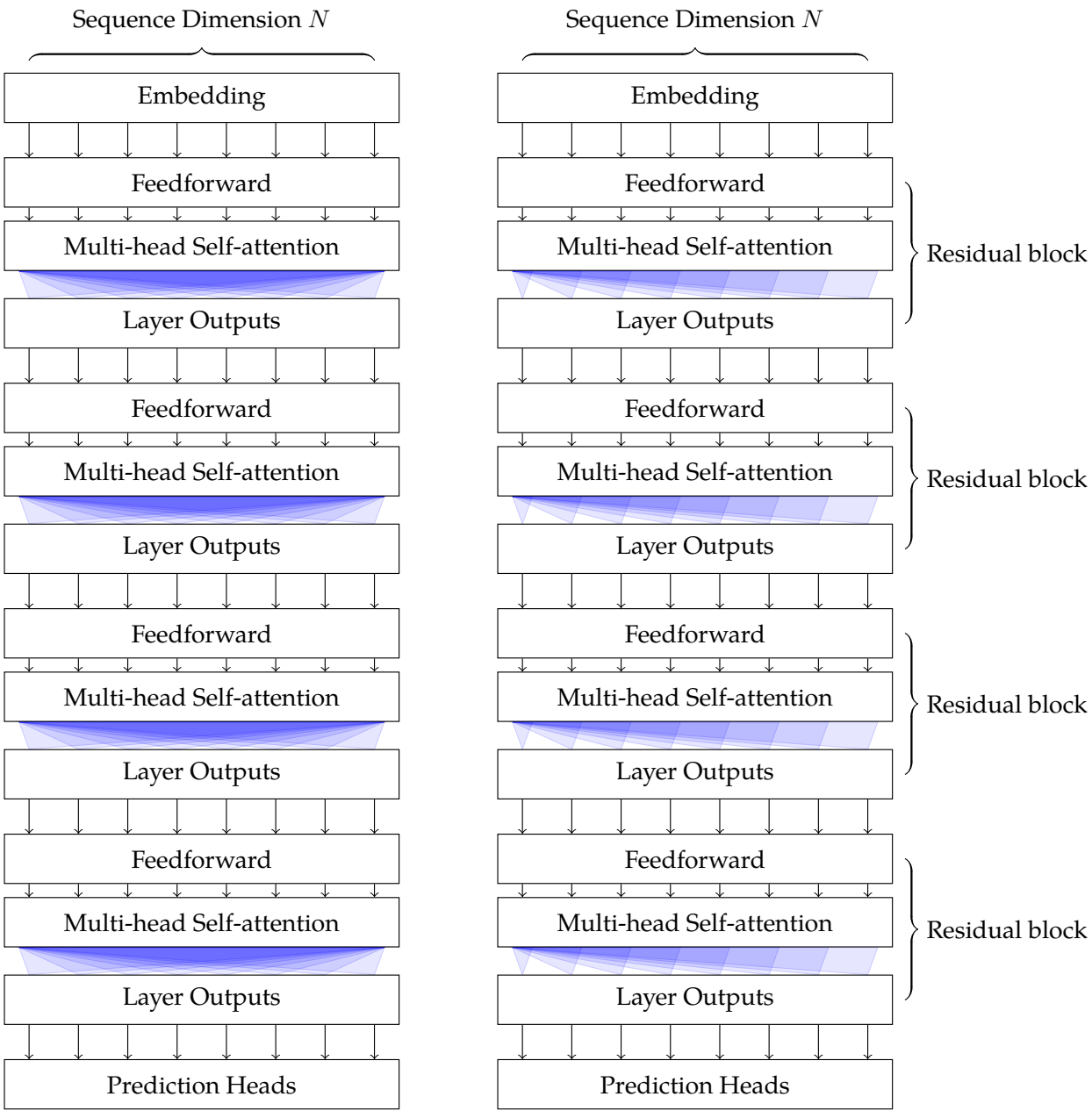

Figure 3.6: Left: Encoder-only model. Right: Deocder-only model. Residual connections have been ommitted for brevity.

layers (see Figure 3.3), which allow the decoder to attend to the encoded context sequence.

These are the most flexible class of model, because they allow predicting some outputs, or sequences of outputs, *given* some inputs.

Examples of this are the original transformer architecture [13], the BART [8] model, and more recently Google's Parti multi-modal text-to-image model [15].

In [13], they train an encoder-decoder architecture for text translation. Their architecture takes one sequence of text tokens (if you are wondering what text tokens are, I will discuss input formats in the next section) in language A as conditioning input, then auto-regressively samples a target sequence in a second language B. The two languages may have very different word orderings or numbers of words to each other, but the cross-attention operation introduces no bias towards aligned word orderings or even word counts.

3.4.4 Unified attention models

As we see in 3.7, there are actually two ways we might include information from the encoded sequence into the decoded sequence. In the first, which is used in most encoder/decoder models including the original, all the layers of the encoder are computed fully, and the final encoded sequence is provided to the decoder at all layers via cross-attention. In the second, the encoder is separated from the decoder only by correct application of masking in the attention layers, and is only computed up to the layer which is being decoded at.

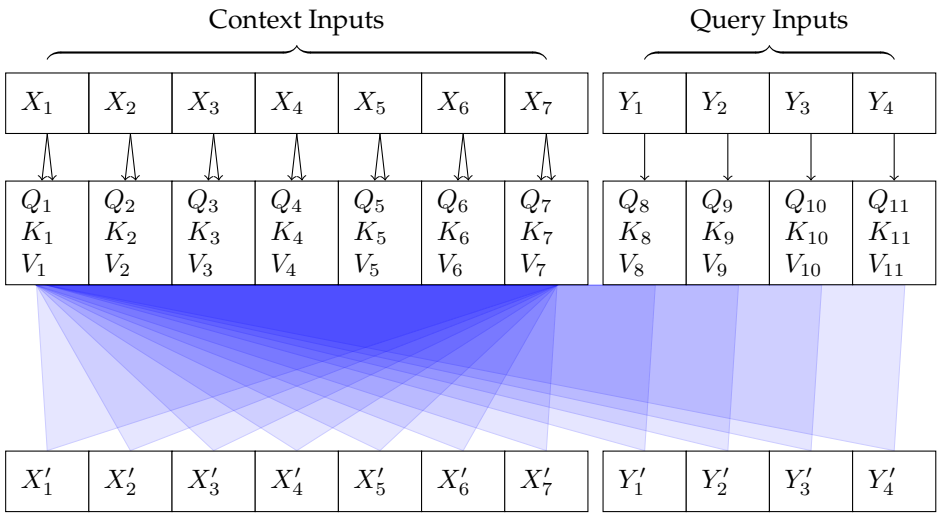

Figure 3.7: Unified self- and cross-attention, as in [5].

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.

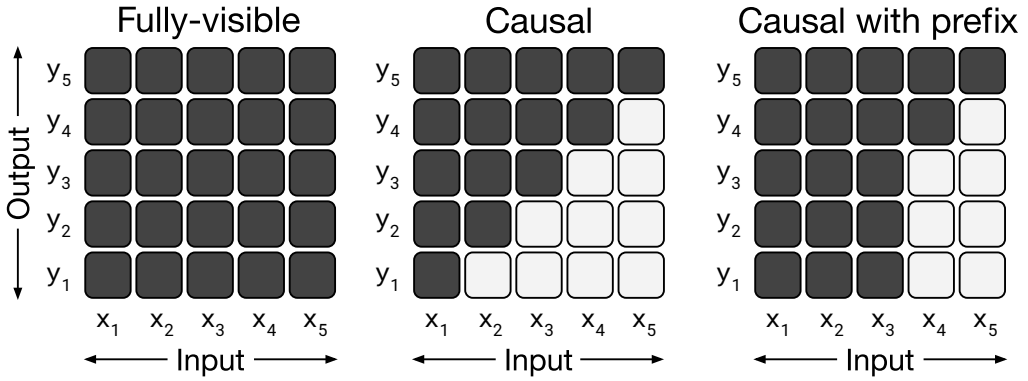

Figure 3.8: Masking in unified attention layers [5, 11]. The bi- and uni-directional attention can be performed within a single attention layer.

The different layouts that can be used are shown in Figure 3.8.

The first method is more efficient because the attention matrices of the encoder and the decoder are split, using $O(N^2 + M^2)$ memory rather than $O(N^2 M^2)$. However, the second method is conceptually simpler and is better when the trained model will be used for fine-tuning/transfer learning [11]., because it uses the decoder to attend to the encoded sequence at any layer, and not just the final layer. This is useful for tasks such as image captioning, where the decoder may want to attend to the encoded image at multiple layers, and not just the final layer.

Tranformer models trained on a mix of pre-training tasks using this architecture are known as T5 models (**T**ext-**T**o-**T**ext **T**ransfer **T**ransformers) [11].

3.5 Pretraining tasks

A *pre-training task* is a self-supervised task used to train a transformer. There are two main pre-training methods for transformer models which I have already mentioned: *Masked Sequence Modeling* and *Auto-regressive Sequence Modeling* (ASM).

In masked sequence modeling (MSM), the model is trained to reconstruct the original sequence, but with some of the inputs masked out. An example of this for a language model is shown in Figure 3.9.

The inputs to the tranformer models are typically sequences of discrete tokens, which each have a corresponding entry in a learned codebook of latent vectors. Masking out an input in this context means adding an additional token and learned embedding"<mask>" to the codebook, which

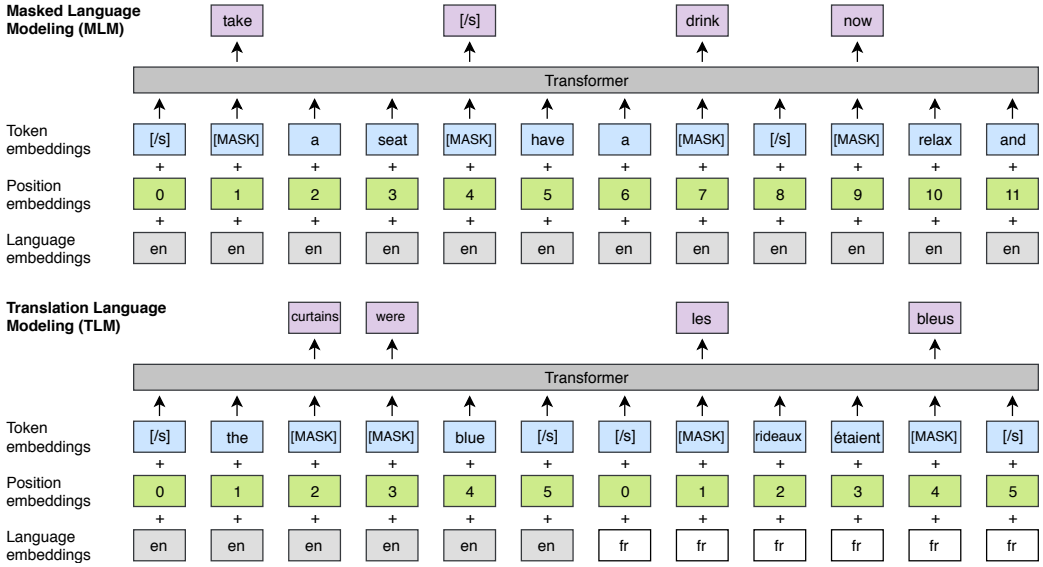

Figure 3.9: Cross-lingual masked language model pretraining, from [3]. Top: The standard MLM pretraining task. A sequence of text tokens are given as input, some are masked and the model is trained to predict the masked tokens.
Bottom: The cross-lingual MLM pretraining task. A sequence of text tokens from two different languages are given as input, some tokens are masked and the model is trained to predict the masked tokens. The model is trained on a mixture of monolingual and cross-lingual examples.

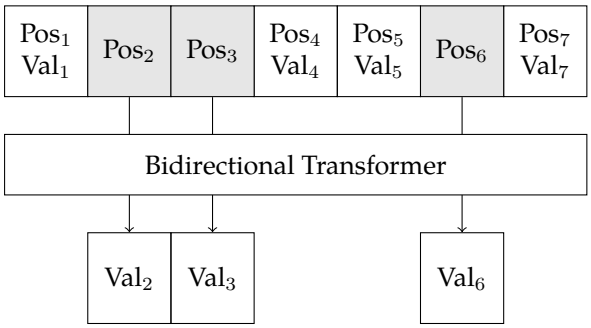

Figure 3.10: Masked sequence modeling pretraining task. The model is trained to reconstruct masked-out elements of the original sequence.

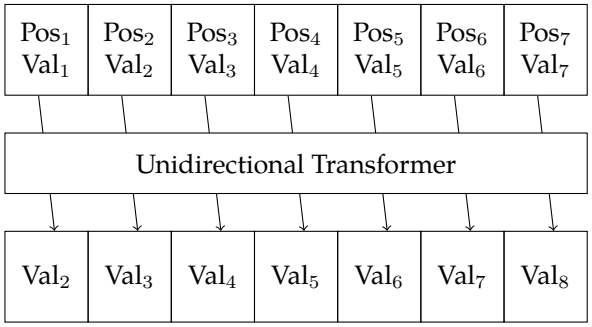

Figure 3.11: Autoregressive pretraining task. The model is trained to predict the next element in the sequence.

is used to represent the masked out inputs. As we can see in the figure, the masked positions still have embeddings of the positional information, which is necessary because a transformer model has no implicit order information available to it.

The second main pre-training task is auto-regressive sequence modeling, where the model is trained to predict the next token in a sequence, given the previous tokens. For this pre-training task the target outputs are present in the input sequence, but one token ahead. The model must therefore be a uni-directional / causally-masked model (Figure 3.6) to prevent information flow from the future to the past and prevent the model learning degenerate behaviour, as in Figure 3.4.

In the next chapter (Chapter 4) I also experiment with a new pre-training task, which is a variation on auto-regressive sequence modeling, which I call *arbitrary-order autoregressive pretraining*, which I will later apply to hand motion prediction.

Chapter 4

Sampling Sequences In Any Order

In this chapter I investigate learning auto-regressive transformers to generate pixels of the MNIST dataset. By changing the model architecture and training procedure, I show that we can learn to generate pixels in any order, including dynamically choosing the order of the pixels while sampling. The learned model can be used as a stochastic process on this 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, I train a transformer-based neural network model so that it can be used auto-regressively sample sequences of pixels in any order.

I then use this model 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, I find that this biases the samples towards images with large amounts of empty background, such as images of 1s. Correspondingly, a highest-entropy-first sampling order biases the samples towards images of 8s and 9s. In the discussion, I argue 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 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 $p(y_n|x_n, \boldsymbol{x}_{<i})$. This gives us N conditionally-independent distributions each for a different x_n. We can choose any statistic of these

distributions to select which one to sample from. 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. Because we sample, we in principle do not change the distribution of results – however in practice, we may change the distribution of results because of the way we choose the statistic. I will discuss this in Section 4.4.

4.1.2 Training task and input formats

As we saw before in Section 3.5 we have two main tasks for training a transformer model:

- **Masked Sequence Modeling**, which we use to train models with bi-directional attention - predict the masked out tokens.
- **Autoregressive Sequence Prediction**, which we use to train a model that uses causal attention - predict the next token in sequence.

As we saw in the above section, transformers can have their input provided as (position, content) pairs, which naturally maps onto the above auto-regressive factorization. However, not all training tasks will result in a model that learns to use the position information in this way.

More generally, the input to a transformer is some kind of D-dimensional embedding. When specifiying both the input and content , we first project the which is unique among the input set, such as special "BEGIN" or "CLASS" tokens.

Both the attention layers and feed-forward layers are invariant to permutations of the input sequence. Additionally, there is no requirement that the input set be contiguous in the sequence dimension - there can be (potentially large) gaps, with no change to to the structure of the model (however, the model must be trained for the particular problem still).

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

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

Arbitrary order auto-regressive pre-training using input triples

This is a variant on auto-regressive sequence modeling that I developed for this project. Let us recall causal/autoregressive pretraining from the previous chapter. (See Figure 3.11) Recall that these predict the next input from the previous input, conditioned on the rest of the sequence via their attention layers.

We can represent the input 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 we are to use random positions, the model is not able to infer the position to predict. We instead construct an input sequence in the following way

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

.

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 is presented in contiguous-forward-only ordering, i is always paired with $i + 1$ and we do not introduce any new information. However, since we create a sequence with an arbitrary order during training, the model learns to utilize this information.

Then, 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.

4.2 Hypothesis

Using the above two methods "triples" and "pure-query-decoder" models, I investigated a hypothesis about selecting a better sampling order on a toy

dataset.

The hypothesis was as follows.

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

When auto-regressively sampling pixels to produce MNIST images, using a "lowest-entropy-first" ordering might produce visually better results than a "highest-entropy-first" ordering.

4.3 Method

To investigate this hypothesis, I developed and trained a transformer model on two separate tasks - one for predicting the next pixel in a sequence, and one for predicting a random. I compared a variety of architecture choices and training methods. I show the results of these experiments as figures for subjective evalutaion throughout this chapter.

4.3.1 Data

This series of experiments uses the MNIST dataset, which is a set of 28x28 grayscale images of handwritten digits. The dataset is 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 I did not use this information in my 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, I used a 2-bit representation, where each pixel is represented as a value between 0 and 3. I discretized the 256 colors into 4 colors using a learned k-means clustering over the whole dataset. This was to simplify the form of the distribution that the model would output, and remove any complextity here as an additional variable to debug. I found that 4 colors was the smallest representation that gave good visual quality when the images were reconstructed.

To turn MNIST data into a sequence prediction task, I used the following procedure. For each image, I created a sequence of 28x28 pairs, where each pair is a pixel position and a pixel value. The pixel position is represented as a pair of integers, where the first integer is the row number, and the second integer is the column number. The pixel value is represented as a single integer, where 0 is black, 1 is dark gray, 2 is light gray, and 3 is white.

The tasks the model was trained on are then both as follows: "Given a sequence of pairs, and a target position, predict the value of the pixel at that position."

However, in the first task the pairs are presented in a contiguous forward-only order, and in the second task the pairs are presented in a non-contiguous, random order. In the random order task, many unique permutations of each training image are created using on-the-fly data augmentation. This is to ensure that the model is not overfitting to a specific ordering of the training data.

We can see examples of this in fig. 4.1.

In addition to the above encoding I experiment with three different pre-

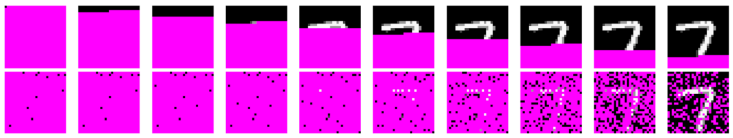

Figure 4.1: Examples of the two tasks, showing input sequences with progressively more pixels filled in. (Pink means the pixel was not provided). Top: Pixels are presented in a contiguous forward-only order. Bottom: Pixels are presented in a non-contiguous, random order.

training task formats for the data. The first is forward-only auto-regressive pretraining as in the previous chapter (Figure 3.11). The second is a "Masked Sequence Modeling" task, where I mask out a random subset of the pixels in the image, and the model is trained to predict the masked out pixels, and the final one is the "Arbitrary-order auto-regressive pretraining" variant I describe above in Section 4.1.2.

4.3.2 Models

Architecture choices

All model architectures shown here are transformer models, but they vary in the following ways:

1. Absolute vs relative positional encodings

2. Whether or not they include a "query decoder".

3. If so, whether this decoder has a pooled input or not.

4. Number of layers, embedding dimensionality etc.

4.4 Discussion

As we can see in **??**, the "lowest-entropy-first" ordering produces distinct images from the "highest-entropy-first" ordering. However, neither are as good as the "random" ordering.

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 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 en-

tropy 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.

Chapter 5

Angles, Joints and Hands

Before I discuss my experiments training a model on hand motion data in the next chapter, I will first introduce some background related to modeling human hands, so we can understand many of the implementation choices I made.

5.1 Parameterizing Hand Configurations

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

As we can see 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.

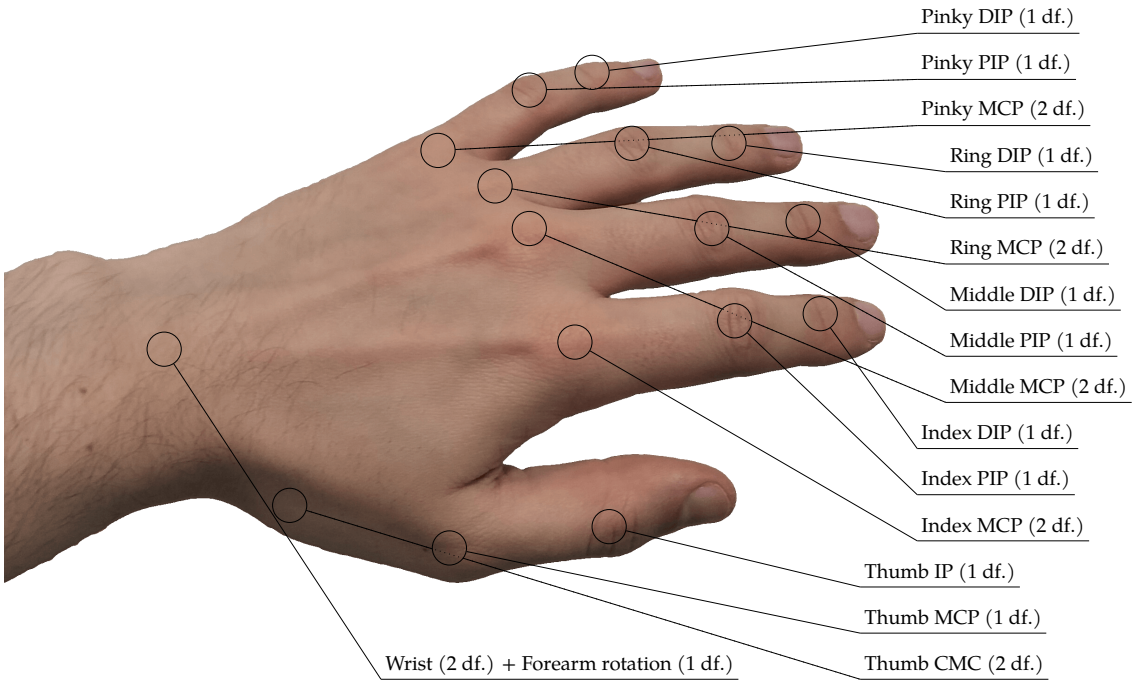

Figure 5.1: Each hand has 16 joints - three per digit, plus the wrist.

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.

5.1.1 Euler Angles

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

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

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

1. there may be multiple sets of Euler angles that correspond to the same rotation

2. as a result, we cannot easily interpolate between two configurations that are close together but have different Euler angles

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

todo: Euler angle figure

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

5.1.2 Axis-Angle

An alternative parameterization 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.

5.2 Loss Functions for learning angles

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

A variant of this is the *angular* mean-squared-error, which I will use

later on in Chapter 6.

$$L_{\theta\text{-MSE}} : \mathbb{R}^{N\times D} \times \mathbb{R}^{N\times D} \to \mathbb{R}$$
$$L_{\theta\text{-MSE}}(y, \hat{y})_{ni} \overset{\text{def}}{=} \frac{1}{N} \sum_n \left[\sum_i (\sin y_{ni} - \sin \hat{y}_{ni})^2 + (\cos y_{ni} - \cos \hat{y}_{ni})^2 \right] \tag{5.1}$$

Here y and $\hat{y}$ are vectors of angles, and the error is defined in terms of the squared arc length between their corresponding components. This loss function is equivalent to maximising the likelihood of a von Mises distribution, the derivation for which can be found in Chapter 5. This is useful when we want to model angles, for example when we want to predict the orientation of a hand, which we will do in Chapter 6.

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

Chapter 6

Hand Motion Model

I experimented with a variety of different model architectures,

Loss Value mean across whole dataset	0.07275154
Total number of frames in dataset	467372
Total length of dataset	4h 19m 39s

6.1 Predicting the next frame

6.2 Learning a probabilistic model

Chapter 7

Conclusions

To conclude, I will summarize the main conclusions from the experiments in Chapter 4 and Chapter 6, and share some reflections on the thesis.

7.1 Conclusions

todo: Add conclusions from the experiments.

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 hy-

perparameters, 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 debugging 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 really enjoyed working on this project, more than I expected at the outset. I have found a deep passion for learning about deep learning. The project has given me the opportunity to learn an insane amount about machine learning techniques, including modern methods such as transformers, and to apply this knowledge to a practical problem.

Bibliography

[1] 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.

[2] 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.

[3] Alexis Conneau and Guillaume Lample. "Cross-lingual language model pretraining". In: *Advances in neural information processing systems* 32 (2019). arXiv: 1901.07291. URL: http://arxiv.org/abs/1901.07291.

[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] 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.

[7] 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.

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.