Conditional Generative Transformers for Hand-Guided Animation Automation

by

Maxwell Clarke

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

Master of Science
in Computer Science.

Victoria University of Wellington

November 2022

Abstract

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

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

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

Contents

List of Figures

List of Tables

Chapter 1

Background

1.1 Neural Networks and Deep Learning

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

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

1.1.1 Notation

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

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

$$\begin{aligned} &\phi_{\text{relu}} \colon \mathbb{R} \to \mathbb{R} \\ &\phi_{\text{relu}}(x) \overset{\text{def}}{=} \max(0, x) \end{aligned} \tag{1.1}$$

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

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

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

This notation also works for multiple dimensions, including when an operation is not applied independently across some dimension. For example, the following is how I will show the *softmax* function, which is a vector valued function, applied independently across the rows of a $B \times N$ matrix $\boldsymbol{X}$ (which is used in the definition of the attention operation in Chapter 2).

The softmax function is defined as:

$$\begin{aligned} &\sigma \colon \mathbb{R}^N \to \mathbb{R}^N \\ &\sigma(\boldsymbol{x}_n) \overset{\text{def}}{=} \frac{e^{\boldsymbol{x}_n}}{\sum_{n'} e^{\boldsymbol{x}_{n'}}} \end{aligned} \tag{1.2}$$

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 [4] or TensorFlow [6]. 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.

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

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

$$\begin{aligned}f_{\text{mlp}}&\colon \mathbb{R}^N \to \mathbb{R}^D \\ f_{\text{mlp}}(\boldsymbol{x}) &\overset{\text{def}}{=} \phi(W\boldsymbol{x}) + b\end{aligned} \tag{1.3}$$

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

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

D-dimensional vector.

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

$$
\begin{aligned}
& f_0 \colon \mathbb{R}^N \to \mathbb{R}^D \\
& f_0(\boldsymbol{x}) = \phi(W_0 \boldsymbol{x}) + \boldsymbol{b} && W_0 \in \mathbb{R}^{N\times D} && \boldsymbol{b}_0 \in \mathbb{R}^D \\
\\
& f_\ell \colon \mathbb{R}^D \to \mathbb{R}^D && \forall \ell \in 1, \dots, L \\
& f_\ell(\boldsymbol{x}) = \phi(W_\ell f_{\ell-1}(\boldsymbol{x})) + \boldsymbol{b}_\ell && W_\ell \in \mathbb{R}^{D\times D} && \boldsymbol{b}_\ell \in \mathbb{R}^D \\
\\
& f_L \colon \mathbb{R}^D \to \mathbb{R}^C \\
& f_L(\boldsymbol{x}) = \sigma(W_L f_{L-1}(\boldsymbol{x}) + \boldsymbol{b}_L) && W_L \in \mathbb{R}^{D\times C} && \boldsymbol{b}_L \in \mathbb{R}^C
\end{aligned}
\tag{1.4}
$$

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

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

1.1.2 Tasks

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

On the following pages, 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 1.1)

2. Data dimensionality / length (Figure 1.3)

3. Dataset format (Figure 1.2)

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*:

$$\begin{aligned} L_{\text{MSE}} &: \mathbb{R}^{N \times D} \times \mathbb{R}^{N \times D} \to \mathbb{R} \\ L_{\text{MSE}}(y, \hat{y})_{ni} &\overset{\text{def}}{=} \frac{1}{N} \sum_n \left[\sum_i (y_{ni} - \hat{y}_{ni})^2 \right] \end{aligned} \tag{1.5}$$

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

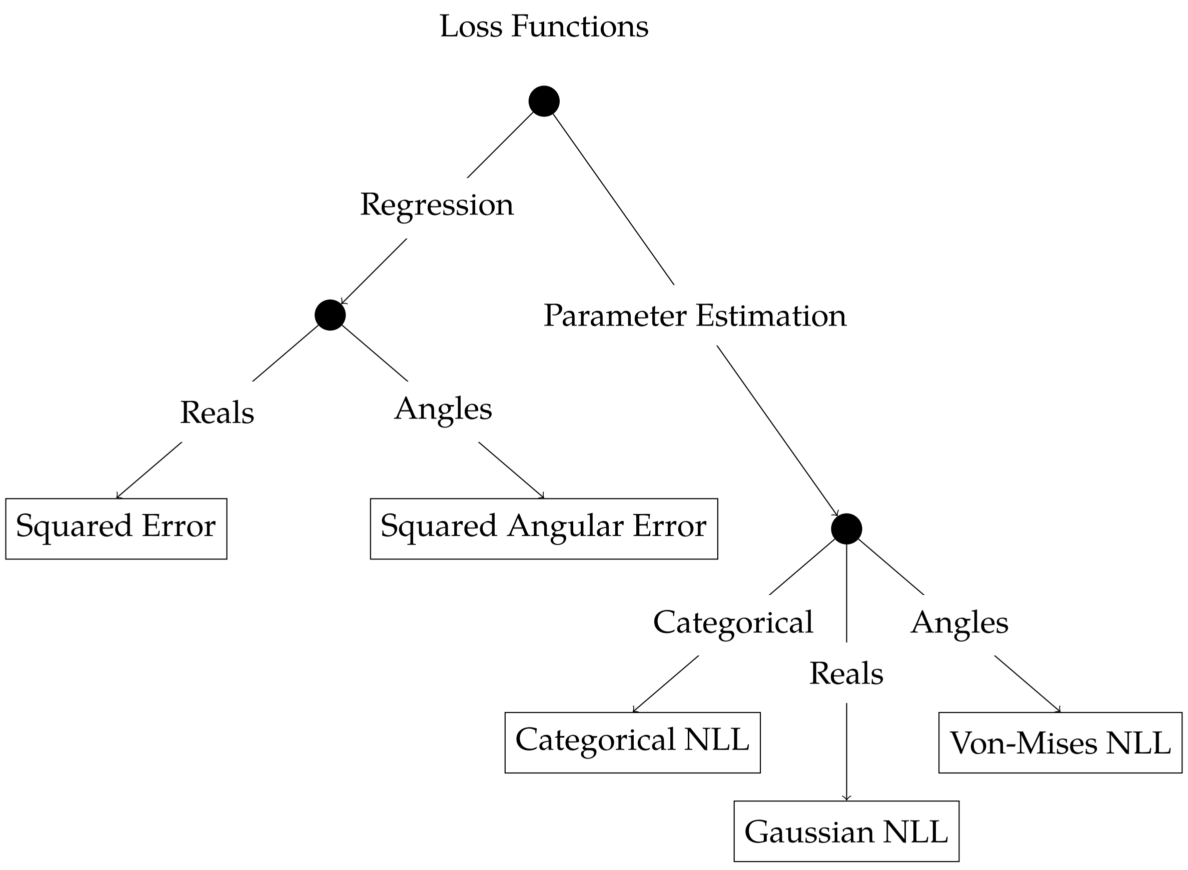

Figure 1.1: We can split loss functions into two categories.

In the former, the loss function has the form of an error function. When minimizing this function, the model learns to output the expected value of the posterior $\mathsf{E}\left[p(y \mid x)\right]$ of the output y given the input x. This is called a regression or maximum-a-posteriori (MAP) 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, etc. Depending on the task, and output format, different functional forms for p may be appropriate.

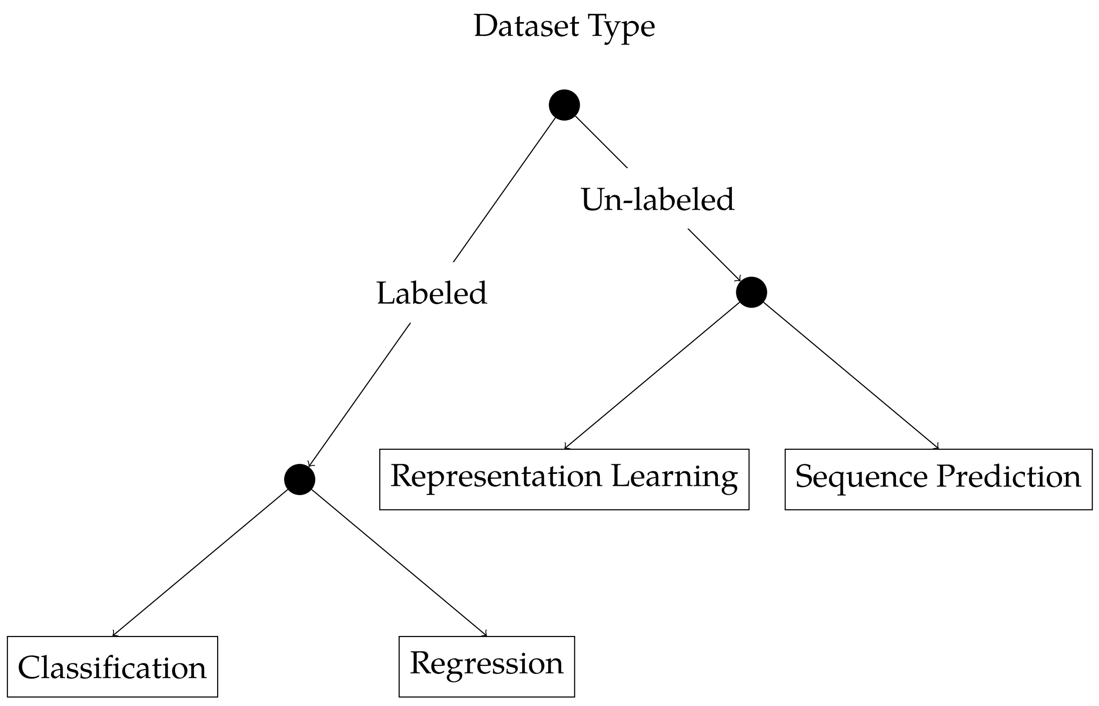

Figure 1.2: Basic ontology of dataset types.

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

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

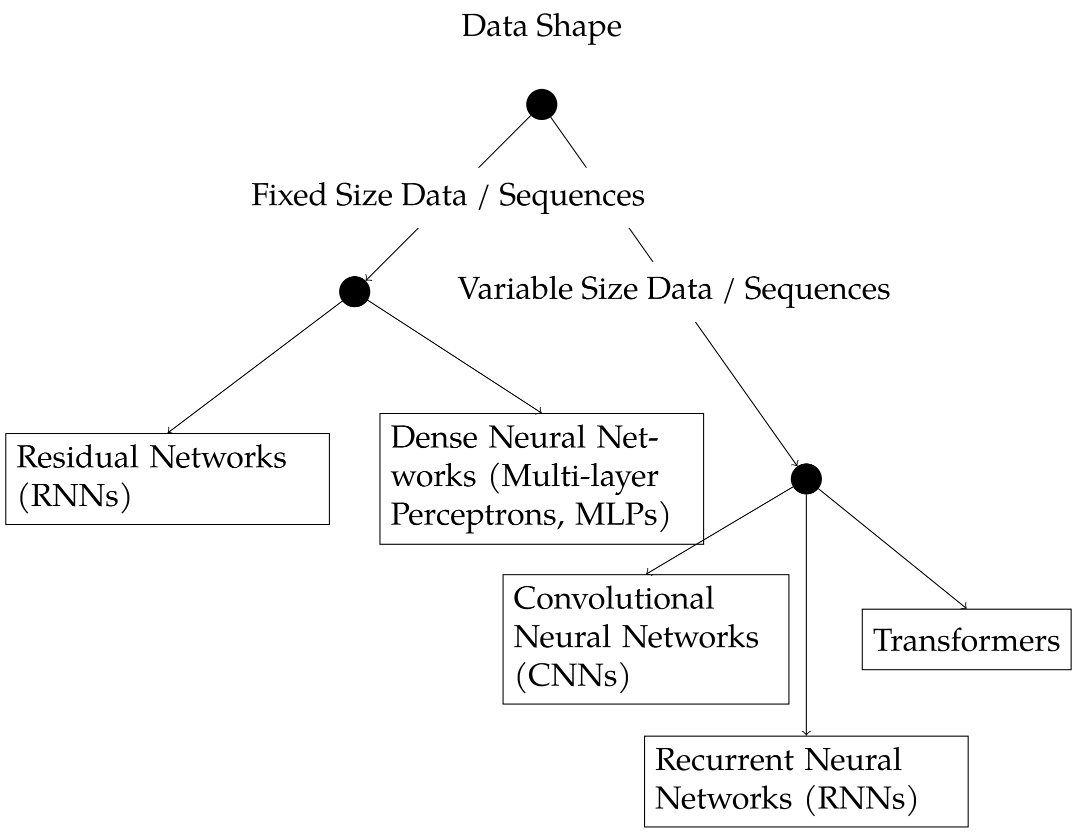
Data Shape
Fixed Size Data / Sequences
Variable Size Data / Sequences
Residual Networks (RNNs)
Dense Neural Networks (Multi-layer Perceptrons, MLPs)
Convolutional Neural Networks (CNNs)
Transformers
Recurrent Neural Networks (RNNs)

Figure 1.3: Neural network variants which support variable input shape.

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

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

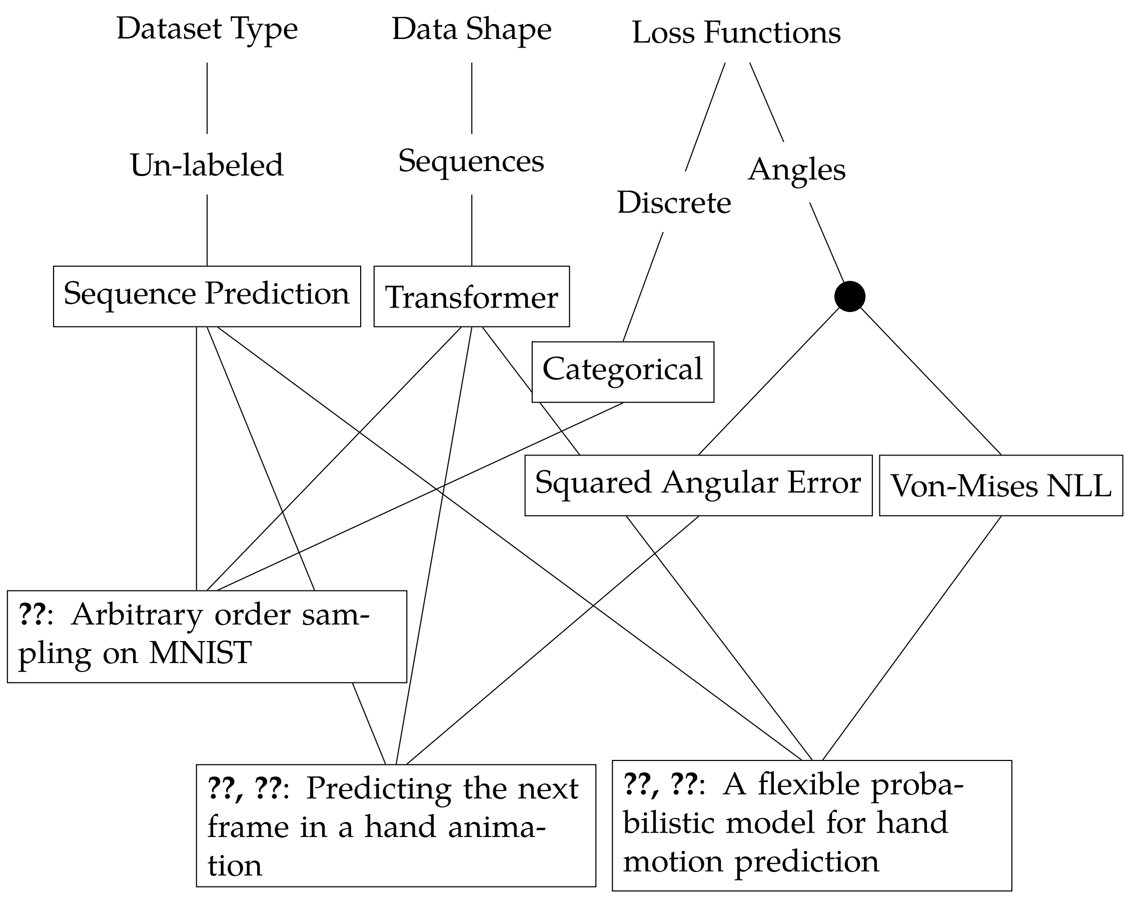
Dataset Type
Data Shape
Loss Functions
Un-labeled
Sequences
Discrete
Angles
Sequence Prediction
Transformer
Categorical
Squared Angular Error
Von-Mises NLL
??: Arbitrary order sampling on MNIST
??, ??: Predicting the next frame in a hand animation
??, ??: A flexible probabilistic model for hand motion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

Figure 1.4: The later work in this thesis sits focuses on learning un-labeled sequence data with transformers, in two different domains.

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

In **??**, I train a transformer-based model on the ManipNet hand motion dataset. **??** focuses on a deterministic model, while **??** focuses on a probabilistic model.

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

1.2 Auto-regressive models

todo: Formulation of auto-regressive models and sampling

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

1.3 Animation

1.4 Angle representations

todo: Quaternions and unitary / complex matrices

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

Chapter 2

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

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

2.1 The Attention Operation

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

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

2.1.1 Mathematical Definition

An attention operation is of the following form, using short summation notation, where σ is the *softmax* operator (see 1.2), and A is the pre-softmax attention logits.

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

$$f_{\text{attn}}(Q, K, V)_{mv} \overset{\text{def}}{=} \sum_{n} \left[\sigma\Big(\sum_{d} Q_{md} K_{nd} \Big)_{mn} V_{nv} \right] \tag{2.1}$$

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

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

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:

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

or when $k(a, b) = a \cdot b$, then

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

$$= QK^T.$$

We can see that the attention weights have shape $M \times N$. This is the primary drawback of the attention operation, since in self attention $M = N$, and so takes $O(M^2)$ 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.

2.1.2 Permutation-invariance in the context inputs

The first interesting property of attention is that it is permutation-invariant with respect to the Key-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 **??**).

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:

$$O_m = \sum V_n \sigma(A_m)_n \tag{2.2}$$

2.1.3 Permutation-equivariance in the query inputs

Attention is also permutation-*equivariant* with respect to the Query inputs. This 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 operationk is equivariant to the order of its inputs, which we can see from the construction in Equation (1.2).

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.

2.1.4 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 this in Equation (2.2). 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.

2.1.5 Attention variants

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

$$\begin{aligned} Q &= W_Q\boldsymbol{X} + \boldsymbol{b}_Q \\ K &= W_K\boldsymbol{X} + \boldsymbol{b}_K \\ V &= W_V\boldsymbol{X} + \boldsymbol{b}_V \end{aligned} \tag{2.3}$$

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 2.1. The blue shaded areas show the receptive field used when computing each output vector x'_i.

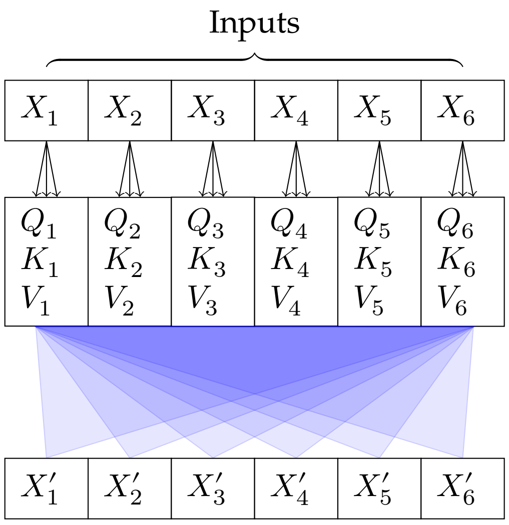

Figure 2.1: Full self-attention, as used in transformer encoders.

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

2.2 Transformer architecture variants

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

We can broadly split the transformer architecture variants into three categories: *encoder-only*, *decoder-only* and *encoder-decoder* architectures.

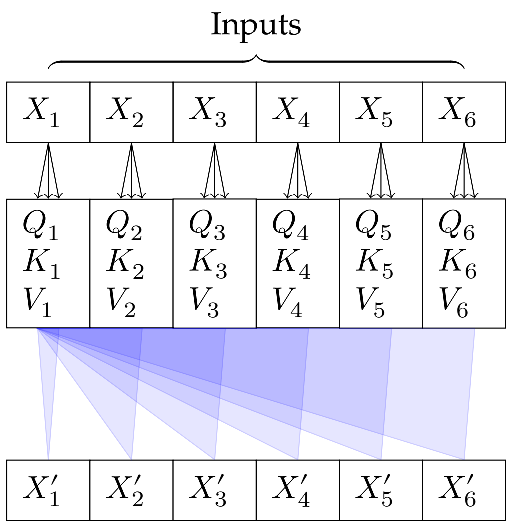

Figure 2.2: Self-attention with causal masking, as used in transformer decoders during training.

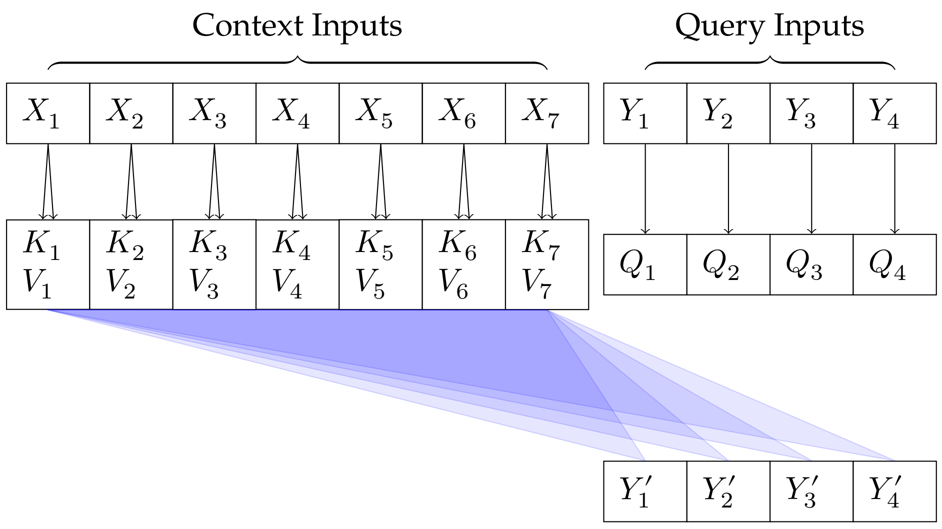

Figure 2.3: Cross-attention, as used in encoder-decoder models.

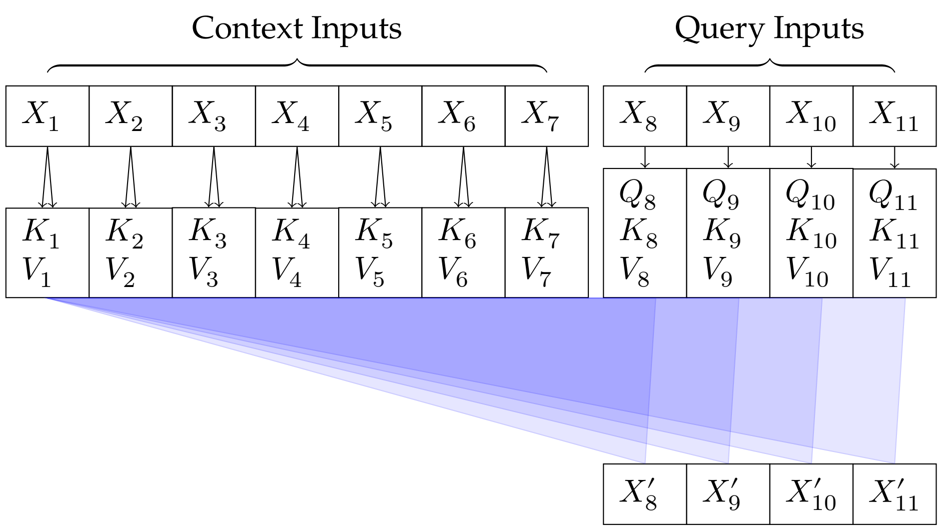
Context Inputs
Query Inputs
X_1 X_2 X_3 X_4 X_5 X_6 X_7
X_8 X_9 X_{10} X_{11}
K_1 V_1 K_2 V_2 K_3 V_3 K_4 V_4 K_5 V_5 K_6 V_6 K_7 V_7
Q_8 K_8 V_8 Q_9 K_9 V_9 Q_{10} K_{10} V_{10} Q_{11} K_{11} V_{11}
X'_8 X'_9 X'_{10} X'_{11}

Figure 2.4: Partial self-attention, as used during incremental inference in models with a decoder.

2.2.1 Masked Sequence Modeling: Encoder-only models

Arguably the simplest attention-based model architecture are encoder-only models. These are models trained on a sequence reconstruction task. Examples are the BERT [3] language model family, Wav2Vec [1] for speech, and SimMIM [9] image model.

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

2.2.2 Sequence Prediction: Decoder-only models

Currently the most common transformer architecture is the decoder-only architecture, a diagram of which I show below in **??**. These models are used for sequence prediction/generation, and trained via self-supervised learning. The distinguishing feature of a decoder-only model is that its only attention layers are self-attention layers which have a causal mask applied during training.

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

- OpenAI's GPT-series [7, 2] language models.
- Latent code prediction (the "prior") in VQ-GAN [11]

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

2.2.3 Encoder-decoder models

When the transformer was introduced in [8], 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 layers (see Figure 2.3), which allow the decoder to attend to the encoded context sequence.

Examples of this are the original transformer architecture [8], the BART [5] model, and more recently Google's Parti multi-modal text-to-image model [10].

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

[4] 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.

[5] Mike Lewis et al. "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension". In: *CoRR* abs/1910.13461 (2019). arXiv: 1910.13461. URL: http://arxiv.org/abs/1910.13461.

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

[7] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[8] 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.

[9] 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.

[10] 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.

[11] Jiahui Yu et al. "Vector-quantized Image Modeling with Improved VQGAN". In: *CoRR* abs/2110.04627 (2021). arXiv: 2110.04627. URL: https://arxiv.org/abs/2110.04627.