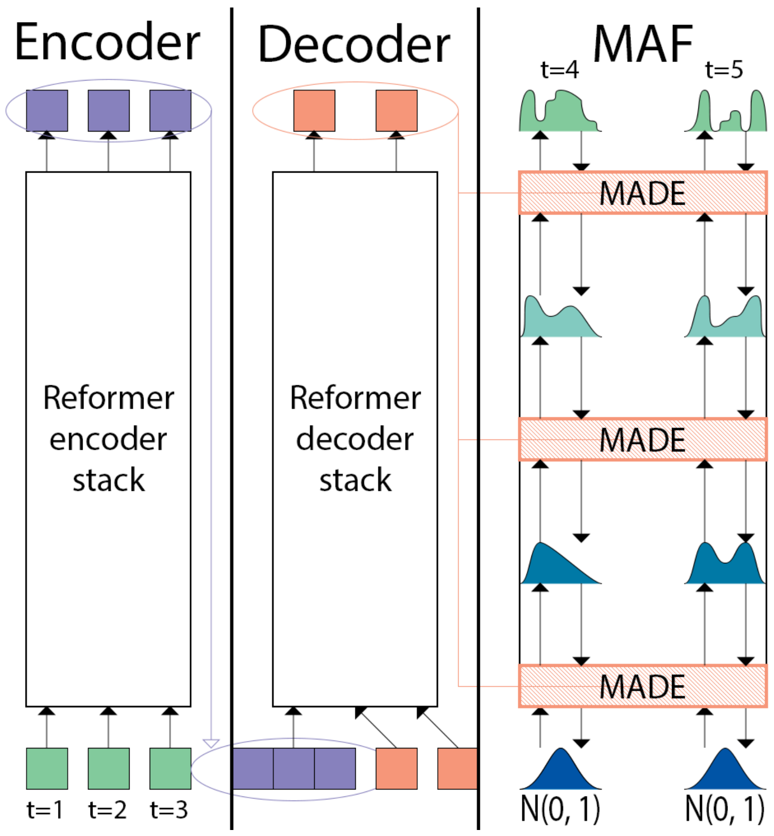
Degree Project in Computer Science and Engineering, specializing in Machine Learning

Second Cycle 30.0 credits

Probabilistic Forecasting through Reformer Conditioned Normalizing Flows

SAMUEL NORLING



Stockholm, Sweden 2022

Probabilistic Forecasting through Reformer Conditioned Normalizing Flows

Samuel Norling

February 2022

Abstract

Forecasts are essential for human decision-making in several fields, such as weather forecasts, retail prices, or stock predictions. Recently the Transformer neural network, commonly used for sequence-to-sequence tasks, has shown great potential in achieving state-of-the-art forecasting results when combined with density estimations models such as Autoregressive Flows. The main benefit of the resulting model, called Transformer Masked Autoregressive Flow (TMAF), is its novel architecture which significantly improves the computational efficiency compared to the older architectures such as recurrent neural networks. However, the Transformer comes with a high computational cost with time complexity of $O(N^2)$. In an attempt to mitigate this limitation, this thesis introduces a new model for forecasting, the Reformer Masked Autoregressive Model (RMAF), based on the Transformer Masked Autoregressive Flow (TMAF), where we replace the Transformer part of the model with a Reformer. The Reformer is a modified Transformer with a more efficient attention function and reversible residual layers instead of residual layers. While The Reformer has shown great promise in reducing the computational complexity in long sequence machine translations tasks, our analysis shows that the overhead induced by the Reformer model leads to a 7-8 times increase of memory allocated compared to the TMAF to reach the same forecasting quality on Solar and Electricity datasets.

Sammanfattning

Prognoser är essentiella för mänskligt beslutsfattande inom flertalet områden, så som väderprgonoser, handelprognoser och aktieprognoser. Nyligen har neurala nätverket Transformer, en modell som vanligtvis används för att mappa sekvenser till sekvenser, visat stor potential i att uppnå bästa möjliga prognosresultat när den kombineras med sannolikhettäthetsmodeller så som Autoregressive Flows. Den huvudsakliga fördelen med den kombinerade modellen, som kallas Transformer Masked Autoregressive Flow (TMAF), är dess banbrytande arkitektur vilket signifikant förbättrar beräkningseffektiviteten jämfört med äldre arkitekturer så som recurrent neurala nätverk. Transformern har dock en hög beräkningskostnad med tidskomplexiteten O(N^2). I ett försök att åtgärda denna begränsande kostnad så introducerar denna uppsats en ny modell för att skapa prognoser, Reformer Masked Autoregressive Flow (RMAF), där vi ersätter Transformer-delen av modellen med en Reformer. Reformern är en modifierad Transformer med en mer effektiv attention-funktion och reversible residual layers istället för residual layers. Medan Reformern har visat lovande resultat kring att reducera beräkningskomplexiteten för maskinöversättningsuppgifter med långa sekvenser så visar vår analys att överhuvudskostnaden som Reformer-modellen introducerar leder till en 7-8 gånger större mängd minnesallokering jämfört med TMAF för att uppnå samma prognoskvalite över Solar dataset och Electricity dataset.

Acknowledgements

First, I would like to thank supervisors Dr. Michael Welle and Ph.D. student Petra Poklukar from the Robotics, Perception and Learning division at Kungliga Tekniska Högskolan. Their feedback and encouragement were of incredible quality.

Second, I want to thank examiner Dr. Danica Kragić Jensfelt, Director of the Centre for Autonomous Systems at Kungliga Tekniska Högskolan, who made this thesis possible by accepting the thesis project idea and granting me her Ph.D. students as supervisors.

Third, I would like to thank Dr. Kashif Rasul, Principal Research Scientist at Zalando, for answering my emails about his research. Without his prior work, this thesis would not have been possible.

Finally, I want to express gratitude and love to my partner Anna.

Contents

1 Introduction

Forecasts are essential for human decision-making; for example, weather reports help us decide what clothes to wear outside, stock price predictions guide our investments, product demand forecasts help us plan business decisions, and more. All of these examples fall under the umbrella of time series forecasting.

The output of forecasts can either be point estimates or probability distributions. In this thesis, we focus on forecasts that produce probability distributions which help us understand the uncertainty of the predictions. Imagine that we are about to launch an expensive space rocket to Mars in order to collect some valuable rocks and bring them back to Earth. Rockets are expensive, and we want to minimize the risk of failure due to external factors such as the weather. If we then do a forecast about the weather, it is beneficial not just to produce point estimates but rather probability distributions because we can then schedule a rocket launch at a date where the forecast probability distribution's expected value is good, and the uncertainty is low. If we did a weather forecast that produced only point estimates, we might book a launch at a date where the forecast, in reality, is uncertain, but it is not something we have considered.

Classical time series methods could only produce univariate forecasts or were very inefficient for multivariate forecasts. A univariate time series is a sequence of values measured over time in discrete or continuous units. In contrast, a multivariate time series consists of sequences of values of several contemporaneous variables changing over time [7]. Let us say we are modeling the amount of visiting customers at restaurants in a shopping mall. Suppose one of the restaurants has a big promotion with reduced food prices and famous musicians performing a Friday evening. In that case, the amount of visitors for this restaurant likely goes up and the amount of visitors for the other restaurants in the shopping mall goes down. This type of effect can be learned with multivariate time series modeling but not with univariate time series modeling.

One can often obtain a better understanding through studying several related variables than just one variable [45], the example above with the restaurants in the shopping mall is one such case. The drawback of modeling relationships between several variables is that the complexity of the model grows.

What makes multivariate time series forecasting more complex than univariate time series forecasting is the introduction of dependencies between the contemporaneous variables. Some classical methods assume linear relationships among variables [6]. However, in the real world, the temporal variations in the data are not so simple and are very hard for even human experts to understand fully. Through the use of neural networks, one can model the dependencies between the contemporaneous variables without expert domain knowledge and achieve state-of-the-art forecasting results [4]. Instead of guessing the dependency patterns ourselves, we can let a neural network find patterns inside the data for us.

During the 2010s, there have been many advances in building neural networks with many layers, often referred to as deep learning models. These advances have increased the expressive power of neural networks available to us when modeling dependencies between sequences in a multivariate time series [47]. Two types of deep learning models that are used today to model the interaction effect between time series are the recurrent neural network [39] and the Transformer [46]. The two model types mentioned in the last sentence are the ones we will explore in this thesis.

The temporal dynamics are how the relationship between multiple variables changes over time [40]. Imagine that we are modeling the total amount of ice cream sold in Sweden each week during a year. Suppose data available to us contains both ice cream sales and the number of tourists in Sweden every week during the last ten years. One could assume a stable linear relationship between the number of tourists and the amount of ice cream sold. However, the effect of tourists is probably very different depending on which week of the year we study. It could be that every tourist buys at least one ice cream in the summer and that in the winter, they buy none. What we just described is an example of temporal dynamics.

A model that deals with the temporal dynamics is all we need to make point estimates. We will use a Sequence-to-Sequence [9, 44] (seq2seq) model to do so. However, if we want to make probabilistic forecasts, we also need an emission model attached to the end of the temporal seq2seq model [31, 24, 37]. The temporal seq2seq model will still produce point estimates. However, these point estimates are used as values for the emission model parameters that describe the output probability density function. If the temporal seq2seq model is a painter, the emission model is a canvas with brushes. Together they are creating a painting with more details added at each time step. The emission model defines the limits of the probability distribution of a forecast. The temporal seq2seq model controls the parameters of the emission model over time, like a captain steers his ship out on the ocean over time.

We will model the output probability density function, the emission model, with a neural network. A particularly efficient way to model a density is with a type of neural network called normalizing flow [33]. A normalizing flow model is a generative model composed of several bijective transformations. Let us once more visit the canvas example. We could think of a normalizing flow model as a very flexible canvas with a wide range of brushes, and the temporal seq2seq model controls the brushes and how they paint the canvas.

The Transformer [46] was mentioned earlier as a possible temporal seq2seq model. This type of model was initially built with natural language processing in mind and has dominated the field of machine translation ever since its release. It is a sequence to sequence model and for machine translation that implies that a sequence of words goes in and a sequence of words goes out. For example, a sequence of English words is provided as input, and the Transformer hands back a sequence of German words. In the context of probabilistic forecasting, however, this would imply that a sequence of past values for a multivariate time series goes in and a sequence of emission model parameters comes out. If we consider the most simple of emission models, a univariate Gaussian, then the Transformer would produce an output sequence consisting of values for μ and σ, which fully defines the current shape of the emission model.

The Transformer can, together with an emission model, produce probabilistic forecasts, and one particular combination is the Transformer with a normalizing flow model called Masked Autoregressive flow (MAF) [32]. Combining the Transformer with the MAF gives you a new model called Transformer Masked Autoregressive Flow [37]. We have chosen to work with this combination because it has produced state-of-the-art forecast prediction scores [37] on datasets commonly used in the forecast academic literature, like Electricity [14], and Solar [28].

The Reformer [27] is a successor to the Transformer. Built with the Transformer as its starting point, the Reformer adds modifications that attempt to tackle some shortcomings of the Transformer. Long sequences are exceptionally costly, and the computational cost is O(n^2) for the Transformer, where n is the sequence length. The Reformer has a more nominal fee related to sequence length, and its time complexity is O($n\ log\ n$). Reversible layers are another feature that the Reformer adds, and in theory, this makes training the neural network cheaper in terms of memory per layer (See Section 3.2.2). The Reformer has lived up to its memory efficiency claims on particular examples such as image reconstruction and Wikipedia text forecasting [27].

This thesis will investigate how the Reformer performs in a multivariate time-series context. More specifically, we will take a state-of-the-art probabilistic forecasting model, Transformer MAF, and replace the Transformer with a Reformer model. We create a new model named the Reformer Masked Autoregressive Flow (RMAF). The Transformer suffers from a relatively high time complexity regarding the input sequence length, as described in the paragraph before. If the Reformer enables us to use longer input sequences, this can be useful in forecasting. One possible example where long input sequences are valid for probabilistic forecasting is climate forecasts. When researchers attempt to model something complex like climate over long periods, efficient memory use is essential to reduce hardware costs.

2 Research Question

The research question of this thesis is:

> Can the probabilistic forecasting model Transformer MAF be made more efficient through replacing the Transformer with a Reformer?

What does more efficiency entail in our scope? We will deem a model more efficient than another one if it can allocate less memory while producing forecasts of at least a similar quality.

3 Background

This section introduces and describes the two model categories relevant to creating our new model, the RMAF, defined in section 5.1. Section 3.1 is about Generative models, models that can generate data and sometimes also estimate the probability density of data [15]. Section 3.2 introduces Sequence-to-Sequence models [9, 44], a category that intuitively fits the problem of forecasting a future sequence given a past sequence.

3.1 Generative models

Generative models are models that can create data.

The most commonly used models answer questions about data and are called discriminative models. One example of a discriminative model is a neural network that detects cats and dogs in images. We will not examine discriminative models in detail in this thesis but instead, focus on generative models.

Generative models can sometimes both generate data and answer questions about data. When a generative model can do both, then the model is an explicit generative model. Otherwise, a model is an implicit generative model [18]. A popular set of implicit generative models are Generative Adversarial Networks (GANs) [21].

Formally, a generative model learns a joint probability distribution $P(X, Y)$ where X is the data, and Y is the set of labels. If there are no labels, the model approximates $P(X)$. To understand the last sentences, it can help to think of the discriminative model again. A discriminative model approximates $P(Y|X)$, for example, it outputs the probability of an image (X) containing a cat or a dog (Y).

In the next subsection, we will examine a subset of explicit generative models called Normalizing Flows [33].

3.1.1 Normalizing flows

In this section we will introduce the theory behind the main type of emission model discussed and used in this paper. Normalizing flows [33] is a flexible way of approximating probability distributions over continous random variables. Suppose that we want to model a probability distribution over variable $\mathbf{x}$, where $\mathbf{x}$ is a D-dimensional real-valued vector. The main idea of a flow-based model is to express $\mathbf{x}$ as a transformation $T : \mathbb{R}^D \mapsto \mathbb{R}^D$ of a real-valued vector $\mathbf{u}$ sampled from a simpler distribution $p_u(\mathbf{u})$:

$$\mathbf{x} = T(\mathbf{u}) \text{ where } \mathbf{u} \sim p_u(\mathbf{u}). \tag{1}$$

We refer to p_u as the *base distribution* of our flow-based model. There are two parameter vectors in this type of model, one for the transformation T that we denote ϕ and another one for the base distribution p_u that we denote ψ. The transformation T must be invertible and both T and T^{-1} must be *differentiable*. This type of transformation is known as a *diffeomorphism* and require that $\mathbf{u}$ must be D-dimensional as well [30]. Given these conditions the density of $\mathbf{x}$ can be obtained through a change of variables [38] [5]:

$$p_x(\mathbf{x}) = p_u(\mathbf{u})|\det\ J_T(\mathbf{u})|^{-1} \tag{2}$$

where $\mathbf{u} = T^{-1}(\mathbf{x})$ and det $J_T(\mathbf{u})$ is the determinant of the Jacobian matrix described below.

The Jacobian $J_T(\mathbf{u})$ is the $D \times D$ matrix of all partial derivatives of T:

$$J_T(\mathbf{u}) = \begin{bmatrix} \frac{\partial T_1}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_1}{\partial \mathrm{u}_D} \\ \vdots & \ddots & \vdots \\ \frac{\partial T_D}{\partial \mathrm{u}_1} & \cdots & \frac{\partial T_D}{\partial \mathrm{u}_D} \end{bmatrix} (\mathbf{u}). \tag{3}$$

Transformations that are both invertible and differentiable are composable. Given two transformations T_1 and T_2, the composition of $T_2 \circ T_1$ is also invertible and differentiable. The inverse and the determinant of the Jacobian are given by:

$$(T_2 \circ T_1)^{-1} = T_1^{-1} \circ T_2^{-1} \tag{4}$$

$$\det\ J_{T_2 \circ T_1}(\mathbf{u}) = \det\ J_{T_2}(T_1(\mathbf{u})) \cdot \det\ J_{T_1}(\mathbf{u}). \tag{5}$$

The consequence of this is that we can build complex transformations through composing multiple simpler (invertible and differentiable) transformations, and therefore without losing the ability to calculate $p_x(x)$.

The term "flow" refers to the gradual transformation, through the chain of transformations $T_1, ..., T_K$, of samples from $p_u(\mathbf{u})$ to samples from $p_x(\mathbf{x})$. The inverse flow, $T_K^{-1} \circ T_{K-1}^{-1} \circ ... \circ T_1^{-1}$, takes samples from the complex distribution $p_x(\mathbf{x})$ and "normalizes" them into samples from the base distribution which in general is chosen to be a multivariate normal.

3.1.2 Autoregressive flows

In section 3.1.1, we defined normalizing flows. Recall that the main idea is to model variable $\mathbf{x}$ as a transformation of variable $\mathbf{u}$. We will, in this section, look at the transformation T, and we will refer to T as a flow. More specifically, we will define a subset of possible flows called autoregressive flows [33].

Auto is a Greek prefix that translates to the English word *self.* Regression in our context refers to some function that produces real-valued output. An autoregressive flow uses properties inside the input itself to produce real-valued output.

We can think of an autoregressive flow as treating the input as sequential data. Consider, for example, if the input $\mathbf{x}$ is a vector that describes the average temperature during one week. Each dimension of the input vector describes the average temperature for one day of a week. Suppose we are to train a model to make temperature predictions. In that case, it is intuitive to assume that a dimension in the input vector depends on all preceding dimensions, and each dimension represents a time step. Modeling such a causal relationship between dimensions in a vector is called introducing an *autoregressive constraint.* The probability density of temperature variable x_i, at time step i, can be modeled as a conditional probability where it is conditioned on all the dimensions before it $\mathbf{x}_{1:i-1}$. The probability density of the whole input vector can then be calculated as the product of one-dimensional conditional probability densities for each individual dimension.

$$p(\mathbf{x}) = \prod_{i}^{n} p(x_i|\mathbf{x}_{1:i-1}) \text{ where } \mathbf{x} \in \mathbb{R}^n. \tag{6}$$

The conditional densities can be modeled with many possible distributions. The distribution we will use to model the conditional densities for the autoregressive flows is the Gaussian. More on that in the next section.

The autoregressive constraint ensures that the flow's Jacobian is triangular, and therefore the determinant of the Jacobian is simple to compute. One might wonder why we would want to treat the input vector $\mathbf{x}$ as sequential or ordered data even when that is not the case. After all, is not a flow with dependencies between every dimension to all other dimensions of its input vector more expressive and powerful than one where dependencies must abide by the autoregressive constraint? A flow that allows for dependencies between all dimensions can indeed be more expressive, but such a flow does not guarantee a tractable computation of the determinant of its Jacobian. A tractable determinant computation of the Jacobian is what makes a flow usable in practice.

3.1.3 Masked Autoregressive Flow

Masked Autoregressive Flow (MAF) [32] is one type of autoregressive flow. This flow is the type of flow we will use in our experiments. A MAF consists of Gaussian conditionals $p(x_i|\mathbf{x}_{1:i-1}) = N(x_i|\mu_i, \sigma_i)$ and Masked Autoencoders for Distribution Estimation (MADE) [17] blocks. One builds a MAF model by stacking MADE blocks. A MADE block is a neural network that produces hyperparameters for conditional densities. The hyperparameters are produced very efficiently through masking in matrix multiplication. In other words, the MADE blocks produce the parameters for our Gaussian distributions, (μ_i, σ_i). The details of MADE are outside the scope of this thesis; therefore, we recommend that the reader visit the following sources if they want to read more [32][17][24].

3.2 Sequence to sequence models

This thesis is about predictions which involve mapping a sequence of past events to a sequence of future forecasts. A very appropriate model type for predictions is the Sequence to Sequence type [9, 44].

A Sequence to Sequence (seq2seq) model maps a sequence of items $\mathbf{x}$ to another sequence of items $\mathbf{y}$ where the input length does not need to be the

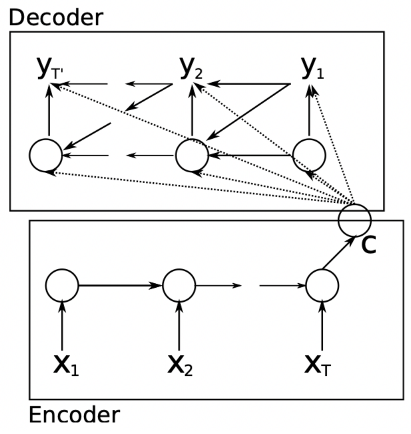


Figure 1: This image shows the general architecture of one of the first seq2seq models, RNN Encoder-Decoder. The idea of using an encoder part and a decoder part is essential, and one can find these parts in other seq2seq models further on in this section. The figure is from [9].

same as the length of the output. See Figure 1 for a visualization of the general architecture of a seq2seq model.

The first seq2seq models with neural networks were introduced in 2014 [9, 44]. Among the findings presented in [9, 44], one finding was that a seq2seq model could translate English sentences to French with state-of-the-art results. The experiment just mentioned is an excellent example because it involves mapping a sequence to another sequence where it is likely that the input is not of the same length as the output.

Encoder and Decoder

Typically we can decompose a seq2seq model into two parts, an encoder and a decoder [46, 9]. A seq2seq model can map sequences of varying lengths to one another because of the cooperation between the encoder and the decoder.

The encoder takes the input sequence and packs it into a fixed-length vector. The fixed-length vector serves as a compressed representation of the input sequence.

The decoder parses the output of the encoder and sends out an output sequence. The decoder does not need to know anything about the length of the input sequence. The decoder only cares about the length of the encoder output vector, which is a fixed length.

The first seq2seq models [9, 44] were built with two recurrent neural networks, one acting as the encoder and the second acting as the decoder. But recurrent neural networks are not required to construct a seq2seq model. A seq2seq model without recurrent networks is the Transformer [46] with special encoder and decoder parts explained below.

3.2.1 Transformer

We will now introduce a type of model that will act as a temporal seq2seq model later in our experiments. The Transformer [46] maps a sequence to another sequence and was invented with natural language processing in mind. Current top-of-the-line language models BERT [12], and GPT-2 [35] are built upon ideas of the original Transformer that we present in this section. Transformers are powerful tools for mapping sequences to sequences in general, not necessarily related to language. For example, transformer models can map a sequence of pixels to another sequence of pixels [8]. Another example involving both language and pixels is the task of image captioning; a transformer model can map a sequence of pixels to a sequence of words [11]. This general ability to map sequences to sequences makes transformers relevant to the forecasting domain, where we are interested in mapping a sequence of past values to a sequence of future values. The Transformer is a network architecture, mapping a sequence to a sequence based solely on the attention mechanism. There is no recurrence or convolution, which distinguishes it from architectures like the Recurrent Neural Network (RNN) and the Convolutional Neural Network (CNN). Since the release of the original Transformer paper [46], many new models build upon its ideas with new parts and modifications [12, 35]. We will focus on describing the original Transformer in this section.

Encoder stack and Decoder stack
The Transformer is made up of two distinct building *blocks*, the encoder stack, and the decoder stack (see Figure 2). There are six encoder *blocks* in the encoder stack and six decoder *blocks* in the decoder stack. The encoder block can be seen as a collection of two sub-blocks: a self-attention block and a feed-forward network. The decoder block is similar but can be interpreted as having three sub-blocks, a self-attention part, an encoder/decoder attention part, and finally, a feed-forward network.

Attention
The attention mechanism handles the relations between tokens in a sequence, e.g., words in a sentence or variable values in a multivariate time series. Suppose our input is a sentence with words. Each word is represented by an embedding vector of size d_{model} [19]. In this case, the first step of the attention mechanism is to let each word produce a query, key, and a value vector through linear transformations at the beginning of the attention head, see Figure 3. We refer to these resulting vectors as q, k, and v, respectively. The second step is to map all the queries, one vector q for each word in the sentence, in a matrix (Q) and the set of key-value pairs matrices (K and V) to an output. The resulting output is a weighted sum of the values where the weight to each part of the value vector is computed through a compatibility function of the query with the corresponding key (See Equation 7).

The attention implemented in the original Transformer is called Scaled Dot-Product Attention (See Figure 3 and Equation (7)). We will refer to this type of attention as Full Attention later on this thesis. Let S be the sequence length of the encoder stack input/output, T be the sequence length of the decoder stack input/output, d_k be the dimension of the keys, queries and value vectors. See figure 8 for more details about how Q, K and V are created. Q, K and V are then matrices of dimensions $S \times d_k$ (inside the encoder) or $T \times d_k$ (inside the decoder). QK^T is the compatibility function whose result is finally multiplied with the value matrix V to produce a weighted value matrix, and this is the result of the RHS in Equation (7). The division of QK^T with $\sqrt{d_k}$ is the operation that makes the result "scaled" and is put in place to stabilize gradients during training.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{7}$$

In Figure 3 to the right there is a variable **h**, find "Attention Head x h" , and it describes the number of attention layers used. Each attention layer is called an attention head. Several layers allow the model to jointly attend to information from different representation subspaces at different positions. The result of all the heads are concatenated and then linearly transformed into a result of the same dimension as the input dimension of the feed forward neural network.

Positional Encoding
The order of the tokens in the input sequence is important and that information is supplied to the transformer model through a positional encoding that takes place right before input vectors enter the encoder and decoder block. The positional encoding is a vector of the same dimension (d_{model}) as the input. Before the input vector goes into the first encoder or decoder block the positional encoding vector is added to the input. The positional encoding is calculated through the use of sine and cosine functions of different frequencies:

$$\text{PE}_{(pos,2i)} = sin(pos/10000^{2i/d_{model}})$$
$$\text{PE}_{(pos,2i+1)} = cos(pos/10000^{2i/d_{model}})$$

where *pos* is the position and i is the dimension.

What happens if the Transformer does not use positional encoding? We lose meaning encoded in the order of the tokens in a sequence. Let us consider a language example. Compare the sentence "I kicked a ball through the window." to "window. kicked a through the ball I". The first sentence makes sense because the order of the words is vital to parse the sentence's meaning.

3.2.2 Reformer

The Transformer [46] is a specific model, while transformers are a class of models inspired by the original Transformer. One such transformer model is the Reformer [27]. This model modifies the Transformer's residual blocks [23], the attention function, and the feedforward network.

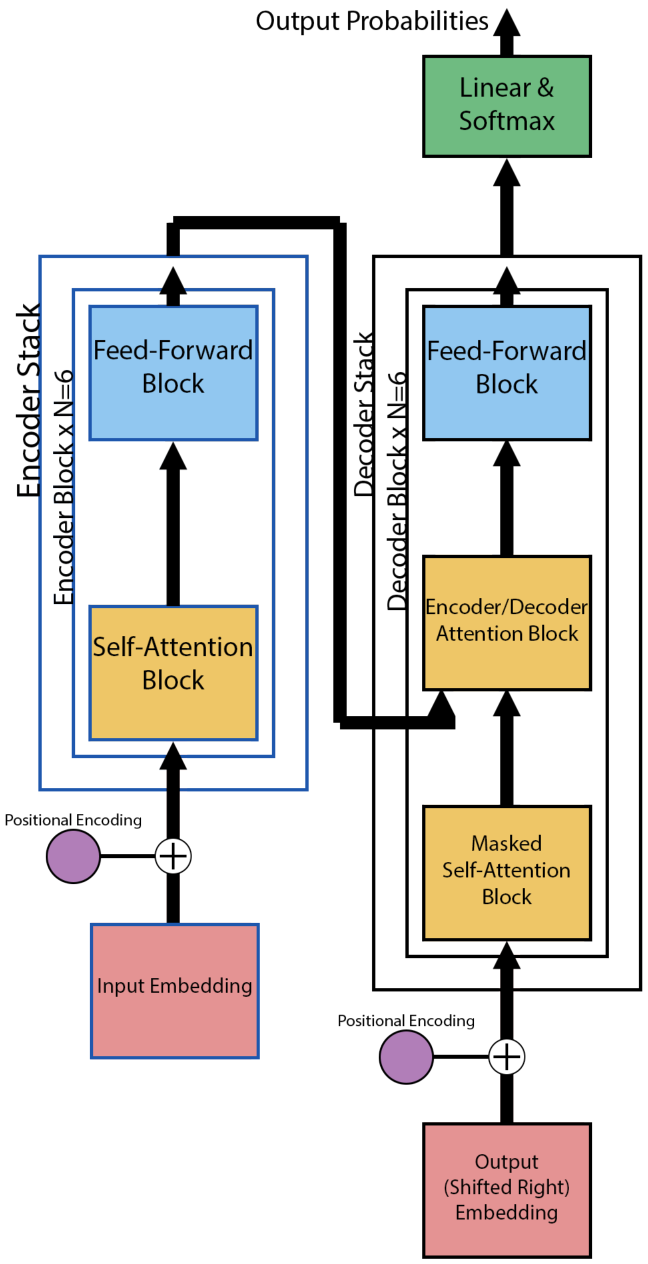


Figure 2: The Transformer's architecture. Here we have an overview of what is brought up in Section 3.2.1. This chart is inspired by [46].

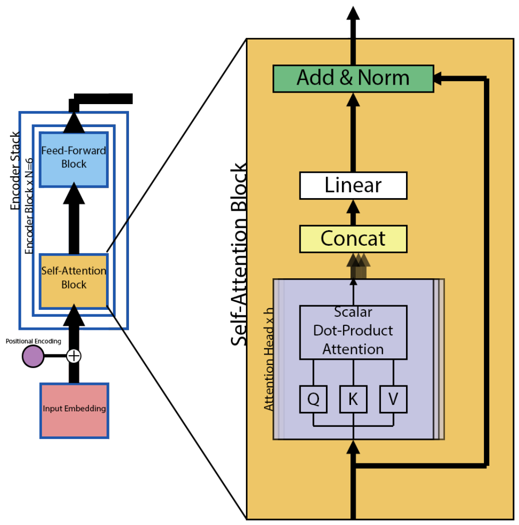


Figure 3: The Self-Attention Block. An essential part of the Transformer model is the attention computation which is visualized above, and described in Section 3.2.1.

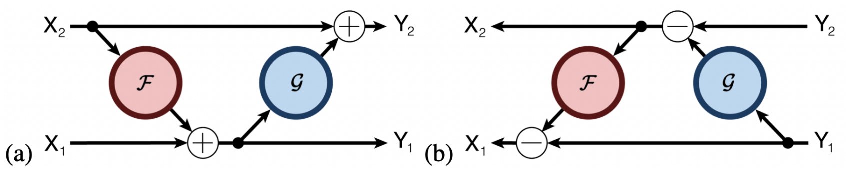


Figure 4: Here we see a forward pass (left) and a backward pass (right) through a reversible residual block. From [20].

We will describe the Reformer [27] in three parts. First, we will describe reversible residual blocks [20]. Secondly, we will unpack LSH Attention. LSH Attention is an approximation of Full Attention used in the Transformer. Finally, we will talk about chunking and what role it plays in the Reformer. Chunking is essentially just the idea that we can perform computations in chunks instead of all at once. Performing computations in chunks helps ease the memory burden for the GPU.

Reversible residual blocks

Let's first define what residual blocks [23] are and how we use them in the Transformer. After that, we will introduce reversible residual blocks [20] and describe what they enable us to do.

A residual block is a part of a neural network that utilizes skip connections. While the vanilla artificial neural network has layers that sequentially parse the input forward, a skip connection allows us to copy the information at a particular stage of the network and pass it along, thereby skipping layers to a future neural network layer. At this future stage of the network, we add the copied input to the transformed version of the input. We can see an example of a residual block in Figure 3, the Self-Attention block is a residual block.

What motivates the use of residual blocks is the stabilizing effect it has on deep neural networks. When we train a neural network with many layers, there are often fewer problems with gradients if we use residual blocks [22].

What is the point of making residual blocks reversible? The point is that this enables us to make a whole neural network reversible. If a neural network consists solely of residual blocks, then the entire network becomes reversible if we make each block reversible.

A reversible neural network does not need to save intermediate activation values during training. All we have to store is the network output. Then we can recreate specific intermediate layer activation values as we need them. This recreation is done by sending the output back into the network, but this time in reverse until we reach the intermediate layer whose activation values we require.

Let us explain how the reversible residual block works. We will first describe the equations of the residual block and then compare them with equations of the reversible residual block. If we look at the equation below, Equation 8, we see the calculations that happen during a forward pass of an input $\mathbf{x}$ through two residual blocks (See Figure 5).

$$\mathbf{y} = F(\mathbf{x}) + \mathbf{x} + G(F(\mathbf{x}) + \mathbf{x}) \tag{8}$$

We can compare the equation above with a forward pass through one reversible residual block which is defined below. We create $\mathbf{x}_1$ and $\mathbf{x}_2$ by copying the input tensor $\mathbf{x}$ twice, first into $\mathbf{x}_1$ and then $\mathbf{x}_2$.

$$\mathbf{y}_1 = \mathbf{x}_1 + G(\mathbf{x}_2 + F(\mathbf{x}_1)) \tag{9}$$

$$\mathbf{y}_2 = \mathbf{x}_2 + F(\mathbf{x}_1) \tag{10}$$

Is it not possible to go backward with a residual block? If the single transformation (F or G in our case) of the residual block is invertible, we can traverse in

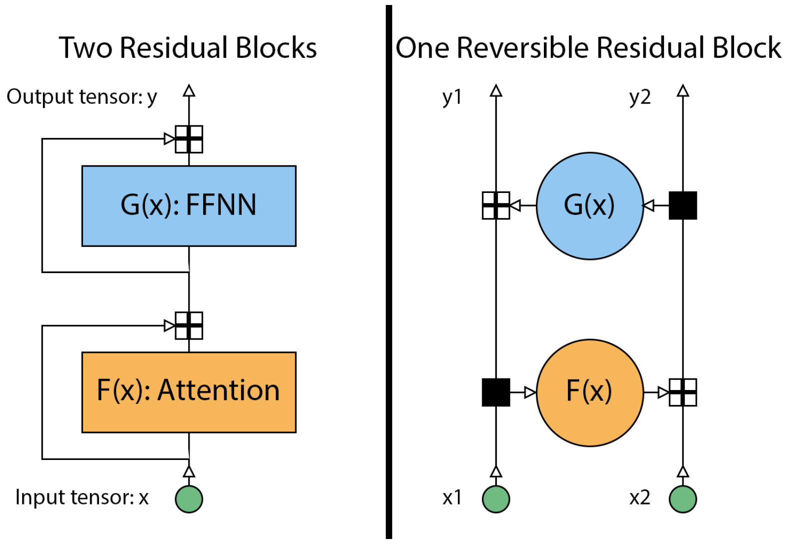


Figure 5: A visualization of residual blocks from the Transformer and the reversible residual block from the Reformer.

reverse. Invertibility is, however, far from certain for our transformation. This is why the construction of the reversible residual block is clever. It enables us to traverse this type of block in reverse without inverting F or G. Below we describe the equations for a backward pass through a reversible residual block.

$$\mathbf{x}_1 = \mathbf{y}_1 - G(\mathbf{y}_2) \tag{11}$$
$$\mathbf{x}_2 = \mathbf{y}_2 - F(\mathbf{y}_1 - G(\mathbf{y}_2)) \tag{12}$$
$$\tag{13}$$

Locality-Sensitive Attention (LSH Attention)
We will now talk about the type of attention function that is in the Reformer. It is a modification of the Full Attention that the Transformer uses (see Section 3.2.1).

Before we describe what LSH Attention is, let us talk about why one would want to modify its predecessor, the Full Attention. With Full Attention we let each token in an input sequence attend to all the other tokens in the input sequence. This attention strategy implies that the number of comparisons performed in a Full Attention block is the input sequence length squared. For many NLP tasks, this sequence length cost is manageable. For example, if you think about sequences you pass into Google Translate, the sequence length is seldom over a thousand.

There are, however, sequence-to-sequence tasks where the sequence length is of considerable size. One can use a transformer model to translate images to other images. A picture is a sequence of pixels. If this picture has the standard resolution Full HD, 1920x1080, this implies that the total number of pixels is 2

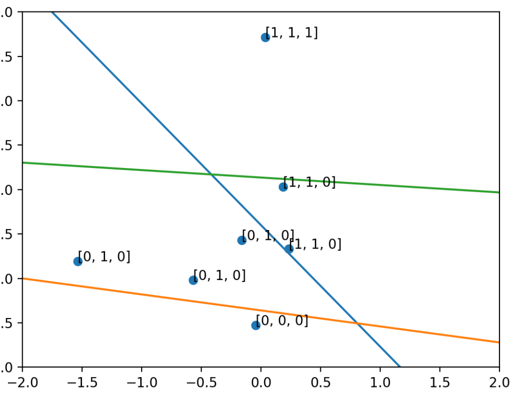


Figure 6: Locality-Sensitive Hashing example. Here we split our two dimensional input space with lines and then assign hash keys for each data point through asking the points about their relation to each line.

073 600. For cases like the one just mentioned, this gives us a sequence length in the range of millions for our transformer model.

When the sequence length is very long, e.g., the input sequence is an image or a sound clip, both spatial and computational complexity for Full Attention rapidly increase. This costly complexity is a fact because each token in the sequence is compared to all other tokens.

We can reduce the attention function's spatial and computational complexity by decreasing the number of tokens each token attends. One way of doing this is to view the attention function as performing a nearest neighbor search. In that case, we can first approximate a nearest neighbor search with hashing to put all the tokens into several buckets. Each token can then only attend to other tokens inside its bucket, which reduces the spatial and computational complexity.

Locality-Sensitive Hashing [43] (LSH) is a hashing scheme that approximates a nearest neighbor search. There are several ways of implementing this type of hashing. One simple example is to think of a sequence of tokens as dots in a two-dimensional plot. For this example, we can create buckets by drawing some arbitrary number of lines inside this plot. We can assign hashing keys to the tokens by asking about the relations they have to each line (see Figure 6).

We have a two-dimensional input space in the example above. In practice, we will most of the time deal with input spaces of a much higher dimension. Does the general idea with lines apply for the n-dimensional input space? It does with some modifications. We have to forget about lines and start thinking about hyperplanes instead.

A hyperplane is a linear subspace to some ambient space, and this subspace is of one less dimension than the ambient space. The ambient space in our specific case is the input space. Examples of hyperplanes are planes inside a

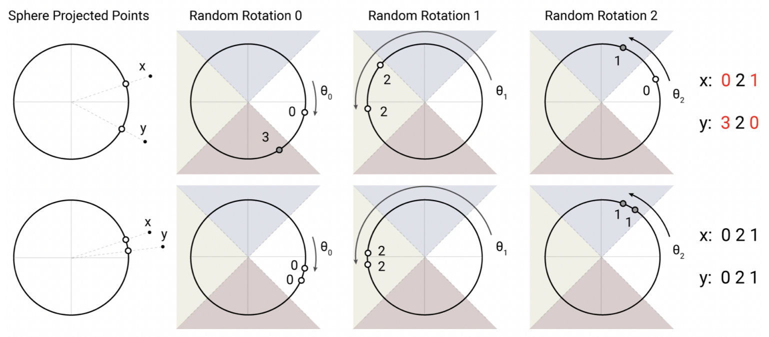


Figure 7: Angular Locality-Sensitive Hashing. Adopted from [27]

three-dimensional input space and lines inside a two-dimensional input space. In summary, lines help us partition two-dimensional spaces, and hyperplanes allow us to partition n-dimensional spaces, where n is some integer.

Since LSH is an approximation, sometimes input tokens that are very close to each other end up in different buckets. If you look at Figure 6 you will see an example of this. In the middle of the figure, there is a dot with the label [1, 1, 0]. This dot lies right on the blue line. The closest neighbor is the dot to its left, but it belongs to another bucket and has the label [0, 1, 0].

To reduce the approximation error of LSH, one can use several independent hashes and let them vote together on which hash key an input token should have. For our example with the two-dimensional input space, this could imply that we randomly generate three lines eight times to get eight independent hashes.

The LSH scheme used in the Reformer is called Angular LSH [2]. It involves projecting input points to a unit sphere, applying random rotations, and then partitioning the unit sphere with hyperplanes (see Figure 7). Let us talk about how to interpret the figure we mentioned in the last sentence. To the right in the figure, we see hash keys for tokens x and y. For the first row in the figure, we see that x and y have different hash keys. The second row contains an example where x and y receive the same hash key with angular LSH. To understand what the hash keys contain, study what happens to x and y in the random rotations seen in the middle of the figure. The Angular LSH has shown in various experiments to have a shorter query time than regular Hyperplane LSH, Hyperplane LSH in seen in our example in Figure 7, and this is one reason to prefer Angular LSH over Hyperplane LSH. For further details about Angular LSH, we recommend reading the paper by Andoni et al. [2] and studying the Reformer's implementation which can be found on GitHub [1].

Moving on, let us talk about how the Reformer modifies the concept of queries, keys, and values of Full Attention. The Reformer throws away the linear transformation that creates the keys and lets the queries serve as both queries and keys (See Figure 8). They show that this has little to no effect on

[1]https://github.com/google/trax/tree/master/trax/models/reformer

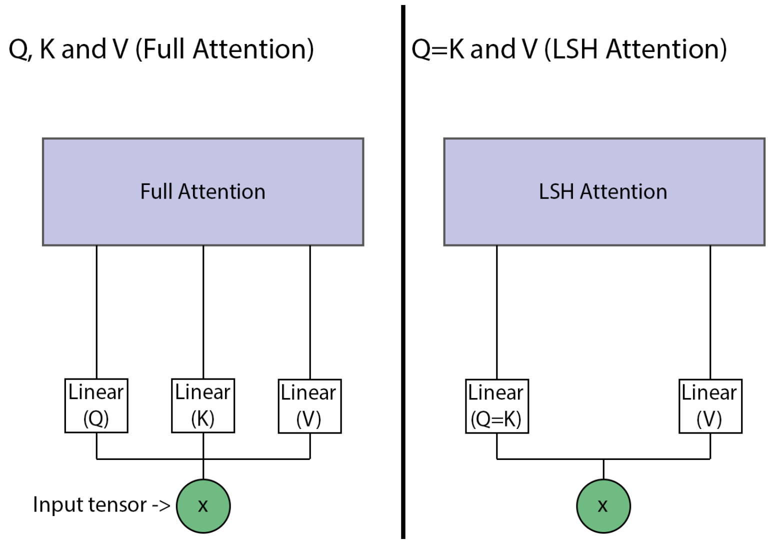


Figure 8: In LSH Attention, we throw away one linear transformation from Full Attention and call the result of one of the transformations queries=keys. This name is motivated because that result serves as both queries and keys in the attention computations.

the performance of the transformer model. As a result of throwing away a linear transformation in every attention block, we don't need to store as many weight parameters in memory. This implementation detail is not something we will focus on but it is important to know in order to understand one of the steps in Figure 9.

Finally, let us describe the move from Full Attention to LSH Attention in terms of equations. The starting point is the attention calcuation in the Transformer (See Equation 7). First we rewrite the equation we just mentioned into a form where we calculate the attention vector a_i for each token position in our input sequence. We exclude the scaling by d_k to make it easier to read the equation.

$$a_i = \sum_{j \in \mathcal{S}} \exp(q_i \cdot k_j - z(i, \mathcal{S}_i))v_j \qquad \text{where } \mathcal{S}_i = \{j : i \geq j\} \tag{14}$$

We introduce two symbols above. First of all, we let S represent the set of tokens that the query, q_i, for a position i can attend. Secondly, we let z denote the normalizing term in Softmax.

The main concern of LSH is the set S_i. To go from Full Attention to LSH Attention is first and foremost a matter of modifying S_i as we described earlier in this section when we talked about how Locality-Sensitive hashing is used to approximate the near neighbor search problem. We make the following change to transform equation 14 into LSH Attention.

Attention Type	**Spatial Complexity**	**Time Complexity**
Full Attention	$\max(bn_hld_k,\, bn_hl^2)$	$\max(bn_hld_k,bn_hl^2)$
LSH Attention	$\max(bn_hld_k,\, bn_hln_r(4l/n_c)^2)$	$\max(bn_hld_k,\, bn_hn_rl(4l/n_c)^2)$

Table 1: Spatial and time complexity of Full Attention and LSH Attention. We write b for batch size, denote l for sequence length, n_h for the number of attention heads, n_c for the number of LSH chunks, n_r for the number of hash repetitions. This table is the subset of a table from [27].

$$\mathcal{S}_i = \{j : h(q_i) = h(k_j)\} \text{ (Positions in single hash bucket)} \tag{15}$$

Where we let h denote the Locality-Sensitive hashing function. For more equations related to implementation details we refer the reader to the Reformer paper [27].

Chunking
To chunk a tensor is to split it, along some dimension, into several parts of equal size. One can chunk a tensor for many reasons.

We use chunking for both attention calculations and feed-forward calculations inside the Reformer. The reason why is that it reduces the memory burden for the GPU. The Reformer splits the attention input/feedforward input tensor into chunks and then sends over one or a few chunks at a time to the GPU. By sending chunks instead of the whole input tensor, we substantially reduce the amount of information that has to be stored in the GPU's memory at the same time. In the paragraphs that follow we will first talk about chunking in the context of feed-forward layers and then move on to describe how chunking is done inside attention blocks.

Let us first dissect chunking for feed-forward layers inside the Reformer. The authors of the Reformer motivate the use of chunking here as a way of tackling the considerable memory cost of the feed-forward layer. What is the root of this memory cost? It is not the amount of layers, this is managed by the reversible residual blocks. It is the inner dimension of the feedforward layer, which we will refer to as d_{ff}, that causes trouble. This dimension is in general much larger than the input dimension, d_{model}, to a feed-forward layer.

The result of chunking the feed-forward layers is that memory complexity is modified from being dependent on d_{ff} to d_{model} (See Table 2).

Moving on to chunking in the context of the attention blocks. Inside the LSH attention block chunking is performed after LSH has divided the input tokens into different hash buckets. Since we only want tokens to attend to other tokens inside the same bucket this serves itself up nicely for chunking where we only want to send one or a few chunks to the GPU at a time. Why don't we chunk the input into precisely the buckets that LSH produced? The Reformer does approximately this. There is a challenge here which lies in the fact that chunking creates partitions that are of equal size while LSH creates buckets of variying size. The Reformer handles this problem by sending more than one

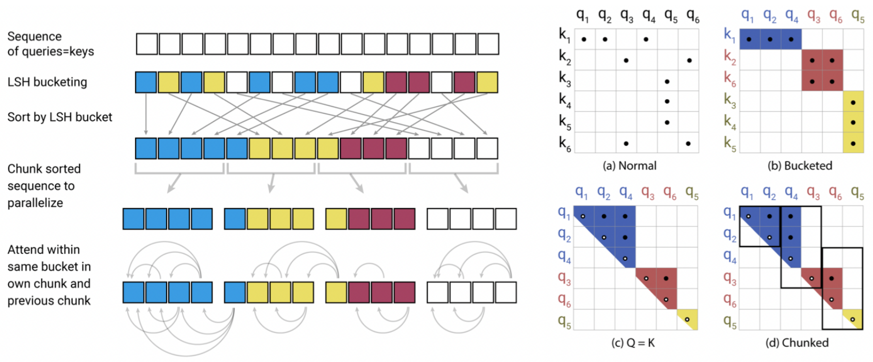


Figure 9: LSH Attention overview. From [27]

Model Type	Spatial Complexity	Time Complexity
Transformer	$max(bld_{ff}, bn_h l^2)n_l$	$(bld_{ff} + bn_h l^2)n_l$
Reversible Transformer	$max(bld_{ff}, bn_h l^2)$	$(bn_h ld_{ff} + bn_h l^2)n_l$
Chunked Reversible Transformer	$max(bld_{model}, bn_h l^2)$	$(bn_h ld_{ff} + bn_h l^2)n_l$
Reformer	$max(bld_{model}, bn_h ln_r c)$	$(bld_{ff} + bn_h n_r lc)n_l$

Table 2: Spatial and time complexity of transformer models. We write d_{model} for input token dimension and d_{ff} for feed-forward inner dimension and assume $d_{ff} \geq d_{model}$, b stands for batch size, l for sequence length, n_l for the number of blocks. We assume $n_c = l/32$ so $4l/n_c = 128$ and we write $c = 128^2$. From [27]

chunk at a time to the GPU and by allowing tokens to attend to both the chunk they belong to but also the chunk that came before it (See Figure 9).

To compare the Transformer to the Reformer in terms of complexities see Tables 1 and 2.

4 Related Work

This section will focus primarily on how the concepts brought up in Section 3 can be put together to enable probabilistic forecasting. In Section 3 we fleshed out concepts related to first the generative model (see Section 3.1) and then the temporal seq2seq model (see Section 3.2). Probabilistic forecasting is a subject that has been explored for many years, but we will only focus on modern deep learning-based approaches to forecasting in this thesis. If the reader wishes to know more about earlier classical forecasting methods, then we recommend reading Hyndman et al. [25].

In this section, we will first go through related deep learning-based probabilistic forecasting models. Finally, we will highlight the model that is the main inspiration for our work in this thesis. In this context, deep learning-based models mean that we are using a neural network to model the temporal seq2seq (e.g., an RNN or a Transformer model) and the generative model (e.g., a normalizing flow model).

Let us introduce essential notation regarding probabilistic forecasting. If we have a multivariate time series of dimension N, then let i denote the index of a specific time series ($i \in [1, N]$). Let the value of a time series i at a specific time t ($t \in [1, T]$) be denoted as $z_{i,t}$. Let the values for all time series at time step t be $\mathbf{z}_t$. The essential goal of multivariate probabilistic forecasting is then to model

$$p(\mathbf{z}_{t_0:T} | \mathbf{z}_{1:t_0-1}, \mathbf{x}_{1:T}) \tag{16}$$

where t_0 is the first time series index that we wish to make a prediction for and $\mathbf{x}_{i,1:T}$ are the covariates that contain information about the several time series sequences that is assumed to be known. The covariates most often contain simple information related to date, time, and time series frequency. In the paragraphs that follow, we will refer to the time range $[1, t_0 - 1]$ as the conditioning range and $[t_0, T]$ as the prediction range.

4.1 DeepAR: Probabilistic Forecasting with Autoregressive Recurrent Networks

DeepAR [41] was one of the first probabilistic forecasting models to properly utilize the power of deep learning and at the release of this model it improved upon the former state-of-the-art dataset accuracy results.

This model consists of a likelihood function whose parameters are set by a recurrent neural network. The likelihood functions explored where Gaussian likelihood and Negative binomial likelihood. The recurrent neural network involves the use of LSTM cells.

$$p(\mathbf{z}_{i,t_0:T}|\mathbf{z}_{i,1:t_0-1}, \mathbf{x}_{i,1:T}) \tag{17}$$

DeepAR fits a global (i.e. shared) sequence-to-sequence model to a collection of time series but can only generate independent predictions for the different time series. This implies that the multivariate probabilistic forecasting goal mentioned earlier, see Equation 16, is not achieved but rather something a bit simpler (see Equation 17).

4.2 High-Dimensional Multivariate Forecasting with Low-Rank Gaussian Copula Processes

Salinas et al. introduced a model that both fits a global model to a collection of time series and generates dependent predictions [42]. A forecast for time step t is now expressed in a vector describing the values of all the time series at time step t instead of a single one. Before we spell out the equation for this model, let us introduce some notations and remove some for clarity. First, let $\mathbf{h}_t$ be the hidden state of an LSTM-RNN model for time step t. Second, we will omit the covariate vector from the equation because it is not of interest to this thesis. Third, let $\mathbf{f}$ be a function where each element f_i is an invertible mapping. The purpose of each f_i is to take the time series i and transform it so that marginally it follows a standard normal distribution.

$$p(\mathbf{z}_t|\mathbf{h}_t) = \mathcal{N}(\mathbf{f}(\mathbf{z}_t)|\mu(\mathbf{h}_t), \Sigma(\mathbf{h}_t)) \tag{18}$$

The multivariate probabilistic forecasting goal (See Equation 16) is now adequately modeled. The models presented below will follow in the footsteps of this one to generate predictions with dependencies between the output values for the different time series.

4.3 Multi-variate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows

Rasul et al. [37] built upon the ideas of Salinas et al. [42] (mentioned above) and introduced the idea of replacing the low-rank gaussian process with a normalizing flow model like Real-NVP [13] or MAF [32]. Real-NVP is a normalizing flow model and can be thought of as a predecessor to MAF. With this modification of the emission model in place, they observed better predictions over datasets commonly used in the academic literature. Another idea that Rasul et al. introduced was to model the multivariate temporal dynamics of the time series with a Transformer model instead of an RNN. They showed that the Transformer component could utilize parallelization better during training than the RNN. They observed better prediction results with the Transformer based models than the RNN based models. See figure 10 for an overview of Transformer conditioned Real-NVP.

Let us wrap up this section by comparing this thesis to the work of Rasul et al. Recall that one of the models presented by Rasul et al. was the Transformer

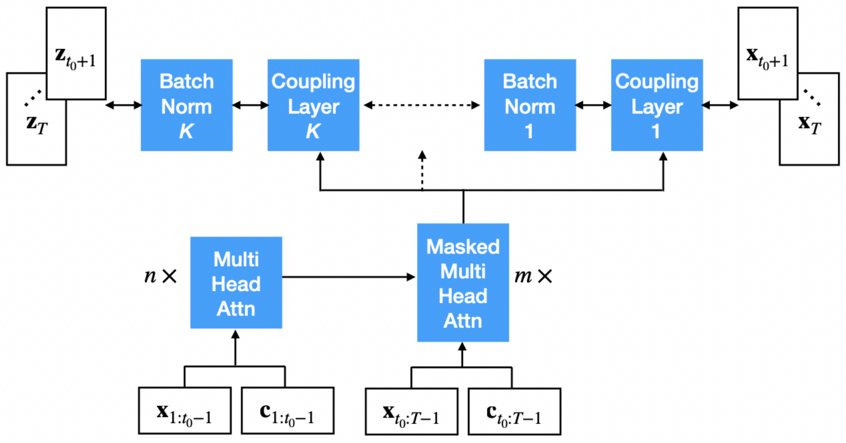


Figure 10: A visualization of the Transformer Real-NVP [37]. This model is very similar to the Transformer MAF. This image shows how the sequence to sequence model, the Transformer, controls the normalizing flow model, Real-NVP.

MAF [37]. This model uses the Transformer to condition a MAF. MAF will also be used in our work. We will, however, replace the Transformer with the Reformer (see Section 3.2.2). The Reformer claims to be more memory efficient than the Transformer, and we will investigate if that claim is valid in the probabilistic forecasting context.

5 Method

In this section, we will describe two things. First, we will describe the new model we introduce, the Reformer Masked Autoregressive Flow (RMAF), and motivate its design choices. Second, we explain and justify our evaluation criteria.

5.1 Reformer Masked Autoregressive Flow

We present a new probabilistic forecasting model called Reformer Masked Autoregressive Flow (Reformer MAF). Earlier in this thesis, we brought up a generative model known as Transformer MAF. To explain how we create the Reformer MAF, we will first talk about the Transformer MAF.

The Transformer MAF can be thought of as two connected lego blocks. One lego block is the Transformer, and the other is the Masked Autoregressive Flow (MAF). Each of these lego blocks has a function by itself (See Section 3.1.3, 3.2.1). But if we combine the two lego blocks, we get a new shape with a unique function (See Section 4).

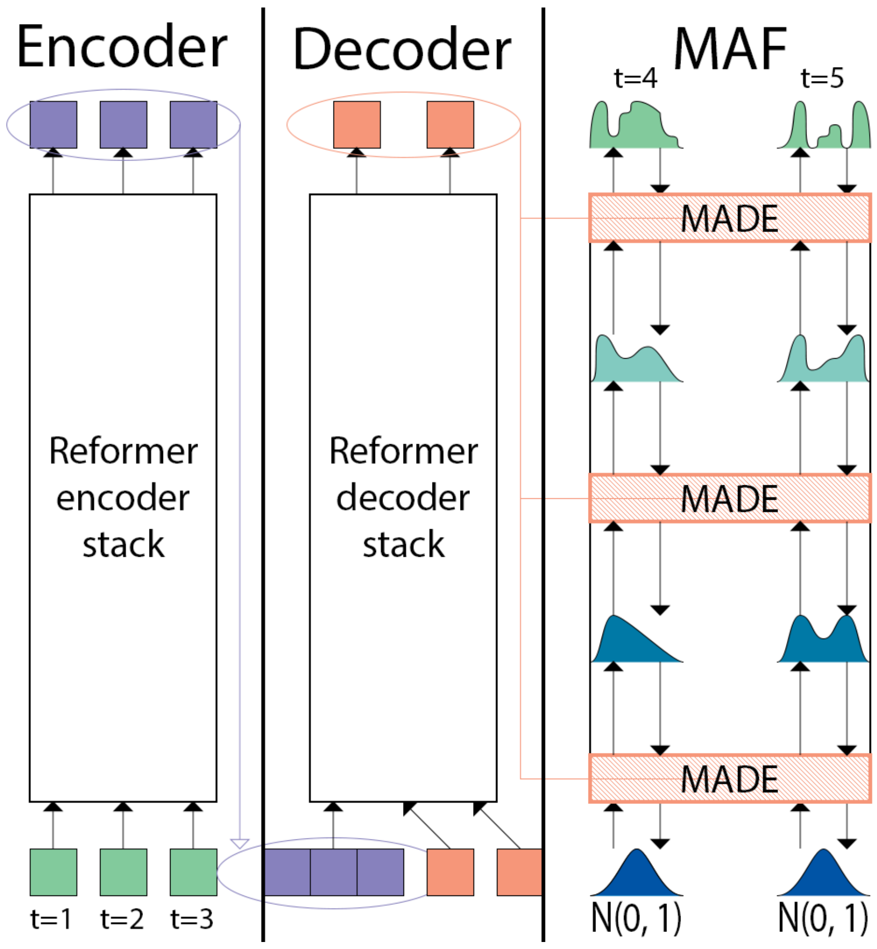


Figure 11: Reformer MAF visualized with a simple example. In this example, we put in three data points from the past, and our goal is to make a probabilistic forecast for two time units into the future. To the left in this image, we see a Reformer encoder-decoder model, which is connected to a MAF model that sits to the far right. First, the past data goes through the encoder and is encoded. Second, the decoder takes the encoded past data points and produces forecasts. Third, the forecast for time unit four is produced from only the encoded data output of the encoder. Forth, the forecast for time unit five is produced from the encoded past and the forecast for time unit four. Fifth, the encoded history and the two predictions are sent through the decoder, and the decoder produces an end-of-sequence token which marks the end of its output sequence. The forecasts of the decoder are projected to parameters of the layers of the MAF, which controls the shape of its probabilistic output.

Imagine that we remove the Transformer lego block and replace it with another lego block, the Reformer. The result of this is the Reformer MAF.

A low-resolution view of the Reformer MAF has now been established. As we continue, we will provide more details regarding the Reformer MAF. First, we will explain why we replaced the Transformer with the Reformer, and second, we will elaborate on details that made it possible.

In the paper that introduced the Transformer MAF [37], the authors recommended that further research be done to see if other transformer models could improve their work. They do not mention any specific transformer model in the later versions of their paper (there are three as we write this). But the first version of [37] mentioned the Reformer and that future work could be done to explore its effects on probabilistic forecasting. That recommendation was one inspiration for the work done in this thesis.

Why did the authors of the Transformer MAF recommend that further research be done with the Reformer? The stated reason was that the Reformer could improve memory efficiency over what they achieved with the Transformer.

Framework details

The Transformer MAF is implemented with PyTorch. In order to switch out the Transformer for the Reformer with as much ease as possible, we decided to use an implementation of the Reformer that also uses PyTorch. With every piece of the model built with the same framework, we can with relative ease connect the Reformer with the MAF and train and use them as one single unit.

The time series libraries we used to build Reformer MAF are PyTorchTS [34] and GluonTS [1]. Amazon built GluonTS, and it contains many of the standard operations that one needs to make forecasts that are either point estimates or probabilistic. Zalando built PytorchTS, and it started as GluonTS with PyTorch as its computational engine. PytorchTS has, since its inception, grown more independent of GluonTS. See Figure 12 for an overview of the libraries we used to build Reformer MAF.

Encoder-decoder Reformer

Earlier we introduced the concept of encoders and decoders and how they are connected to seq2seq models, see Section 3.2. The Reformer can be used either as a decoder or an encoder-decoder model. The Transformer was built solely as an encoder-decoder. We decided to work with the version of the Reformer that is an encoder-decoder to keep things as similar as possible as we swapped out the Transformer for the Reformer. An overview of our new model can be seen in Figure 11.

5.2 Evaluation

How did we know if model A was an improvement over model B? First of all, we had to define what model qualities we valued. The Hypothesis was that Reformer MAF is more memory efficient than the Transformer MAF. To put this claim to the test, we measured memory usage, forecasting ability, and

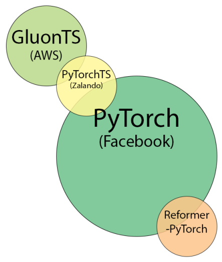


Figure 12: We built the Reformer MAF primarily on the libraries in the image above. The libraries with the suffix TS contain time series-related methods. PyTorchTS also includes a MAF implementation. Reformer-PyTorch has the Reformer implementation that we used. All of the libraries above are available to the public on GitHub.

failures during our experiments.

First of all, the amount of memory allocated on the GPU. The memory allocated is measured in bytes. Second, we measure forecasting ability, and the metric we choose for this job is Continuous Ranked Probability Score (CRPS) which you can read about below. Third, the number of failed trials. A failed trial is a trial where the model produced by the training process has bad parameter values that cause many inputs to produce NaN (Not a Number) or Infinite outputs (See Figure 21 and 20).

5.2.1 Continuous Ranked Probability Score

Continuous Ranked Probability Score (CRPS) [29] is a generalization of the mean absolute error metric to the probabilistic case. It assumes the perfect probabilistic prediction is a probability distribution where all of the mass is allocated to the real outcome and measures how far off the forecast is from that outcome.

$$CRPS(F,x) = \int_{-\infty}^{\infty} (F(y) - \mathbf{1}(y-x))^2 \tag{19}$$

Where F is the predictive density and x is an observed real outcome. In Figure 13 we have a fictional example that helps the reader grasp the intuition behind this metric.

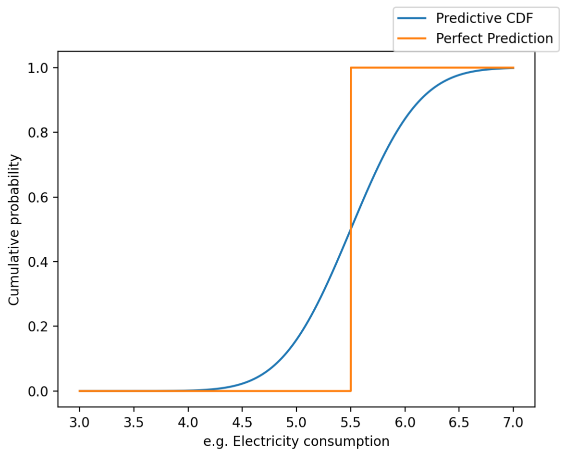


Figure 13: If we have a probabilistic prediction (blue line) and know the observed outcome is 5.5, as in this example, then CRPS is a score that measures how similar the probabilistic prediction is to a perfect prediction (orange line).

6 Experiments and Results

All experiments were performed on a computer with operating system Manjaro, GPU NVIDIA GeForce RTX 2800 Ti, and CPU Intel Core i9-9900K 3.60Ghz.

We built our experiments with code from two Github repositories. First, we used the published code from the Transformer MAF paper [2] that is built upon the probabilistic forecasting framework GluonTS [1]. Second, we used code that implements the Reformer model with PyTorch[3]. We chose to combine these two implementations as they are released in the same framework.

6.1 Datasets

There are two datasets used in this Thesis, Electricity [14], and Solar [28]. These two datasets are often used in the literature [37, 42, 1, 41] and facilitate easy comparison with other research.

6.1.1 Electricity

The Electricity dataset [14] contains 370 time series of electricity consumption measured in kilowatts (kW). We can visualize this dataset as one timestep column with 370 electricity consumption columns to its right. There are several modified versions of this dataset. We've worked with the version of the dataset that was used in [37] as well as [42]. This version of the dataset has an hourly frequency and 5790 timesteps.

[2]https://github.com/zalandoresearch/pytorch-ts
[3]https://github.com/lucidrains/reformer-pytorch

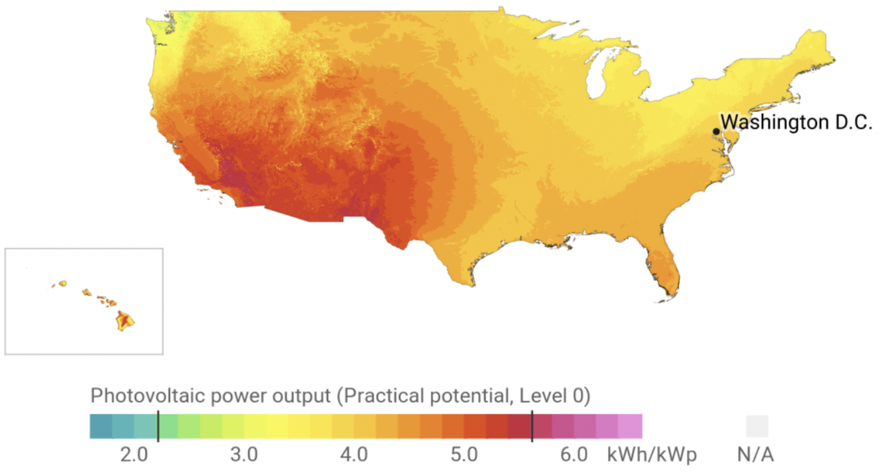


Figure 14: The Solar dataset contains values about photovoltaic production. The image above is related to another dataset but provides a good visualization of the type of data the Solar dataset contains. Image is from [3].

6.1.2 Solar

The Solar dataset [28] contains 137 time series of photovoltaic production measured in kW per hour (kWh). Each time series corresponds to a solar monitoring station in United States, Alabama. The National Renewable Energy Laboratory, which serves under the U.S. Department of Energy, published this dataset. Just like the Electricity dataset has several versions, so is the case for the Solar dataset. We've worked with the version of the dataset that was used in [37] and [42]. This version has an hourly frequency and 7009 timesteps. A visualization of photovoltaic power can be found in Figure 14, but it should be noted that this visualization is made from another dataset.

6.2 Experiments - overview

In the three subsections that follow, we describe the three experiments that we performed. For all the experiments, we perform several trials. In this context, our definition of a trial is that we train and evaluate a probabilistic forecasting model (most often Transformer MAF or Reformer MAF). Here is a high level overview of a trial:

1. Create and train a probabilistic forecasting model (TMAF, RMAF etc)
2. Make several forecasts with the trained model
3. Evaluate the forecasts made

Data set	LSTM Real-NVP	LSTM MAF	Transformer MAF	LSH Transformer MAF	Reformer MAF
Solar	**0.321** ± 0.016	**0.361** ± 0.029	0.394 ± 0.036	0.447 ± 0.032	0.394 ± 0.036
Electricity	**0.022** ± 0.002	**0.024** ± 0.004	0.025 ± 0.002	0.03 ± 0.002	0.025 ± 0.001

Table 3: Test set CRPS_{sum} comparison (lower is better) of models from Rasul et al. [37] and our models LSH Transformer MAF and Reformer MAF. The two best method results are in bold and the mean and standard deviation of the methods are obtained by rerunning them 30 times and then choosing the 20 runs with lowest CRPS_{sum}.

In general, we perform around 20 trials for most model configurations. The exact number of times we perform a specific trial can be found in the caption to the tables for each experiment.

6.3 Experiment 1 - Reproduction and benchmarking

Our model the Reformer MAF is heavily inspired by the Transformer MAF [37]. In our first experiment we attempt to reproduce some results presented in their paper with the three models introduced in their paper which are LSTM Real-NVP, LSTM MAF and Transformer MAF. We use hyperparameters presented in the Transformer MAF paper and its related GitHub repository.In addition to the reproduction we attempt to train our new models with similar hyperparameters in order to benchmark them against earlier models. The new models we use here are the Reformer MAF and a hybrid between the Reformer MAF and the Transformer MAF that we choose to call LSH Transformer MAF.

Details on hyperparameters
We repeat the experiment 20 times and then compute the mean and standard deviation of the 20 resulting prediction evaluations.

Results
Table 3 displays the CPRS-Sum results after training models and running evaluation over predictions. Lower is better. The Reformer MAF and the Transformer MAF produced forecasts of close to exactly the same quality. If you read those two models' mean and standard deviation results you will see that they are almost the same. The LSH Transformer performed worse than the Reformer MAF, especially on the Solar dataset. The RNN based models, the ones that use LSTM, performed best but are not far ahead of the Transformer MAF and the Reformer MAF.

6.4 Experiment 2 - Layer count and reversibility

We designed this second experiment to reveal the effect that reversible residual blocks have on memory allocation. The Reformer MAF has LSH attention turned off here to focus the experiment on the impact of the reversibility (here, Reformer MAF uses Full Attention as the Transformer MAF does). The ex-

Model	**Layer Count**	**CPRS-Sum**	**Std**	**Failures**
TMAF	3	0.064	0.168	1
TMAF	4	0.032	0.005	1
TMAF	5	0.031	0.003	2
TMAF	6	0.033	0.007	0
RMAF	3	-	-	25
RMAF	4	-	-	25
RMAF	5	-	-	25
RMAF	6	-	-	25

Table 4: Prediction performance after Early Stop training based on validation loss. Transformer MAF (TMAF) vs Reformer MAF (RMAF). We ran the same experiment with each configuration 25 times. The column Filtered count describe how many of these 25 times that we experience numerical instability, exploding and/or vanishing gradients that lead to flawed results.

periment included two training strategies that result in two different types of trials. First, we trained the Transformer MAF and the Reformer MAF with an early stopping algorithm where the model's training would continue until the validation loss stopped increasing. Secondly, we tried training the models with a fixed amount of epochs. We tried using both early stopping and a fixed amount of epochs because the Transformer MAF paper did not state if they used early stopping or not.

We choose to use only the dataset Electricity for this experiment and focus on measuring probability forecast scores (CRPS), training process failures, and GPU memory allocation.

6.4.1 Early stopping trials

The results of the early stopping trials can be seen in Table 4. The idea behind early stopping is to stop the training process when our model's performance no longer improves when evaluated over a validation data set. Early in this context implies that we want to stop the training before the model overfits to the training data set. We calculate the model's performance (with the loss function) during training after each epoch to make early stopping happen. The best loss value achieved is saved as a high score that the model should repeatedly beat. We stop the training process when the model has failed to produce a better best loss value three times consecutively. The loss function mentioned in the paragraph above is an approximated log-likelihood function computed by the flow-based emission model [37]. We chose not to investigate the memory allocated during this experiment because the RMAF experienced failure every trial.

6.4.2 Fixed epoch counts trials

The results of these trials consist of two tables (See Tables 5a, 5b) and two box plots (See Figures 19b, 19a). The tables describe CRPS-sum and times of failure. The box plots illustrate the amount of GPU memory allocated during the trials. These trials use a more straightforward strategy than the trials in the subsection above. We simply train the Reformer MAF and the Transformer MAF for a fixed amount of epochs and then evaluate probabilistic forecast score (measured in CRPS-sum), the number of times we experience failures, and sample memory allocation.

6.4.3 Results summary

The early stop training strategy worked for the TMAF, which led to models that could forecast CRPS scores similar to the ones in experiment 1. The RMAF combined with the early stop training strategy led to training failures where the trained RMAF model produced NaN values or infinite values as output.

The CPRS histograms for the TMAF and RMAF, see Figure XXX, show similar distributions.

The Failure histograms for the TMAF and RMAF, see Figure YYY, show different distributions. The TMAF histogram has most trials in the range of 0 to 1 failures, while the RMAF histogram has most trials in the range of 6 to 8.

The fixed epoch training strategy worked better than the early stop training strategy for the RMAF, seeing as it didn't work at all in the second case. After training, the RMAF would more often than not produce valid forecasts. In terms of CRPS, TMAF and RMAF made forecasts of similar quality. However, the TMAF had the most stable results with fewer failed trials and a lower spread in the CRPS scores.

The main result of the second experiment was that the RMAF induced a significant memory overhead cost compared to the TMAF, see (See Figures 19b, 19a). The RMAF allocated more memory than the TMAF during the fixed epoch trials. The RMAF allocated about 7-8 times more memory than the TMAF.

Epoch count	**Layer count**	**CRPS-Sum**	**Std**	**Failures**
5	3	0.033	0.006	1
10	3	0.051	0.036	6
15	3	0.043	0.020	3
20	3	0.066	0.099	3
5	4	0.057	0.117	0
10	4	0.079	0.108	8
15	4	0.069	0.109	9
20	4	0.133	0.217	6
5	5	0.032	0.006	2
10	5	0.080	0.153	5
15	5	0.036	0.009	6
20	5	0.044	0.030	6
5	6	0.034	0.021	1
10	6	0.118	0.173	3
15	6	0.074	0.100	7
20	6	0.103	0.212	4

(a)

Epoch count	**Layer count**	**CRPS-sum**	**Std**	**Failures**
5	3	0.029	0.002	0
10	3	0.033	0.003	0
15	3	0.034	0.004	2
20	3	0.032	0.003	2
5	4	0.054	0.058	1
10	4	0.029	0.004	0
15	4	0.066	0.155	1
20	4	0.030	0.003	4
5	5	0.070	0.071	0
10	5	0.075	0.080	1
15	5	0.052	0.079	0
20	5	0.064	0.107	2
5	6	0.094	0.061	0
10	6	0.088	0.076	2
15	6	0.078	0.075	3
20	6	0.051	0.053	1

(b)

Table 5: Transformer MAF and Reformer MAF trained with fixed amount of epochs, prediction performance evaluation. The two first columns from the left describe the configuration and the other three are the results. We ran the same experiment with each configuration 20 times. The column Filtered count describe how many of these 20 times that we experience numerical instability, exploding and/or vanishing gradients that lead to flawed results.

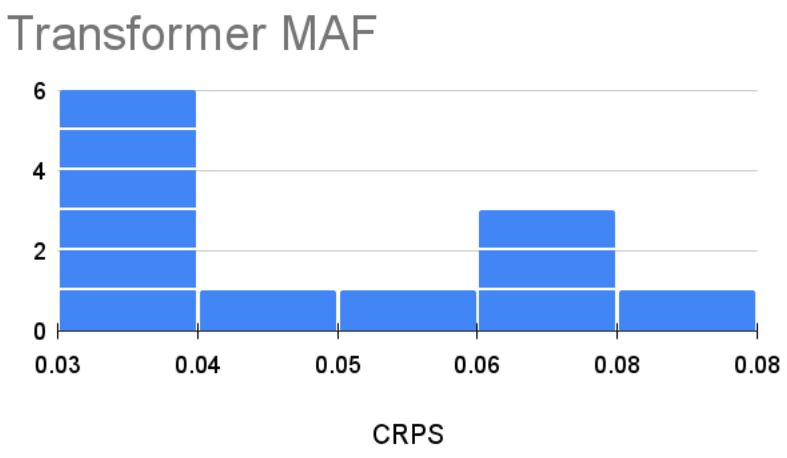


Figure 15: This histogram shows us the CRPS distribution of our trials for the Transformer MAF in experiment 2 with the fixed epoch training strategy. The histogram summarizes data which can be read in Table 5. The y-axis is the number of trials.

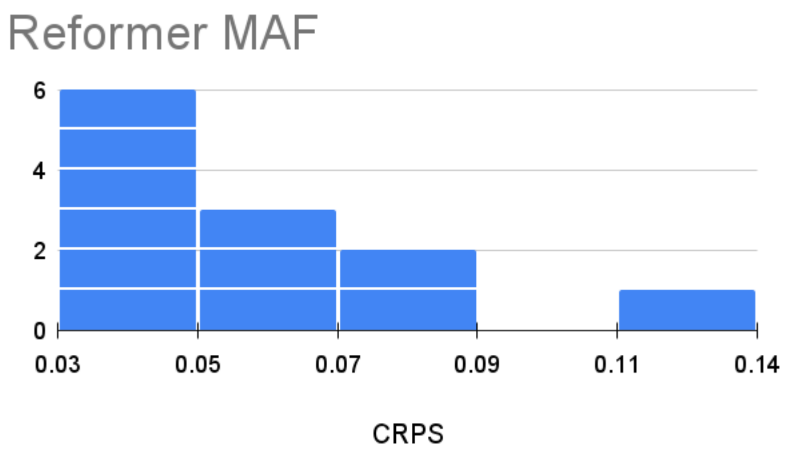


Figure 16: This histogram shows us the CRPS distribution of our trials for the Reformer MAF in experiment 2 with the fixed epoch training strategy. The histogram summarizes data which can be read in Table 5. The y-axis is the number of trials.

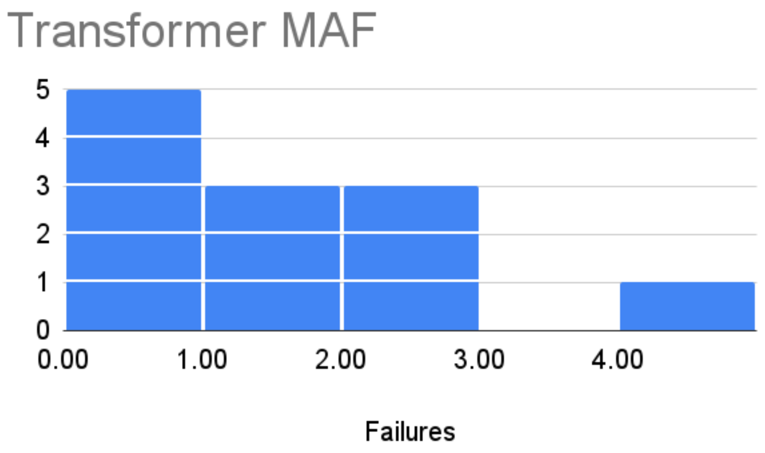


Figure 17: This histogram shows us the distribution of failures in our trials for the Transformer MAF in experiment 2 with the fixed epoch training strategy. The histogram summarizes data which can be read in Table 5. The y-axis is the number of trials.

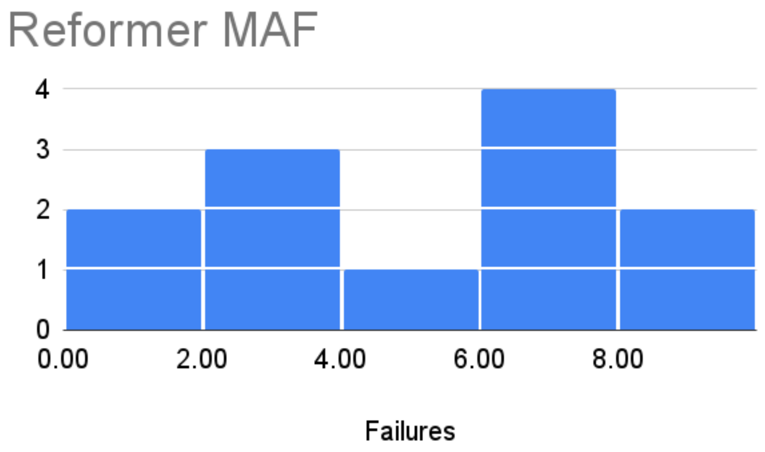
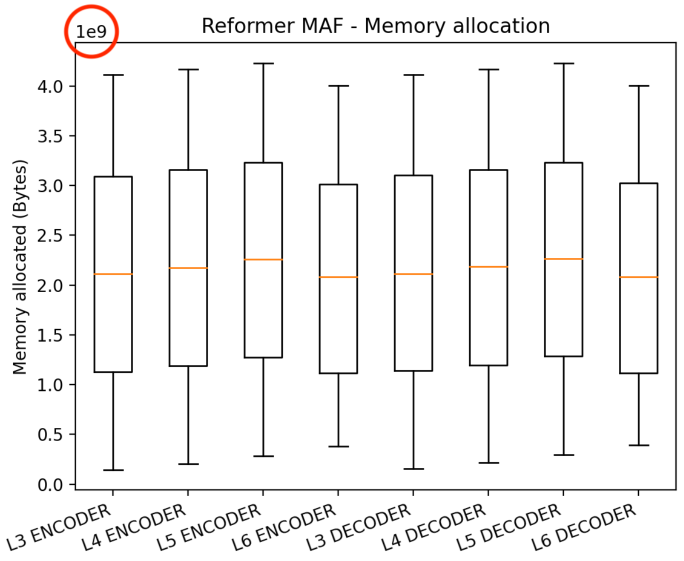
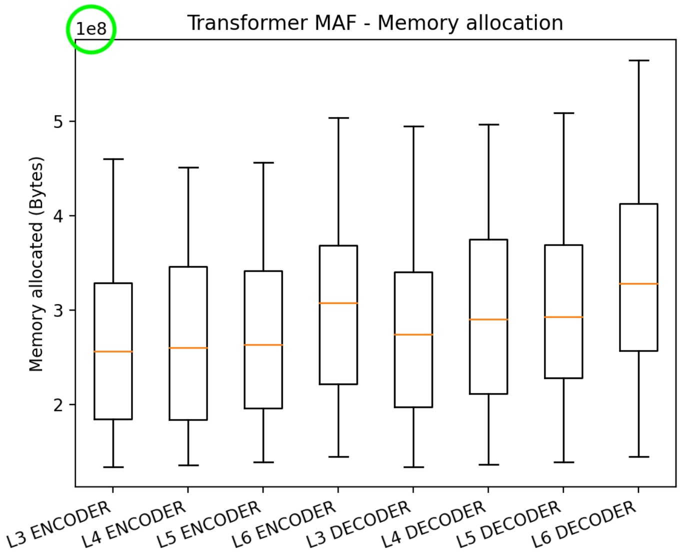


Figure 18: This histogram shows us the distribution of failures in our trials for the Reformer MAF in experiment 2 with the fixed epoch training strategy. The histogram summarizes data which can be read in Table 5. The y-axis is the number of trials.



(a)



(b)

Figure 19: A summary of memory allocation during training of the Reformer MAF and Transformer MAF for all experiments with fixed amount of epochs. The amount of memory allocated is sampled after a pass through the encoder stack and after a pass through the decoder stack. Note the unit size of the y-axis, it it 10^8 for the Transformer MAF graph whereas the Reformer MAF graph has a unit size of 10^9.

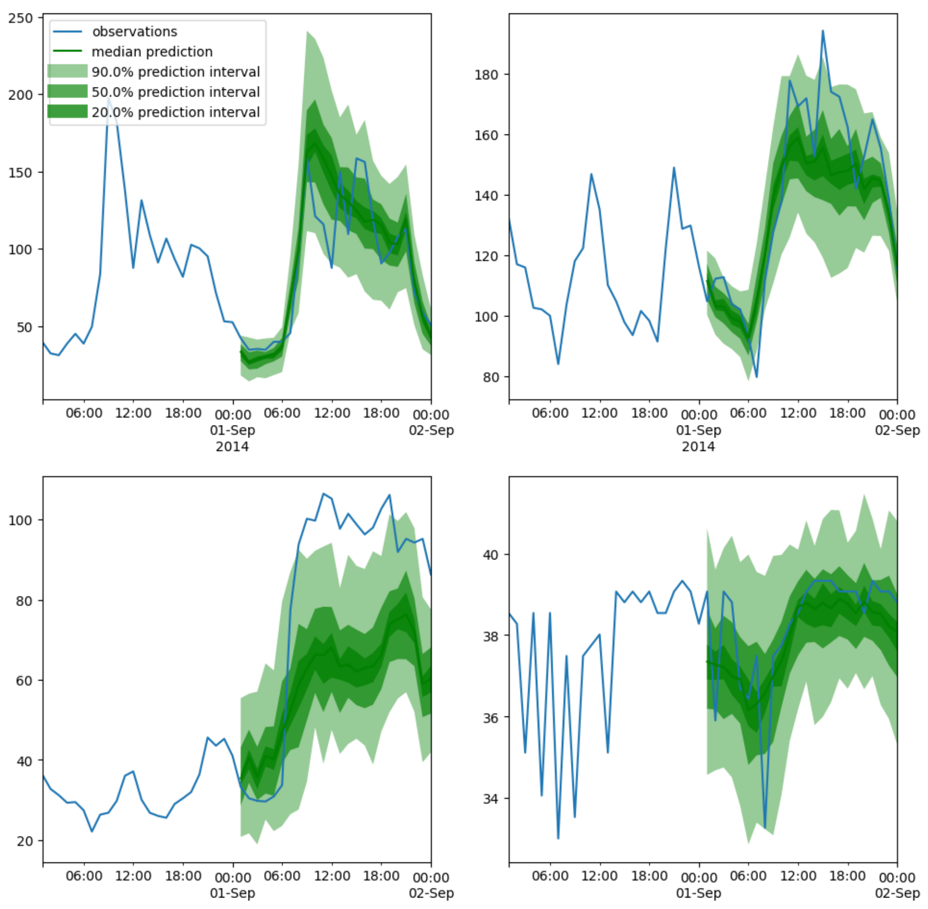


Figure 20: This figure shows an example of a **successful** multivariate forecast from experiment 2. For each trial in experiment 2 we make a forecast for 370 time series and here 4 of those are plotted.

6.5 Experiment 3 - Sequence length and LSH

In this experiment, we put LSH Attention to the test. We evaluate the GPU memory allocation of the Reformer MAF and the Transformer MAF for different conditioning range sequence lengths. Both reversibility and LSH Attention is turned on in the Reformer MAF for this section. Our goal is to produce observations that can help us deduce if the LSH Attention improves memory efficiency as the sequence length increases. The memory sampling is not done in the exact same place in the TMAF and the RMAF because the implementations are quite different, but we try to sample as close to the attention computation as possible.

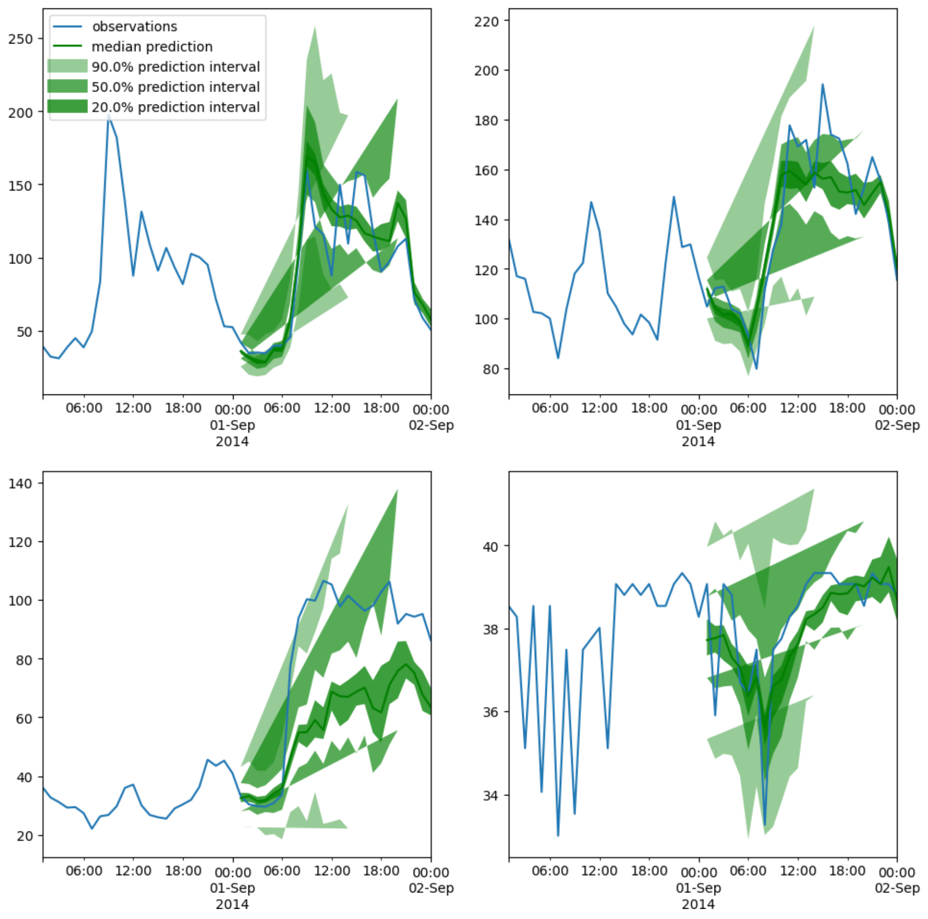


Figure 21: This figure shows an example of a **failed** multivariate forecast from experiment 2. For each trial in experiment 2 we make a forecast for 370 time series and here 4 of those are plotted.

Model	**CR size**	**Sample position**	**Space allocated (GB)**
RMAF	48	After QK computation	0.78
RMAF	48	After LSH attention	0.82
RMAF	96	After QK computation	0.96
RMAF	96	After LSH attention	1.03
RMAF	144	After QK computation	1.15
RMAF	144	After LSH attention	1.25
RMAF	192	After QK computation	1.39
RMAF	192	After LSH attention	1.52
TMAF	48	After self attention in Encoder	0.74
TMAF	48	After self attention in Decoder	0.79
TMAF	48	After Encoder/Decoder attention	0.80
TMAF	96	After self attention in Decoder	0.82
TMAF	96	After Encoder/Decoder attention	0.82
TMAF	96	After self attention in Encoder	0.74
TMAF	144	After self attention in Encoder	0.86
TMAF	144	After self attention in Decoder	0.90
TMAF	144	After Encoder/Decoder attention	0.90
TMAF	192	After self attention in Decoder	0.96
TMAF	192	After Encoder/Decoder attention	0.97
TMAF	192	After self attention in Encoder	0.87

Table 6: TMAF vs RMAF - Memory use over Electricity dataset. The sample positions are either inside the attention computation or right after the attention computation. We did not follow a precedent here. We choose to put the sample positions where we believed a memory effect could be observed. CR size above refers to conditioning range size.

Model	**CR size**	**Sample position**	**Space allocated (GB)**
RMAF	48	After QK computation	0.48
RMAF	48	After LSH attention	0.52
RMAF	96	After QK computation	0.61
RMAF	96	After LSH attention	0.68
RMAF	144	After QK computation	0.72
RMAF	144	After LSH attention	0.81
RMAF	192	After QK computation	0.91
RMAF	192	After LSH attention	1.03
TMAF	48	After self attention in Decoder	0.36
TMAF	48	After Encoder/Decoder attention	0.36
TMAF	48	After self attention in Encoder	0.34
TMAF	96	After self attention in Encoder	0.41
TMAF	96	After self attention in Decoder	0.40
TMAF	96	After Encoder/Decoder attention	0.40
TMAF	144	After self attention in Decoder	0.42
TMAF	144	After Encoder/Decoder attention	0.42
TMAF	144	After self attention in Encoder	0.40
TMAF	192	After self attention in Decoder	0.49
TMAF	192	After Encoder/Decoder attention	0.49
TMAF	192	After self attention in Encoder	0.49

Table 7: TMAF vs RMAF - Memory use over Solar dataset. The sample positions are either inside the attention computation or right after the attention computation. We did not follow a precedent here. We choose to put the sample positions where we believed a memory effect could be observed. CR size above refers to conditioning range size.

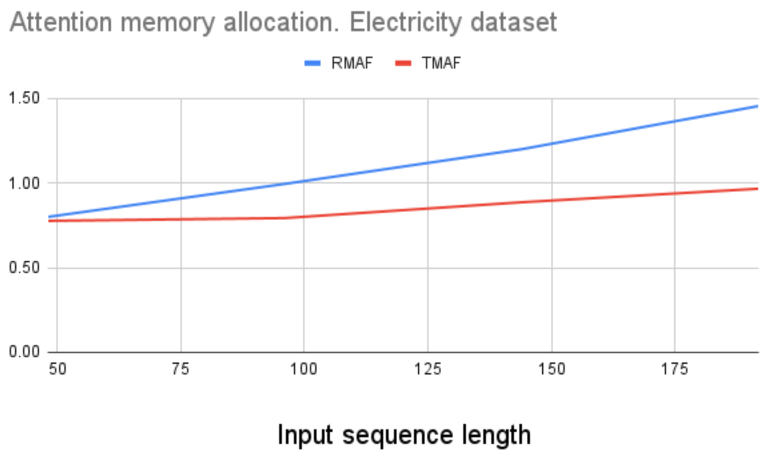


Figure 22: This line chart shows the average memory measured close to the attention computation for different sequence lengths for the RMAF and the TMAF. The line chart summarizes data which can be read in Table 6. The unit of the y-axis is gigabytes.

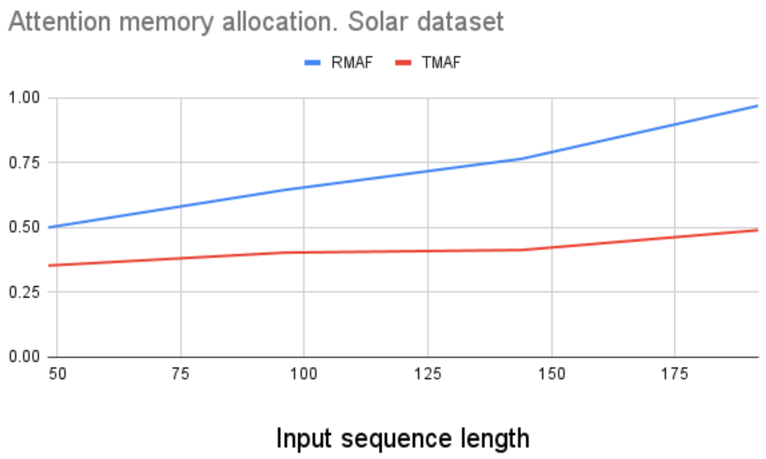


Figure 23: This line chart shows the average memory measured close to the attention computation for different sequence lengths for the RMAF and the TMAF. The line chart summarizes data which can be read in Table 7. The unit of the y-axis is gigabytes.

6.5.1 Results summary

We observed a higher memory cost for the RMAF than the TMAF as the sequence length was increased in this experiment, see Tables 6 and 7. The most petite sequence length, 48, resulted in the TMAF and the RMAF allocating almost the same amount of memory. The most extended sequence length, 192, resulted in the RMAF allocating around 70 percent more memory than the TMAF.

7 Discussion and Conclusions

The question that this thesis tries to answer is:

> Can the probabilistic forecasting model Transformer MAF be made more efficient through replacing the Transformer with a Reformer?

We did three experiments to produce observations that could help us answer the main research question by answering three more minor questions.

7.1 Experiment 1 - Reproduction and benchmarking

The first experiment focused on answering the question:

> Can the Reformer MAF produce forecasts of similar quality as the Transformer MAF?

We find that the Reformer MAF produces forecasts of the same quality as the Transformer MAF. Our conclusion is motivated by the similar CRPS-scores for the TMAF and the RMAF that resulted from the first experiment. Both models had a mean CRPS score of 0.394 for the Electricity dataset and 0.025 for the Solar dataset.

7.2 Experiment 2 - Layer count and reversibility

The second experiment was our attempt at producing observations that could answer the question:

> Does the reversible residual blocks of the Reformer MAF make it more memory efficient than the Transformer MAF?

The results of our second experiment show two things. First that the Reformer's reversible residual blocks introduce a considerable overhead cost in terms of memory use. Secondly, it appears as though the reversible residual blocks deliver on their promise not to increase memory use as we increase the number of layers. Although we must conclude that the Reformer MAF was not more memory efficient than the Transformer MAF in this experiment, the fact that memory allocation appears to not increase with more layers shows some promise for future work.

7.3 Experiment 3 - Sequence length and LSH

In the third experiment, we change our focus from the reversible blocks to the LSH Attention. We ask ourselves the question:

> Does LSH Attention of the Reformer MAF make it more memory efficient than the Transformer MAF?

After running the third experiment, we observed results that were similar to those of the second experiment. LSH attention works, but it introduces an overhead cost in memory use that makes the Reformer MAF perform worse in memory consumption than the Transformer MAF. We believe the LSH Attention does not improve memory efficiency because the input sequence lengths of our experiments (below 1000) are much smaller than the sequence lengths in the Reformer paper (above 1000).

7.4 Memory overhead cost

The RMAF induced a significant memory overhead for our second experiment. In Figures 19b and 19a, we can read that the RMAF allocated 7-8 times as much memory as the TMAF did. We will now raise three possible reasons why the Reformer allocated more memory than the TMAF in our experiments. As the MAF part of RMAF and TMAF is the same, we will focus on the differences between the Transformer and the Reformer.

7.4.1 Reversibility

The reversibility of the Reformer doubles the size of the input by design. This design is visualized in Figure 5 and there put in contrast to the method used in the Transformer. If the memory cost is doubled, why does the Reformer use reversibility? We explained the positive effects of reversibility in Section 3.2.2 but let us repeat it. Reversible neural networks do not need to save intermediate activation values during training, reducing the memory cost of model layers. Summarized, reversibility increases memory cost related to input size but reduces memory cost per model layer. We suspect that our experiments did not have enough layers in our models to make reversibility worth its price. Future work should be done that investigates when reversibility is worth its cost.

7.4.2 Encoder to Decoder

The Encoder stack output is passed to the Decoder stack in both the Transformer and the Reformer, but there are differences in how this communication is done. We suspect that the transmission in the Reformer is less memory efficient than the one in the Transformer. During our experiments, there appeared to be larger tensor sizes in the Reformer's decoder stack than in the Transformer's decoder stack. We believe this size difference results from how the encoder stack output is concatenated to the decoder input. Future research should explore if the whole connection step between the encoder and decoder stack could be optimized.

7.4.3 Attention Computation

The Reformer implementation is likely not as optimized as the Transformer regarding memory management and computation in general. In our second experiment, we turned off the LSH attention in the Reformer, which makes it use full attention as the Transformer does. But the Reformer's full attention is not implemented the same way as the Transformer's full attention. Both the Transformer and the Reformer used in this paper were implemented in PyTorch. Still, the implementation quality varies. There is one clear example in the code that we wish to highlight for future research. The Reformer uses the general-purpose PyTorch method *.einsum* to perform an essential step of the full attention computation. The Transformer uses what we believe to be a more optimized matrix multiplication method *.bmm*. Below we have a quote from the PyTorch documentation about *.einsum* which we believe is evidence for our case that einsum is a suboptimal way of performing attention computation.

> This function does not optimize the given expression, so a different formula for the same computation may run faster or consume less memory.[4]

The Reformer's LSH attention implementation also uses *.einsum*. Therefore it is likely that the LSH attention also could be improved by replacing *.einsum*.

7.5 Summary

Let us summarize the conclusions about the three experiments we did. The Reformer MAF can produce forecasts of similar quality as the Transformer MAF, but we find no evidence that the Reformer MAF is more memory efficient than the Transformer MAF. Indeed we showed that the overhead cost is too large to be more memory efficient for the considered cases. The results of our experiments are surprising to us because the theoretical complexity of the Reformer is better than that of the Transformer (See Tables 1, 2). We presented three reasons why we believe we observed a high memory overhead with the Reformer in Section 7.4.3.

7.6 Future Work

Although we did not find evidence that replacing the Transformer with a Reformer improved the memory efficiency for our forecasting experiments, we believe that future work could find such evidence. Let us describe four leads that could guide future work.

First, let us discuss the implementation of the Reformer. The Reformer currently does not have an official implementation in either PyTorch or Tensorflow. However, it seems likely that at least Tensorflow receives one in the future since the Reformer is the work of researchers employed by Google, and Google drives the development of Tensorflow. With an official Tensorflow or PyTorch implementation of the Reformer, we believe one could replicate our experiments and evaluate the performance again.

Second, altering the sequence length and chunking parameters could lead to experiments where the properties of the Reformer make it perform much better

[4]https://pytorch.org/docs/stable/generated/torch.einsum.html

and even better than the Transformer. A good starting place is to make the sequence length far longer than in our experiments and then design experiments exploring different chunking parameter values.

Third, one could explore other transformer models that claim to improve upon the memory efficiency of the Transformer. The Reformer is not the only transformer model that claims to improve upon the memory use of the Transformer. Examples of other models that make similar claims as the Reformer are the Performer [10], the Linear Attention Transformer [26], and the Compressive Transformer [36].

Fourth, the normalizing flow part of our model could be explored further. We did not have any specific experiments that examined the parameters of the Masked Autoregressive Flow and their impact on the results. Future work could explore different parameter settings or replace the MAF with an entirely different normalizing flow model.

7.7 Ethical concerns

While it is hard for us to see how the specific modifications we made in our experiments can introduce new ethical concerns or sustainability issues, it is not hard to think of problems with probabilistic forecasting in general.

The first ethical problem with probabilistic forecasting that comes to mind is the consequences these types of forecasts can have in the context of insurance rates. Let us explore a fictive example, a company called Safe Homes Forever AB sells burglary insurance. To make as much money as possible, they want to give their customers different rates based on how many burglaries there are in their home area. After training a probabilistic forecasting model, e.g., Transformer MAF or Reformer MAF, the company can predict burglary rates linked to specific regions for many years to come. The ethical problem then is that the people who need the insurance the most, people who live in poor neighborhoods with a lot of crime, will have the highest insurance rate. The consequence of using this type of model is that it can increase the inequalities in our society.

The second ethical problem that we see is linked to the government's use of probabilistic forecasting models. In times of crisis where the demand for political action is high, it is an easy way out for politicians to abdicate all responsibility and say that they act according to the projections made by a forecast model. The model then enables politicians to escape responsibility for their actions and actively encourages politicians to be passive.

7.8 Sustainability

Finally, there is a general problem linked to the training process of deep learning models in terms of how much energy is consumed [16]. We did not think about energy consumption during our research, but we believe we should do it going forward. It seems a good practice with machine learning to avoid using a more complex model than necessary because it can be costly in the hardware required.

Adding to that, it is also a good practice in terms of energy consumption because more layers entail more computations, resulting in more energy consumption.

References

[1] Alexander Alexandrov et al. "GluonTS: Probabilistic Time Series Models in Python". In: *CoRR* abs/1906.05264 (2019). arXiv: 1906.05264. URL: http://arxiv.org/abs/1906.05264.

[2] Alexandr Andoni et al. "Practical and Optimal LSH for Angular Distance". In: *Advances in Neural Information Processing Systems*. Ed. by C. Cortes et al. Vol. 28. Curran Associates, Inc., 2015. URL: https://proceedings.neurips.cc/paper/2015/file/2823f4797102ce1a1aec05359cc16dd9-Paper.pdf.

[3] The World Bank. *Global Photovoltaic Power Potential by Country*. 2020. URL: https://www.worldbank.org/en/topic/energy/publication/solar-photovoltaic-power-potential-by-country.

[4] Konstantinos Benidis et al. "Neural forecasting: Introduction and literature overview". In: *CoRR* abs/2004.10240 (2020). arXiv: 2004.10240. URL: https://arxiv.org/abs/2004.10240.

[5] Vladimir I Bogachev. *Measure theory*. Vol. 1. Springer Science & Business Media, 2007.

[6] George EP Box et al. *Time series analysis: forecasting and control*. John Wiley & Sons, 2015.

[7] Kanad Chakraborty et al. "Forecasting the behavior of multivariate time series using neural networks". In: *Neural networks* 5.6 (1992), pp. 961–970.

[8] Mark Chen et al. "Generative pretraining from pixels". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 1691–1703.

[9] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[10] Krzysztof Choromanski et al. "Rethinking Attention with Performers". In: *CoRR* abs/2009.14794 (2020). arXiv: 2009.14794. URL: https://arxiv.org/abs/2009.14794.

[11] Marcella Cornia et al. "Meshed-memory transformer for image captioning". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2020, pp. 10578–10587.

[12] Jacob Devlin et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: *CoRR* abs/1810.04805 (2018). arXiv: 1810.04805. URL: http://arxiv.org/abs/1810.04805.

[13] Laurent Dinh, Jascha Sohl-Dickstein, and Samy Bengio. "Density estimation using Real NVP". In: *CoRR* abs/1605.08803 (2016). arXiv: `1605.08803`. URL: `http://arxiv.org/abs/1605.08803`.

[14] Dheeru Dua and Casey Graff. *UCI Machine Learning Repository*. 2017. URL: `http://archive.ics.uci.edu/ml`.

[15] David Foster. *Generative deep learning: teaching machines to paint, write, compose, and play*. O'Reilly Media, 2019.

[16] Eva Garcia-Martin et al. "Estimation of energy consumption in machine learning". In: *Journal of Parallel and Distributed Computing* 134 (2019), pp. 75–88.

[17] Mathieu Germain et al. "Made: Masked autoencoder for distribution estimation". In: *International Conference on Machine Learning*. PMLR. 2015, pp. 881–889.

[18] Harshvardhan GM et al. "A comprehensive survey and analysis of generative models in machine learning". In: *Computer Science Review* 38 (2020), p. 100285. ISSN: 1574-0137. DOI: `https://doi.org/10.1016/j.cosrev.2020.100285`. URL: `https://www.sciencedirect.com/science/article/pii/S1574013720303853`.

[19] Yoav Goldberg and Omer Levy. "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method". In: *arXiv preprint arXiv:1402.3722* (2014).

[20] Aidan N. Gomez et al. "The Reversible Residual Network: Backpropagation Without Storing Activations". In: *CoRR* abs/1707.04585 (2017). arXiv: `1707.04585`. URL: `http://arxiv.org/abs/1707.04585`.

[21] Ian Goodfellow et al. "Generative adversarial networks". In: *Communications of the ACM* 63.11 (2020), pp. 139–144.

[22] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778. DOI: `10.1109/CVPR.2016.90`.

[23] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[24] Chin-Wei Huang et al. "Neural autoregressive flows". In: *International Conference on Machine Learning*. PMLR. 2018, pp. 2078–2087.

[25] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2018.

[26] Angelos Katharopoulos et al. "Transformers are rnns: Fast autoregressive transformers with linear attention". In: *International Conference on Machine Learning*. PMLR. 2020, pp. 5156–5165.

[27] Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. "Reformer: The Efficient Transformer". In: *CoRR* abs/2001.04451 (2020). arXiv: `2001.04451`. URL: `https://arxiv.org/abs/2001.04451`.

[28] Guokun Lai et al. "Modeling long-and short-term temporal patterns with deep neural networks". In: *The 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 2018, pp. 95–104.

[29] James E Matheson and Robert L Winkler. "Scoring rules for continuous probability distributions". In: *Management science* 22.10 (1976), pp. 1087–1096.

[30] John Milnor and David W Weaver. *Topology from the differentiable viewpoint*. Princeton university press, 1997.

[31] Junier B. Oliva et al. *Transformation Autoregressive Networks*. 2018. arXiv: `1801.09819 [stat.ML]`.

[32] George Papamakarios, Theo Pavlakou, and Iain Murray. *Masked Autoregressive Flow for Density Estimation*. 2018. arXiv: `1705.07057 [stat.ML]`.

[33] George Papamakarios et al. *Normalizing Flows for Probabilistic Modeling and Inference*. 2021. arXiv: `1912.02762 [stat.ML]`.

[34] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[35] Alec Radford et al. "Language models are unsupervised multitask learners". In: *OpenAI blog* 1.8 (2019), p. 9.

[36] Jack W. Rae et al. "Compressive Transformers for Long-Range Sequence Modelling". In: *CoRR* abs/1911.05507 (2019). arXiv: `1911.05507`. URL: `http://arxiv.org/abs/1911.05507`.

[37] Kashif Rasul et al. "Multi-variate Probabilistic Time Series Forecasting via Conditioned Normalizing Flows". In: *CoRR* abs/2002.06103 (2020). arXiv: `2002.06103`. URL: `https://arxiv.org/abs/2002.06103`.

[38] Walter Rudin. *Real and complex analysis*. 2006.

[39] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[40] Masahiro Ryo et al. "Basic Principles of Temporal Dynamics". In: *Trends in Ecology & Evolution* 34.8 (2019), pp. 723–733. ISSN: 0169-5347. DOI: `https://doi.org/10.1016/j.tree.2019.03.007`. URL: `https://www.sciencedirect.com/science/article/pii/S0169534719300874`.

[41] David Salinas et al. "DeepAR: Probabilistic forecasting with autoregressive recurrent networks". In: *International Journal of Forecasting* 36.3 (2020), pp. 1181–1191. ISSN: 0169-2070. DOI: `https://doi.org/10.1016/j.ijforecast.2019.07.001`. URL: `https://www.sciencedirect.com/science/article/pii/S0169207019301888`.

[42] David Salinas et al. "High-Dimensional Multivariate Forecasting with Low-Rank Gaussian Copula Processes". In: *CoRR* abs/1910.03002 (2019). arXiv: `1910.03002`. URL: `http://arxiv.org/abs/1910.03002`.

[43] Malcolm Slaney and Michael Casey. "Locality-sensitive hashing for finding nearest neighbors [lecture notes]". In: *IEEE Signal processing magazine* 25.2 (2008), pp. 128–131.

[44] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. "Sequence to sequence learning with neural networks". In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

[45] Ruey S Tsay. *Multivariate time series analysis: with R and financial applications*. John Wiley & Sons, 2013.

[46] Ashish Vaswani et al. "Attention Is All You Need". In: *CoRR* abs/1706.03762 (2017). arXiv: `1706.03762`. URL: `http://arxiv.org/abs/1706.03762`.

[47] Guoqiang Zhang, B. Eddy Patuwo, and Michael Y. Hu. "Forecasting with artificial neural networks:: The state of the art". In: *International Journal of Forecasting* 14.1 (1998), pp. 35–62. ISSN: 0169-2070. DOI: `https://doi.org/10.1016/S0169-2070(97)00044-7`. URL: `https://www.sciencedirect.com/science/article/pii/S0169207097000447`.

TRITA -EECS-EX-2022:34