Unifying Transformers and Convolutional Neural Processes

Lakee Sivaraya

A thesis presented for the degree of
Master of Engineering

Department of Engineering
University of Cambridge
Date of Submission: May 26, 2024

Acknowledgement

I would like to thank my supervisor Prof. Richard E. Turner for his guidance and support throughout this project. I also would like to thank Matt Ashman for his considerable assistance with the code and theory upon which this project is built on. His insights have been invaluable in shaping my understanding of the material. Furthermore, I would extend my sincere appreciation to Cristiana Diaconu, Startis Markou, James Requeima and Junhyuck Kim for their incredibly helpful discussions and feedback during our supervision meetings. Everyone's advice and insights have been invaluable in shaping this project and my understanding of the material.

Thank you all for your support and contributions.

Abstract

Neural Processes (NPs) are a class of general purpose models designed for uncertainty-aware meta learning. Knowledge of the uncertainty in predictions is crucial for real-world applications such as climate modelling, time series forecasting and decision-making in healthcare. In these applications, the ability to quantify uncertainty is as important as the prediction itself.

We highlight the limitations of the standard DeepSet based NP model, particularly its inability to model complicated relationships in the data in high dimensions. To extend the capabilities of NPs, we replace the DeepSet with two powerful architectures: Convolutional Neural Networks (CNNs) and Transformers giving rise to Convolutional Neural Processes (ConvNPs) and Transformer Neural Processes (TNPs) respectively. Both models are bound to have their own strengths and weaknesses.

Our experiments demonstrate that the TNP outperforms the ConvNP, especially in higher dimensions. It was observed that far outside the training data, the TNP behaves unpredictably, conversely the ConvNP remained stable. We hypothesize the stability of the ConvNP is due to the *translation equivariance* property of CNNs. Translation equivariance is the property of a model to produce the same output when the input is translated, such property is greatly beneficial when modelling stationary or periodic data. As the TNP does not have this property it generalizes poorly to data far from the training data.

To address the limitations of the TNP, we introduce translation equivariance to the TNP, by modifying the self-attention mechanism in the TNP. We call this model the *Translation Equivariant Transformer Neural Process* (TETNP). Our experiments show the TETNP outperforming the TNP and ConvNP on datasets that exhibit strong discontinuities and periodicity. On smooth datasets, the TETNP performs comparably to the TNP which is expected since these datasets do not require translation equivariance.

The quadratic computational complexity of full self attention is a key bottleneck in the TNP and TETNP models, limiting their scalability to large datasets. This thesis explores several methods to reduce the computational complexity of the TNP by the use of pseduotokens, and approximations to the self-attention mechanism via kernelization and MLPs. While our experiments demonstrates a substantial reduction in computational complexity, the approximations come at the cost of performance. All linear models struggled to model the discontinuous datasets, highlighting the loss of expressiveness in the model.

Overall this thesis provides a comprehensive study of the TNP and ConvNP models. We establish the importance of translation equivariance in the TNP and demonstrate its benefits. We conclude that the TETNP model is the most promising model, with strong

performance across a range of datasets especially in higher dimensions. ConvNPs scale terribly with dimensionality, making the TNP and TETNP a clear choice for large scale applications.

Contents

Chapter 1

Introduction

1.1 Motivation

Machine learning models have been immensely successful in variety of applications to generate predictions in data-driven domains such as computer vision, robotics, weather forecasting. While the success of these models is undeniable, they tend to lack the ability to understand the uncertainty in the predictions. This is a major drawback in the deployment of these models in real-world applications, for example, in weather forecasting, the uncertainty of the prediction of the weather is arguably as valuable as the prediction itself. In this work, we aim to implement a model that is **uncertainty aware** whilst also possessing further desirable properties.

1.2 Desirable Properties

On top of being uncertainty aware, we would like to insert some desirable inductive biases that help the model to generalize better and be more interpretable. These properties are:

Flexible: *The model should be able to work on a variety of data types.* As long as a data point can be represented as a vector, the model should be able to operate on it. This allows the model to be used in a variety of applications and domains.

Scalable: *The model should be able to learn large datasets and scale to as many inputs.* Which is not the case with many traditional models such as Large Language Models (LLMs) which are usually limited to a max number of tokens. Another aspect of scalability is the ability learn high-dimensional data with good computational efficiency.

Permutation Invariant: *The prediction of the model should not change if the order of the input data is changed.* When each data point contains the information about input and output pairs, the model should not care about the order in which they are fed into the model. For example, in the case of a weather forecasting model, which uses data from multiple weather stations, the model should not care about the order in which the data from the weather stations is fed into the model, thus making the model permutation invariant.

Translation Equivariant: *Shifting the input data by a constant amount should result in a constant shift in the predictions.*

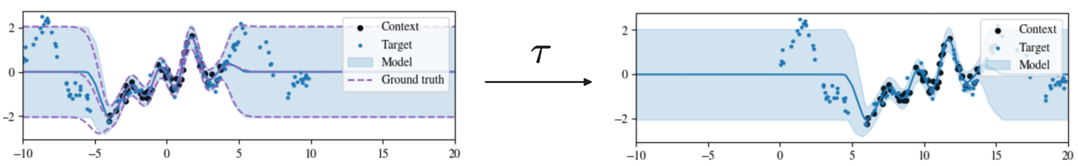


Figure 1.2.1: The Translation Equivariant property on a 1D dataset.

Figure 1.2.1 illustrates this property, when the input data on the left plot is shifted by a constant amount, the prediction should also shift by the same amount (right). Mathematical, a model f is translation equivariant if it satisfies the following property:

$$f : \boldsymbol{x} \to (\boldsymbol{x}, \hat{\boldsymbol{y}}) \tag{1.2.1}$$

$$f : \boldsymbol{x} + \boldsymbol{\tau} \to (\boldsymbol{x} + \boldsymbol{\tau}, \hat{\boldsymbol{y}}) \tag{1.2.2}$$

where $\boldsymbol{x}$ is the input and $\hat{\boldsymbol{y}}$ is the output and $\boldsymbol{\tau}$ is a constant shift in the input. Such property allows the model to be more robust and generalize better to unseen data, particularly in the case of stationary data.

Off the Grid Generalization: *The model should be capable of operating on off-the-grid data points.* Off the grid data points are the data points that are not in a regular gridded structure, such as images that have missing pixel values. Traditional models like Convolutional Neural Networks (CNNs) are not able to operate on off-the-grid data points since they require a regular structure to apply the convolution operation. By making the model off-the-grid generalizable, we can create models that can work on many types of datasets and easily handle missing data points. Furthermore, aiding in the performance of the model outside the context data. Applications such as image inpainting can particularly benefit from off-the-grid generalization.

Neural Processes (NPs) [Garnelo, Schwarz, et al. 2018] are a class of models that satisfy the above properties. The framework undermining NPs is general purpose, and thus can be modified with a variety of neural network architectures.

1.3 Aims and Objectives

In this work, we aim to implement and compare two different neural network architectures for Neural Processes, the first being based on a Convolutional Neural Network (CNN) called Convolutional Neural Processes (ConvNP) and the second being based on a Transformer architecture called Transformer Neural Processes (TNP). Our objective is to compare the performance of two models on a variety of datasets, focusing on their generalization capabilities and scalability.

We introduce extra inductive biases into the TNP to enhance its ability to generalize. Furthermore, we explore new Transformer architectures that have better computational efficiency compared to the original Transformer architecture.

Our objective is to develop a comprehensive understanding of the properties of these models. We aim to identify the best practices for using these models in different scenarios, highlighting contexts where they perform well and where they do not. This investigation will provide us valuable insights into the capabilities of these models and how they can be used in real-world applications.

Chapter 2

Neural Processes

2.1 Introduction

Neural Process (NP) is a meta-learning framework introduced in [Garnelo, Rosenbaum, et al. 2018; Garnelo, Schwarz, et al. 2018] that is used for few-shot uncertainty aware meta learning. There exists two variants of the Neural Process, the Conditional Neural Process (CNP) and the Latent Neural Process (LNP), we will focus entirely on the CNP for this project and hence we will implicitly refer to CNP as NP.

Neural Processes learn a distribution over the input locations *conditioned on the training data*. In the NP literature we refer the training data as the *context set* and the input locations we want to predict the output for as the *target set*. The model is trained on a meta datasets of context-target pairs by maximizing the likelihood of the target set *conditioned* the context set.

2.2 Architecture

2.2.1 Conditional Neural Processes

The general framework for a CNP requires us to take a context set $\mathcal{C} = \{\mathcal{C}_i\}_{i=1}^{N_c}$ containing input-output pair points $\mathcal{C}_i = (\boldsymbol{x}_i^{(c)}, \boldsymbol{y}_i^{(c)})$ and a target set $\mathcal{T} = \{\boldsymbol{x}_i^{(t)}\}_{i=1}^{N_t}$ containing inputs $\boldsymbol{x}_i^{(t)}$ we want to predict the outputs for.

The data points in the context set $\mathcal{C}_i$ are encoded into an embedding using network as follows:

$$\boldsymbol{r}(\mathcal{C}_i) = \mathrm{Enc}_\theta(\mathcal{C}_i) = \mathrm{Enc}_\theta([\boldsymbol{x}_i^{(c)}, \boldsymbol{y}_i^{(c)}]) \tag{2.2.1}$$

where $\boldsymbol{r}$ is the embedding of the context point $\mathcal{C}_i$ and θ are the parameters of the encoder. The embeddings of the context sets under processing to obtain a global representation of the dataset $\mathcal{C}$ as follows:

$$\boldsymbol{R}(\mathcal{C}) = \mathrm{Process}(\{\boldsymbol{r}(\mathcal{C}_i)\}_{i=1}^{D}) \tag{2.2.2}$$

The 'processing' must be **permutation invariant**, so typically it is a simple summation of the embeddings. The global representation $\boldsymbol{R}$ is then used to condition the decoder to predict the outputs of the target set to giving us a posterior distribution over the outputs $\boldsymbol{y_i^{(t)}}$.

$$p(\boldsymbol{y}_i^{(t)}|\boldsymbol{x}_i^{(t)}, \mathcal{C}) = \mathrm{Dec}_\theta(\boldsymbol{x}_i^{(t)}, \boldsymbol{R}(\mathcal{C})) \tag{2.2.3}$$

The overall architecture for the CNP is shown in Figure 2.2.1.

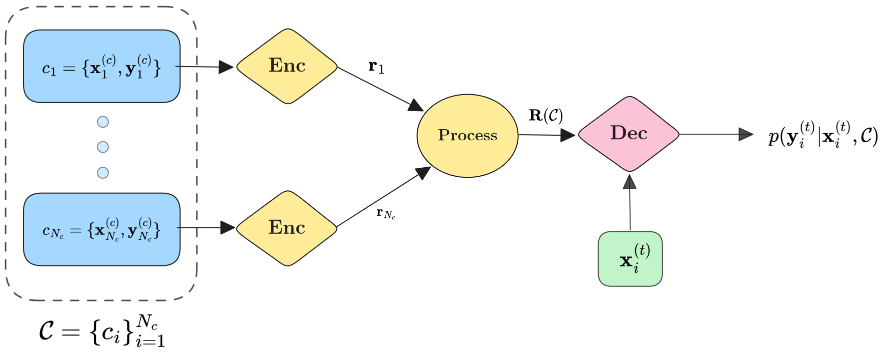


Figure 2.2.1: CNP Architecture: The model vectorizes each individual data point $\mathcal{C}_i$ in the context set $\mathcal{C}$ and then processes them to obtain a global representation $\boldsymbol{R}(\mathcal{C})$ which is then used to condition the decoder to predict a distribution over the target points $\boldsymbol{y}^{(t)}$.

In the original CNP paper, the encoder and decoder are implemented as simple Multi-Layer Perceptrons (MLPs) and the processing is implemented as a mean operation, which is an implementation of the 'DeepSet' architecture [Zaheer et al. 2018].

Importantly, CNPs make the strong assumption that the posterior distribution *factorizes* over the target points:

$$p(\boldsymbol{Y}^{(t)}|\boldsymbol{X}^{(t)}, \mathcal{C}) \overset{(a)}{=} \prod_{i=1}^{N_t} p(\boldsymbol{y_i^{(t)}}|\boldsymbol{x_i^{(t)}}, \boldsymbol{R}(\mathcal{C})) \tag{2.2.4}$$

$$\overset{(b)}{=} \prod_{i=1}^{N_t} \mathcal{N}(\boldsymbol{y_i^{(t)}}|\boldsymbol{\mu_i}, \boldsymbol{\sigma_i^2}) \tag{2.2.5}$$

The factorization assumption (a) allows the model can scale linearly with the number of target points with a tractable likelihood. However, this assumption means **CNPs are unable to generate coherent sample paths, they are only able to produce distributions over the target points.** Furthermore, we need to select a marginal likelihood for the distribution (b) which is usually a Heteroscedastic Gaussian Likelihood (Gaussian with a variance that varies with the input) [Garnelo, Rosenbaum, et al. 2018]. This adds an assumption since likelihood chosen may not be appropriate for the data we are modeling.

The model can be trained using simple maximum likelihood estimation (MLE) by minimizing the negative log-likelihood, giving us the following loss function:

$$\mathcal{L} = \mathbb{E}_{(\mathcal{C},\mathcal{T})}\left[-\sum_{i=1}^{|\mathcal{T}|}\log p(\boldsymbol{y}_{\boldsymbol{i}}^{\boldsymbol{(t)}}|\boldsymbol{x}_{\boldsymbol{i}}^{\boldsymbol{(t)}},\mathcal{C})\right] \tag{2.2.6}$$

2.3 Performance of Vanilla NP

Whilst the CNP using DeepSets is flexible and scalable, in reality it is unable to perform well on more complicated and higher dimensional data as the model is unable to learn a good representation of the data using a simple MLP and summation operation.

Could we replace the encoder and decoder with more powerful networks? And if so, what would be the best architecture to use?

We aim to answer this by exploring the use of a Convolutional Neural Network (CNN) and a Transformer as encoders of our NP. CNNs and Transformers have been shown to perform well on a variety of tasks and at scale, thus we hypothesize that they will be able to learn a better representation of the context set and improve the performance of the NP. Both are bound to have their unique advantages and disadvantages which we will explore in the following chapters.

Chapter 3

Convolutional Neural Processes

3.1 Introduction

Convolutional Neural Networks (CNNs) have been the state of the art in many image processing tasks [He et al. 2015; Simonyan and Zisserman 2015; Krizhevsky, Sutskever, and Geoffrey E Hinton 2012], due to their ability to learn spatial patterns in the data via convolutional filters. Since convolutions are translation equivariant, CNNs learns to recognize patterns regardless of their position in the image, making them desirable for a wide range of tasks, such as image classification, object detection, and segmentation. The performance of CNNs has made them to be a desirable backbone for a Neural Process, motivating the development of the Convolutional Neural Processes (ConvNPs) [Gordon et al. 2020]. We will briefly discuss the ConvNP model and its architecture however for a more comprehensive explanation, we refer the reader to the original paper.

3.2 Architecture

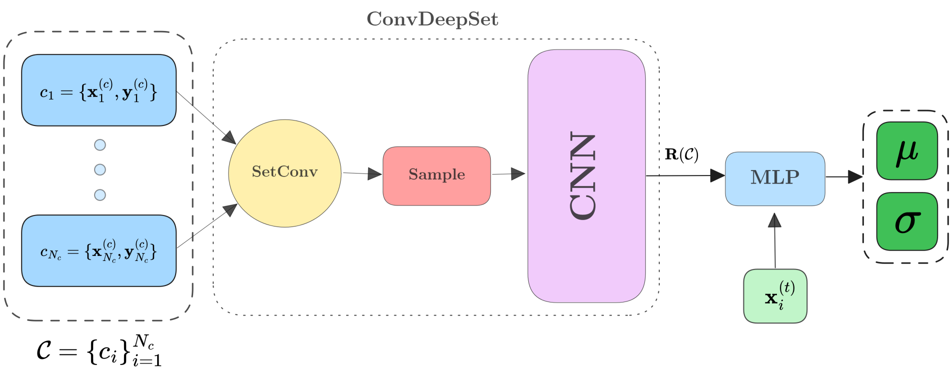


Figure 3.2.1: ConvNP Architecture. All the data points are transformed into a continuous functional embedding using the SetConv, then discretely sampled to be fed into a CNN. The CNN processes the data whose output is converted back to a continuous function using the SetConv operation.

There are a few hurdles are preventing us from directly plugging data into a CNN when working with off-grid data.

CNNs operate on **on-grid data**, implying the data is on a regular grid, such as an image. However, the data we are working with is sometimes **off-grid data**, e.g. time series dataset with irregular timestamps. Furthermore, to 'bake-in' the translation equivariance property, we need to be able to shift the data in the input space and the output space consistently. This is not trivial when using standard vector representations of the data as their is no inherent notion translation in a vector. Gordon et al. 2020 propose a solution to this problem by using **functional embeddings** to model the discrete data as *continuous functions*. With functional embeddings, the data can be translated in the input space and the output space consistently thus permitting translation equivariance.

The **SetConv** operation takes a set of input-output pairs and outputs a continuous functional representation of the data as follows:

$$\text{SetConv}(\{\boldsymbol{x_i}, \boldsymbol{y_i}\}_{i=1}^{N})(x) = \sum_{i=1}^{N} [1, \boldsymbol{y_i}]^T \mathcal{K}(\boldsymbol{x} - \boldsymbol{x_i}) \tag{3.2.1}$$

Where $\mathcal{K}$ is a Kernel function that measures the distance between the query x and the data point $\boldsymbol{x_i}$.

This operation has some key properties:

- We append 1 to output $\boldsymbol{y_i}$ when computing the SetConv, this acts as a flag allowing the model to knows which data points are observed and which are not. Say we have a data point of $\boldsymbol{y_i} = 0$, then if we did not append the 1, the model would not be able to distinguish between an observed data point and an unobserved data point (as both would be 0).
- The 'weight' of the Kernel depends only on the *relative distance* between points on the input space hence the model is translation equivariant.
- The summation over the data points naturally introduces **Permutation Invariance** to the model.

The function representation of the context set, is sampled at **evenly** spaced points giving us discrete data in a **on-grid** format. It is then fed into a CNN. After the CNN processes the data, it need to convert it back to a continuous function by using the SetConv operation. The final output of the encoder is a continuous function that represents the context set which can be queried at any point in the input space using the target set, given by:

$$R(\boldsymbol{x_t}) = \text{SetConv}(\text{CNN}(\{\text{SetConv}(\mathcal{C})(\boldsymbol{x_d})\}_{d=1}^{D}))(\boldsymbol{x_t}) \tag{3.2.2}$$

The decoder is a simple MLP that takes the output of the encoder and the target set as input and generates the mean and variance of the predictive distribution $\boldsymbol{\mu}(\boldsymbol{x_t}), \boldsymbol{\sigma}(\boldsymbol{x_t}) = \text{MLP}(R(\boldsymbol{x_t}))$

Chapter 4

Transformer Neural Processes

4.1 Transformers

4.1.1 Introduction

The Transformer [Vaswani et al. 2017] is a deep learning model that has been revolutionary in the field of machine learning. Originally devised as a sequence-to-sequence model that uses attention to learn the relationships between the input and output sequences. Transformers have been the state of the start in the field of natural language processing (NLP) leading to the development of models like BERT [Devlin et al. 2019] and GPT [Brown et al. 2020]. As of recent, they are gaining traction in fields like computer vision [Dosovitskiy et al. 2021] becoming the new state of the art in a field that has been dominated by Convolutional Neural Networks (CNNs). The attractiveness of transformers stems from their general purpose nature and strong ability to learn relationships with a large context window whilst being massively parallelizable. A brief overview of the transformer model is given below.

4.1.2 Embedding

Data points it into a vector representation called an *embedding* or *token* ensuring positional information is added to these tokens via a positional encoding. The embedding is a simple linear transformation of the input data $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ where N is the number of data points and D is the dimensionality of the data.

4.1.3 Self-Attention

Self-attention is the key component of the transformer model. It is a way to learn the relationships between the data in the input sequence itself. Consider a language modelling task, the sentence `The quick brown fox jumps over the lazy dog` has strong attention between the data like `fox` and `jumps` representing an action, `brown` and `fox` representing the color of the fox and so on, then there are very weak attentions between data `quick` and `dog` representing the lack of relationship between the two data. Using self-attention we can learn these relationships between the data in the input sequence, giving a powerful mechanism to learn the relationships.

In the transformer models we will use the embeddings $\boldsymbol{X} \in \mathbb{R}^{N\times D}$ as the input to generate a query $\boldsymbol{Q} \in \mathbb{R}^{N\times d_k}$, a key $\boldsymbol{K} \in \mathbb{R}^{N\times d_k}$ and a value $\boldsymbol{V} \in \mathbb{R}^{N\times d_v}$ matrices via a simple linear transformation matrix $\boldsymbol{W_q} \in \mathbb{R}^{D\times d_k}$, $\boldsymbol{W_k} \in \mathbb{R}^{D\times d_k}$ and $\boldsymbol{W_v} \in \mathbb{R}^{D\times d_v}$ respectively.

Where each row of the matrices is the query, key and value vectors for each data point in the input sequence.

The query, key and value matrices are then used to compute the attention matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$ as follows:

$$\boldsymbol{A} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right) \tag{4.1.1}$$

The intuition is to compute the similarity between the query and the key vectors using dot product between the query and key vectors. The softmax is used to normalize the attention matrix so that the rows sum to 1. The softmax is also scaled by $\sqrt{d_k}$ to prevent the softmax from saturating. The attention matrix is then used to compute the output matrix $\boldsymbol{H} \in \mathbb{R}^{N\times d_v}$ as $\boldsymbol{H} = \boldsymbol{A}\boldsymbol{V}$

The overall attention function for a layer is given by:

$$\boldsymbol{H} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{4.1.2}$$

4.1.4 Multi-Head Self-Attention

Currently, the attention matrix $\boldsymbol{A}$ is computed once so the model only learns one attention relationship, however, we can take advantage of using multiple attention ‘heads’ in parallel to learn many attention relationships, this scheme is called the *Multi-Head Attention* (MHSA).

Each attention head is computed using simple dot product attention of a transformed query, key and value matrix. They are transformed by a simple linear layer (a matrix) which is unique for each head of the MHSA, $\boldsymbol{W_q}^{(i)} \in \mathbb{R}^{d_k\times d_k}$, $\boldsymbol{W_k}^{(i)} \in \mathbb{R}^{d_k\times d_k}$ and $\boldsymbol{W_v}^{(i)} \in \mathbb{R}^{d_v\times d_v}$ where $i \in [1, h]$ for a head count of h. Then the attention for the particular head is computed as follows:

$$\boldsymbol{H}^{(i)} = \text{Attention}(\boldsymbol{Q}\boldsymbol{W_q}^{(i)}, \boldsymbol{K}\boldsymbol{W_k}^{(i)}, \boldsymbol{V}\boldsymbol{W_v}^{(i)}) \in \mathbb{R}^{N\times d_v} \tag{4.1.3}$$

The output of the MHSA is the concatenation of the outputs of each head $\boldsymbol{H}^{(i)}$ (stacked on to of each other) multiplied by a learnable matrix $\boldsymbol{W_O} \in \mathbb{R}^{hd_v\times D}$ which transforms the concatenated output to the original dimensionality of the input sequence.

$$\text{MHSA}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{cat}(\boldsymbol{H}^{(1)}; \boldsymbol{H}^{(2)}; \ldots; \boldsymbol{H}^{(h)})\boldsymbol{W_O} = \begin{bmatrix} \boldsymbol{H}^{(1)} \\ \boldsymbol{H}^{(2)} \\ \vdots \\ \boldsymbol{H}^{(h)} \end{bmatrix} \boldsymbol{W_O} \in \mathbb{R}^{N\times D}$$

4.1.5 Encoder

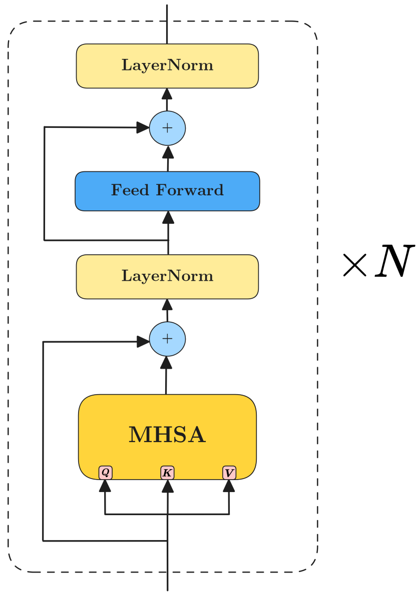


Figure 4.1.1: Transformer Encoder [Vaswani et al. 2017]

Figure 4.1.1 shows the encoder of the transformer model. The encoder is composed of a stack of N identical layers. Each layer is composed of two sub-layers, the MHSA and a simple feed-forward network. The output of each sub-layer is passed through a residual connection and a layer normalization operation [Ba, Kiros, and Geoffrey E. Hinton 2016]. The final output of the encoder is the output of the last layer which shall be denoted as $\boldsymbol{Y} \in \mathbb{R}^{N \times D}$.

Key Points

The Transformer Encoder Layer takes an input set of embeddings $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ and outputs a set of embeddings $\boldsymbol{Y} \in \mathbb{R}^{N \times D}$ of the **same dimensionality** but with the **patterns of the input sequence learned**. It can be viewed as a function that takes a set and outputs a set of the same dimensionality.

4.2 Transformer Neural Process

An Attention based encoder for the Neural Process was investigated by Kim et al. 2019, although the results were impressive, the model fails to perform at larger scale and 'tends to make overconfident predictions and have poor performance on sequential decision-making problems' [Nguyen and Grover 2023]. Consequently, it is natural to progression to consider Transformer [Vaswani et al. 2017] as a potential candidate that can the performance of the Neural Process. Nguyen and Grover 2023 introduced the Transformer Neural Process (TNP) which uses an *encoder-only* Transformer to learn the relationships between the context and target points via self-attention with appropriate masking.

4.2.1 Model Architecture

Similar to the standard Neural Process architecture, we are required to encode data points within the context set into a vector representation, in language modelling literature we refer to this as tokenization. The tokenization is achieved using a simple Multi-Layer Perceptron (MLP) to encode the data points into tokens with a configurable token dimension, D_{em}.

The TNP tokenizes the target points with padded with zeros to represent the absence of values for the target data points. Subsequently, both context and target tokens are fed into the transformer at the same time, in contrast to the standard NP approach of computing a context representation and then inferencing the target points.

A flag bit is introduced into the tokens to indicate whether the token is a context or target token. For the context dataset $\mathcal{C} = \{(\boldsymbol{x_i}, \boldsymbol{y_i})\}_{i=1}^{N_c}$ and target set $\mathcal{T} = \{(\boldsymbol{x_i})\}_{i=N_c+1}^{N}$, the model will encode a data point into a token $\boldsymbol{t_i}$ as follows:

$$\boldsymbol{t_i} = \begin{cases} \mathrm{MLP}(\texttt{cat}[\boldsymbol{x_i}, 0, \boldsymbol{y_i}]) & \text{if } i \leq N_c \\ \mathrm{MLP}(\texttt{cat}[\boldsymbol{x_i}, 1, \mathbf{0}]) & \text{if } i > N_c \end{cases}$$

A flag bit of 0 represents indicate a context token and 1 represents a target token, `cat` is the concatenation operation and $\mathbf{0}$ is a vector of zeros of the same dimension as $\boldsymbol{y_i}$.

The tokens are passed into a standard Transformer encoder to learn the relationships between the context and target points. Importantly the Transformer encoder is masked such that the target tokens can only attend to the context tokens and previous target tokens. Alternatively one can perform self attention on the context tokens then cross attention between the context and target tokens giving a more efficient implementation [Feng et al. 2022] (we use this implementation in our experiments).

The output of the Transformer encoder is passed through a Multi-Layer Perceptron (MLP) to generate the mean and variance of the predictive distribution of the target points.

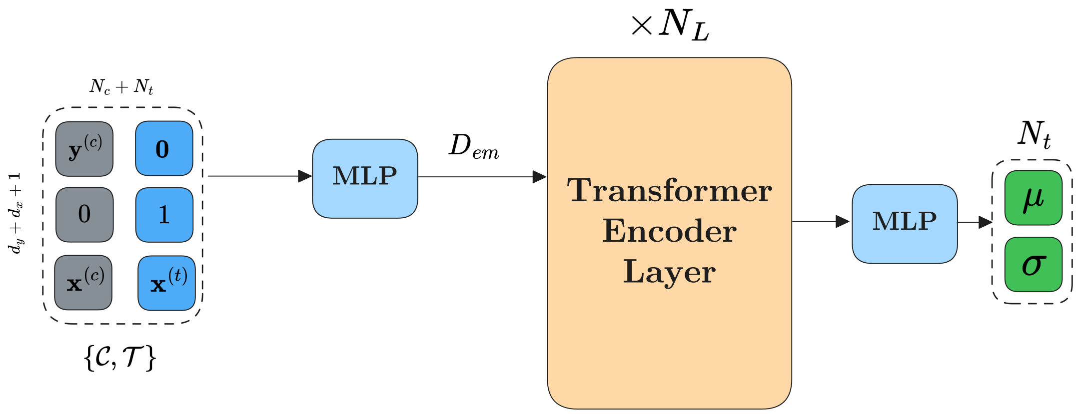


Figure 4.2.1: Vanilla Transformer Neural Process Architecture. All the context and target sets are tokenized and passed through the Transformer Layer. The output of the Transformer Layer is passed through an MLP to generate the mean and variance of the predictive distribution.

4.2.2 Performance

One of the key benefits of Transformers (and attention) is their ability to learn a global view of the data, analogous to an 'infinite receptive field' in Convolutional Neural Networks. This feature enables the model to learn complicated relationships across many length scales in the data. However, this can be a double-edged sword as the model can overfit to the data within its training region and fail to generalize to unseen data.

Translation Equivariance is a key property behind CNNs which allows them to generalize excellently to unseen data by learning features irrespective of their position in the data. Such feature learning is critical to modelling real world data, where the absolute positions of the data are often less significant the *relative positions* between the data points. For instance, in image classification the position of the object in the image does not matter as much as the features that characterize object itself. Transformers lack this property, resulting in unpredictable behaviors when the data is shifted out of context region, as shown in Figure 4.2.2.

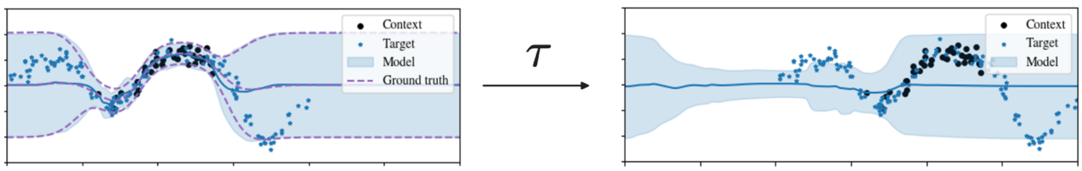


Figure 4.2.2: Vanilla Transformer Neural Process failing to generalize in out of context data. The TNP successfully models the data in the context region (left) but fails to make predictions when the data is shifted (right).

What if we could combine the best of both worlds? Could we create a Translation Equivariant Transformer Neural Process (TETNP) which possesses the global view of the Transformer and the generalization properties of the CNN? This is the question we will investigate in the next section and is the main contribution of this work and Ashman et al. 2024.

4.3 Translation Equivariant TNP

Translation Equivariance requires the model to yield the same output when the input is shifted. Such property must imply that the model only learns the relative distances between the data points ($\boldsymbol{x_i} - \boldsymbol{x_j}$) and *not* the absolute positions of the data points. How could one enforce such a property in the Transformer Neural Process? The solution used is very simple, we add a term to the attention mechanisms in the Transformer to enforce the Translation Equivariance property!

Consider the standard attention mechanism in the Transformer, the attention weights are computed as:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\boldsymbol{E}\right)\boldsymbol{V} \tag{4.3.1}$$

$$\boldsymbol{E}_{ij} = \frac{\boldsymbol{q_i} \cdot \boldsymbol{k_j}}{\sqrt{d_k}} \tag{4.3.2}$$

To create a Translation Equivariant Attention mechanism we add a term to the attention weights which enforces the Translation Equivariance property. The new attention weights are computed with the following equation:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{X}) = \text{softmax}\left(\boldsymbol{E} + F(\boldsymbol{\Delta})\right)\boldsymbol{V} \tag{4.3.3}$$

$$\boldsymbol{\Delta_{ij}} = \boldsymbol{x_i} - \boldsymbol{x_j} \tag{4.3.4}$$

where F is some function that introduces non-linearity into the attention weights, which is applied to each entry of the matrix independently $(F(\boldsymbol{\Delta})_{ij} = F(\boldsymbol{\Delta_{ij}}))$, we will investigate the effect of different functions in the experiments later on. If all the input locations are shifted by a constant $\boldsymbol{\tau}$, the relative term will remain the same $\boldsymbol{F(\Delta_{ij})} = F(\boldsymbol{x_i} + \boldsymbol{\tau} - \boldsymbol{x_j} - \boldsymbol{\tau}) = F(\boldsymbol{x_i} - \boldsymbol{x_j})$ and the attention weights will remain the same.

Importantly, as we are enforcing the Transformer to learn the relative distances between the data points, we must remove the $\boldsymbol{x}$ values from the tokenization of the data points. Therefore, the tokenization of the data points is given by:

$$\boldsymbol{t_i} = \begin{cases} \text{MLP}(\texttt{cat}[0, \boldsymbol{y_i}]) & \text{if } i \leq N_c \\ \text{MLP}(\texttt{cat}[1, \boldsymbol{0}]) & \text{if } i > N_c \end{cases}$$

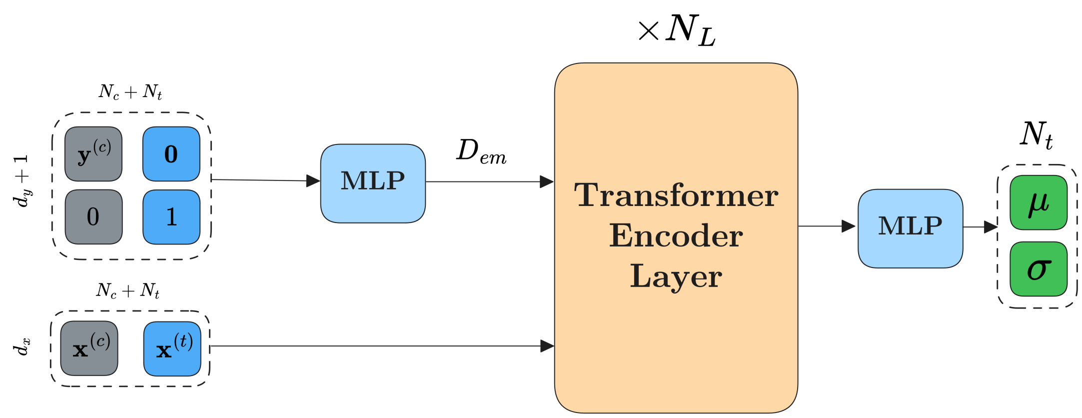


Figure 4.3.1: Translation Equivariant Transformer Neural Process Architecture. All the outputs of the context and target sets are tokenized and passed through the Transformer Layer, whilst the input locations are passed in raw to the attention mechanism to enforce the Translation Equivariance property. The output of the Transformer Layer is passed through an MLP to generate the mean and variance of the predictive distribution.

The removal of the $\boldsymbol{x}$ values from the tokenization allows the tokens to solely encode the $\boldsymbol{y}$ values, thereby reducing the dimensionality of the tokens. These separations the encoding for $\boldsymbol{x}$ and $\boldsymbol{y}$ values could be beneficial for the model, as in the vanilla TNP the model loses distinction between the $\boldsymbol{x}$ and $\boldsymbol{y}$ due to their joint tokenization. We hypothesize that separation of input and output tokens could aid in the learning of the relationships between input and output data points and improve the generalization of the model.

Chapter 5

Experimentation on 1D Datasets

In this section we will conduct a thorough investigation of the models on 1D datasets. Initially, we will optimize the hyperparameters of the TNP by assessing their performance on the data using a constant 1 million parameter budget. Following this, we will compare the performance of the TNP and ConvNP on smooth dataset generated from a Gaussian Process (GP) [Rasmussen and Williams 2006] and a more complex dataset generated from a sawtooth function. The validation loss will be used as a metric to measure the in-context performance, and the model fits will be used as a qualitative metric to measure the generalization of the models. Finally, we will investigate the computational complexity of the models on 1D data to evaluate their scaling properties with respect to the number of context and target points.

5.1 Datasets

We will use two datasets to evaluate the models: a smooth Gaussian process and a discontinuous and periodic sawtooth function. From both datasets, choose to sample $N_c \sim \mathcal{U}(32, 64)$ context points and $N_t = 128$ target points. The context region is defined as $x \in [-4, 4]$ and the target region is defined as $x \in [-10, 10]$.

5.1.1 Gaussian Process

Our objective is for our model to learn the underlying function of the Gaussian Process using the given data. The Gaussian Process is defined as follows:

$$f(x) \sim \mathcal{GP}(0, k(x, x')) \tag{5.1.1}$$

Where we use the squared exponential kernel:

$$k(x, x') = \exp\left(-\frac{(x - x')^2}{2l^2}\right) \tag{5.1.2}$$

This kernel represents a smooth function with length scale l. The log of the length scale is sampled from a uniform distribution $\log_{10}(l) \sim \mathcal{U}(-0.301, 0.301)$. Gaussian noise with standard deviation $\sigma = 0.2$ is added to our generated data.

5.1.2 Sawtooth

Whilst the GP is useful for testing the models on a smooth function, we also want to test the models on a more complex function, particularly one with discontinuities. The sawtooth function is a great candidate for this. The sawtooth function with frequency f is defined as:

$$f(x) = x - \frac{1}{f}\lfloor xf \rfloor + n \tag{5.1.3}$$

Where n is Gaussian noise with standard deviation $\sigma = 0.025$ and f is sampled from a uniform distribution $f \sim \mathcal{U}(0.5, 2)$.

5.2 Relative Attention Function

As discussed in section 4.3 the matrix of differences ($\boldsymbol{\Delta}$) is passed through a function F to apply non-linear transformations and acts as hyperparameter of our model. As a baseline, we will use a simple linear function with no bias and gradient 1 ('identity') which is linear. For non-linear transformer, we evaluate using a Gaussian Radial Basis Function (RBF) and a Multi-Layer Perceptron (MLP). The functions are defined below:

$$F_{\text{identity}}(\Delta) = \Delta \qquad F_{\text{RBF}}(\Delta) = \exp\left(-\frac{\Delta^2}{2\sigma^2}\right) \qquad F_{\text{MLP}}(\Delta) = \text{MLP}(\Delta) \tag{5.2.1}$$

Where σ is a hyperparameter of the RBF function and MLP(Δ) is a 2-layer MLP with ReLU [Agarap 2019] activation functions.

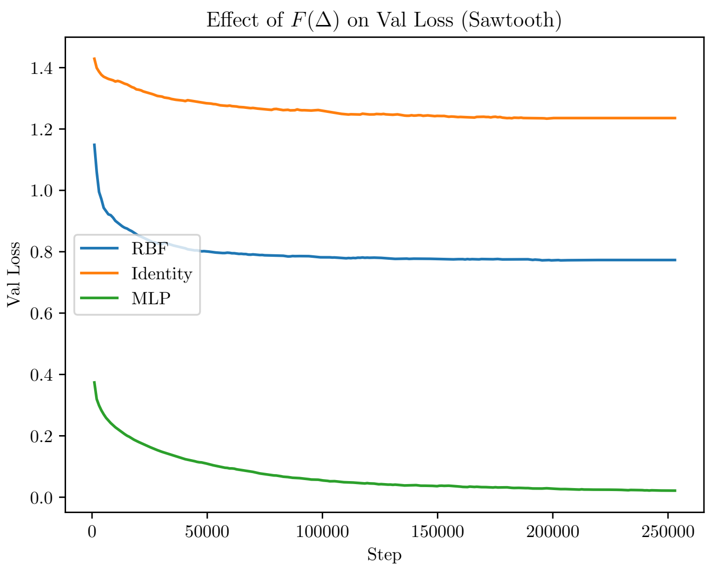
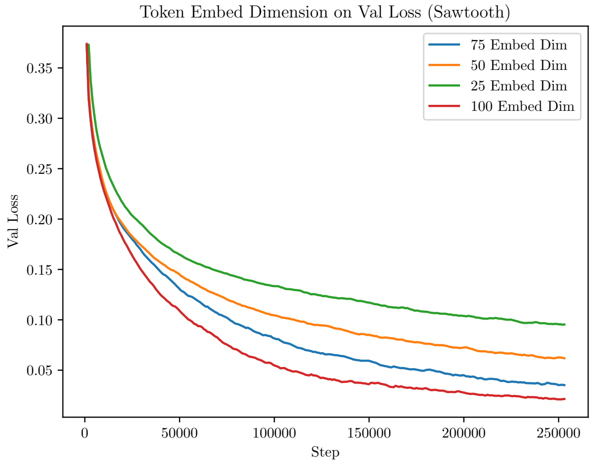


Figure 5.2.1: Relative Attention Functions on Validation Loss for the TETNP on the 1D Sawtooth Dataset. Lower validation loss is better.

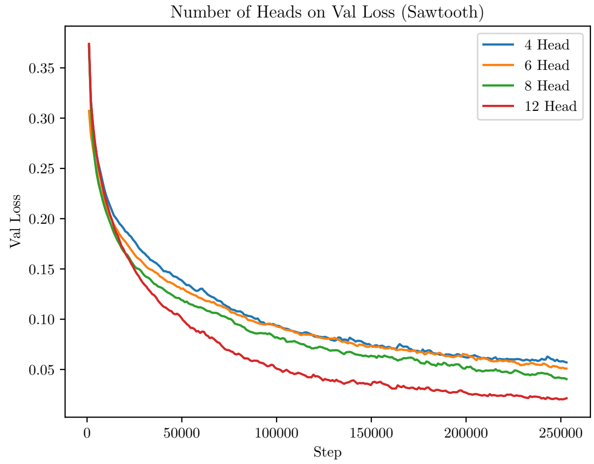
Figure 5.2.1 presents the validation loss curves for the TNP with different relative attention functions. The results shows that the MLP function outperforms the other two functions by a large margin. This can be attributed to the MLPs ability to **learn** representations whilst the other two functions have a fixed closed form. We note that the RBF function performs better than the identity function since the effect of adding the raw difference corrupts the dot product attention. We can conclude that the MLP function is the best function out of three we considered. The computational cost of using the MLP function is not significantly higher than the other two functions, since the MLP is very small.

5.3 Optimizing Hyperparameters

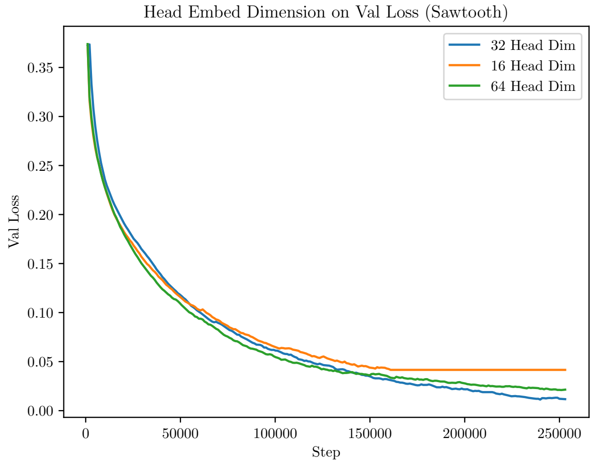
The multi-headed attention mechanism in the Transformer Encoder has three hyperparameters that we will investigate. These are the token embedding dimension of the data (D_{em}), the number of attention heads (N_h) and the embedding dimension of the attention heads (D_h). We will investigate how changing these hyperparameters affects the performance of the model. To gauge the effect of these hyperparameters, we use a 1 million parameter model and reduce the value of one of the hyperparameters and see the effect on the validation loss. This process is repeated for the other hyperparameters to see which hyperparameters have the most effect on the validation loss.



(a) Token Embedding Dimension



(b) Number of Heads



(c) Head Embedding Dimension

Figure 5.3.1: Hyperparameter Selection. The three plots consider the effect of reducing the size of a single hyperparameter on the validation loss. Lower validation loss is better.

From Figure 5.3.1 it is clear that reducing the token embedding dimension has a significant effect on the validation loss, with the number of heads making a smaller difference and the head embedding dimension making very little difference. This highlights the importance of the token embedding dimension when using low dimensional data as the input data (in this case sawtooth function). Furthermore, the head embedding dimension has very little effect on the validation loss, so we can set this to a small value to reduce the number of parameters in the model and distribute the parameters to the other hyperparameters. Using this knowledge, we will now investigate how to select the hyperparameters using a *constant parameter budget* of 1 million parameters.

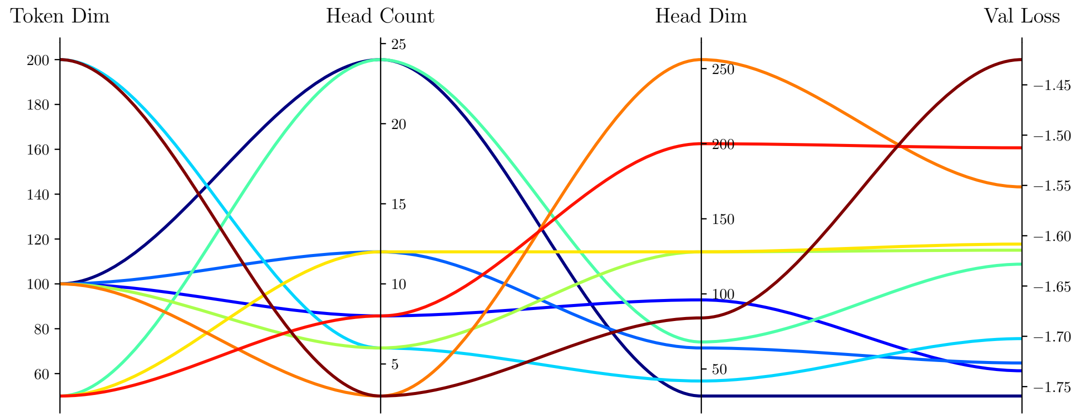


Figure 5.3.2: Constant Parameter Budget Hyperparameter Selection. The parallel coordinates plot shows the validation loss for different hyperparameter configurations. Dark blue is the best and dark red is the worst.

Figure 5.3.2 shows a parallel coordinates plot of the validation loss for different hyperparameter configurations, where dark blue is the best and dark red is the worst. The model that performs the best has a very high token embedding dimension, high headcount and low head embedding dimension, which is consistent with the previous results. We can also see that if we go too high in the token embedding dimension (200), the model performs worse as we have to sacrifice the number of heads in the transformer. These results give us a set of best practices for selecting hyperparameters for the TNP: *high token embedding dimension, high number of heads and low head embedding dimension.*

5.4 TNP vs ConvNP

Using the 'best' TNP model, we will compare the performance of the ConvNP and TNP on fitting sawtooth and GP (using EQ Kernel) data using 1 million parameters.

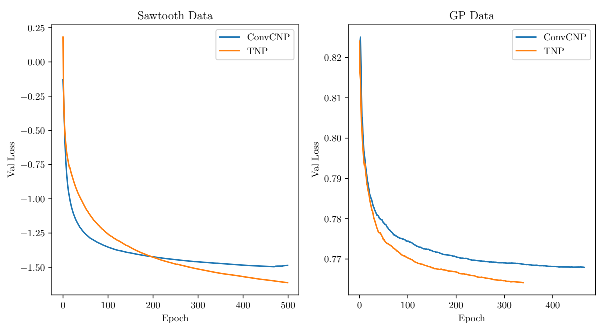
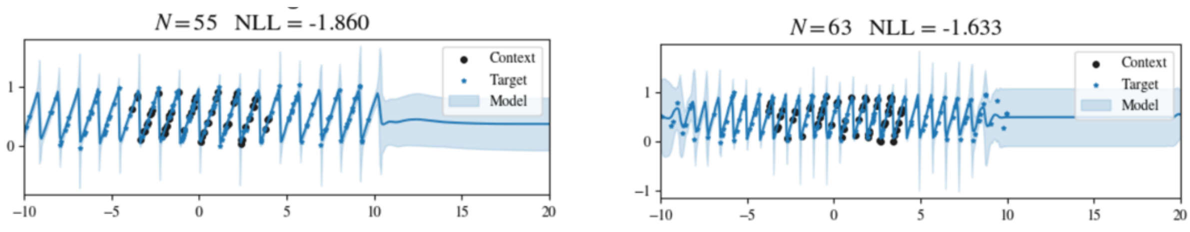


Figure 5.4.1: ConvNP vs TNP on Sawtooth and GP Data

Figure 5.4.1 shows the validation loss curves for the ConvNP and TNP on sawtooth and GP data. The validation loss for the TNP is lower than the ConvNP for both datasets which is very promising and indicates the TNP is a better model than the ConvNP.



(a) TNP (b) ConvNP

Figure 5.4.2: ConvNP vs TNP on Sawtooth Data. The context set inputs are between [-4, 4] and the target set inputs are between [-10, 10] which extends beyond the context set to test the models' extrapolation capabilities.

Inspecting the model fits on sawtooth data Figure 5.4.2, we observe that the TNP can extrapolate the structure of the sawtooth function beyond the range of the context set (black points) whilst the ConvNP performs decently but fails to retain the structure as well as the TNP, since the amplitude of the sawtooth reduces the further away from the context set. Hence, we can conclude that the TNP can better understand the structure of the data than the ConvNP.

5.4.1 Computational Complexity

N_c	N_t	ConvNP Memory (MB)	TNP Memory (MB)	TETNP (MB)
10	10	18	13	
100	10	16	18	
1000	10	16	339	
5000	10	124	9357	
10	1000	36	399	
100	1000	36	469	
1000	1000	36	1510	
5000	1000	124	13480	

Table 5.4.1: Memory Usage in inference of the ConvNP and TETNP on 1D data using N_c context points and N_t target points.

Table 5.4.1 shows that the TETNP uses drastically more memory than the ConvNP for the same number of context and target points. This is due to the quadratic complexity of the TETNP which scales with $\mathcal{O}(N_c^2 + N_c N_t)$ compared to the linear complexity of the ConvNP which scales with $\mathcal{O}(N_c D_x^3 + N_t D_x)$. Whilst on the 1D dataset the ConvNP is more memory efficient, the TETNPs complexity does not scale with the number of dimensions of the data, whilst the ConvNP *scales cubically with the number of dimensions*. This may suggest that the TETNP is more suitable for high-dimensional data than the ConvNP since the TETNP has a fixed memory cost per dimension. We will investigate this in the next chapter on 2D datasets to see if the discrepancy in memory usage is still present.

Chapter 6

Experimentation on 2D Datasets

6.1 Datasets

6.1.1 Gaussian Process

The 2D Gaussian Process is the natural extension of the 1D Gaussian Process described in subsection 5.1.1 where we use the squared exponential kernel. We continue to use the same range of lengthscale across both input dimensions as the 1D Gaussian Process.

6.1.2 Sawtooth

The 2D Sawtooth dataset is the natural extension of the 1D Sawtooth dataset described in subsection 5.1.2. We continue to use the same period T and noise n across both input dimensions as the 1D Sawtooth dataset.

6.1.3 Restricted Sawtooth

The restricted sawtooth limits the 'direction of travel' of the sawtooth function to the line of $x_1 = x_2$ or $x_1 = -x_2$. Under this dataset the model only learn a subset of the 'full sawtooth' function. We can use this probe how well the models can generalize to samples from the full sawtooth function.

6.2 Post or Pre MLP

In our original formulation of the TETNP (section 4.3) we pass the matrix of differences ($\boldsymbol{\Delta}$) between x values through an MLP to apply non-linearities then add it to the dot product attention Equation 4.3.3, whilst this performs well we can also consider applying this non-linearity after combining the dot product attention and the relative attention, this method is called the 'Post MLP'. The formulation of the 'Post MLP' is as follows:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{X}) = \text{softmax}\left(\boldsymbol{E}\right)\boldsymbol{V} \tag{6.2.1}$$

$$\text{Pre MLP:} \quad \boldsymbol{E_{ij}} = \boldsymbol{D_{ij}} + \text{MLP}(\boldsymbol{\Delta_{ij}}) \tag{6.2.2}$$
$$\text{Post MLP:} \quad \boldsymbol{E_{ij}} = \text{MLP}(\texttt{cat}[\boldsymbol{D_{ij}}, \boldsymbol{\Delta_{ij}}]) \tag{6.2.3}$$

Where

$$\boldsymbol{D_{ij}} = \boldsymbol{q_i} \cdot \boldsymbol{k_j} / \sqrt{d_k} \qquad \boldsymbol{\Delta_{ij}} = \boldsymbol{x_i} - \boldsymbol{x_j} \tag{6.2.4}$$

We will investigate the performance of the TETNP with the 'Post MLP' compared to the original 'Pre MLP'. The Sawtooth dataset is chosen for this experiment as it is more difficult to learn than the Gaussian Process dataset, thereby allowing us to observe the differences between the two functions more clearly.

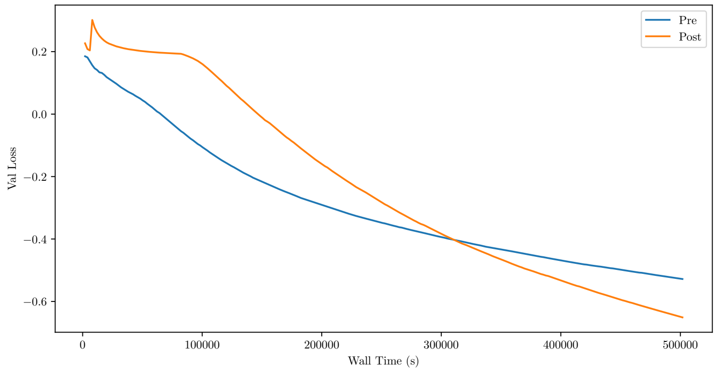


Figure 6.2.1: Validation Loss of TETNP with the 'Post MLP' and 'Pre MLP' on the 2D Sawtooth Dataset. Lower validation loss is better.

The results show that the TETNP with the 'Post MLP' outperforms the TETNP with the 'Pre MLP' by quite a large margin. Trivially the Post function can further refine the dot product attention through the MLP whilst in the 'Pre' function the MLP is *only* applied to the $\boldsymbol{\Delta}$ matrix. The computational complexity of these two functions are not too different as the MLP are small and applied to matrices of the same size.

6.3 Model Comparison

We have discovered in the 1D section that the TETNP outperforms the vanilla TNP in all cases, hence for the 2D experiments we will only compare the ConvNP to the TETNP. When performing our experiments we will use models which are both 1 million parameters in size, to ensure a fair comparison.

6.3.1 Gaussian Process

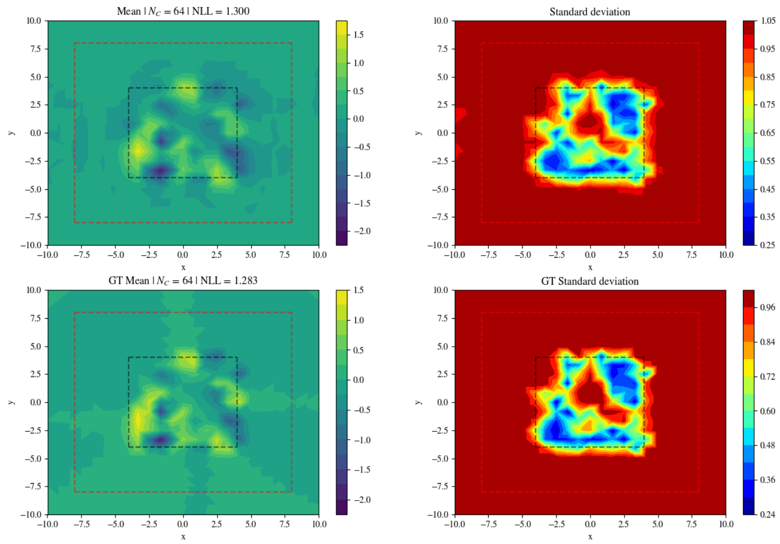
As mentioned previously, the Gaussian Process dataset is not very difficult to learn and the ConvNP and TETNP both perform very well on this dataset with the TETNP outperforming the ConvNP by a small margin as shown in Table 6.3.1.

Observing the samples from the ConvNP and TETNP for the 2D Gaussian Process dataset Figure 6.3.1, we note that both models successfully learn the underlying process, with the TETNP being able to perform slightly better than the ConvNP. Given the

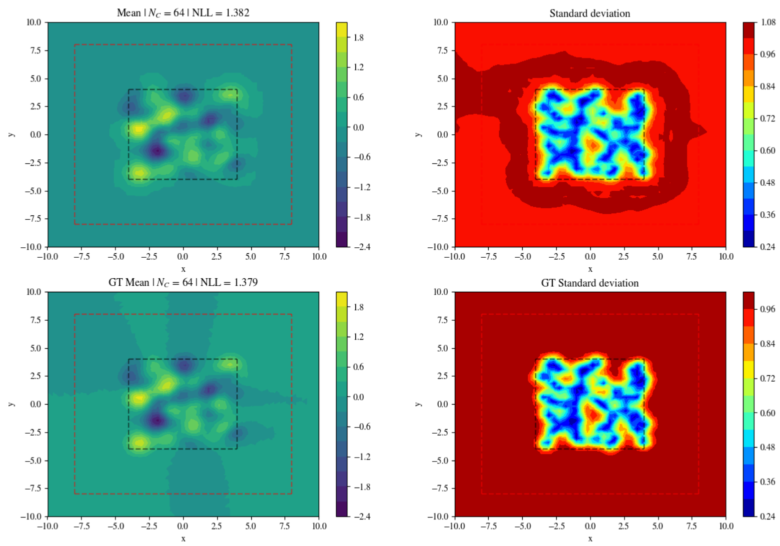
Model	Validation Loss
ConvNP	1.168
TETNP	1.134

Table 6.3.1: Validation Loss of ConvNP and TETNP on the 2D Gaussian Process dataset after training for 4 hours using 1 million parameters models and 64 context points. Lower is better.

smooth nature of the GP dataset, it is not surprising that both models perform well. To highlight the differences between the ConvNP and TETNP, we will proceed to the Sawtooth dataset which is more difficult to learn.



(a) ConvNP (top plot is the model prediction and bottom is the ground truth GP)

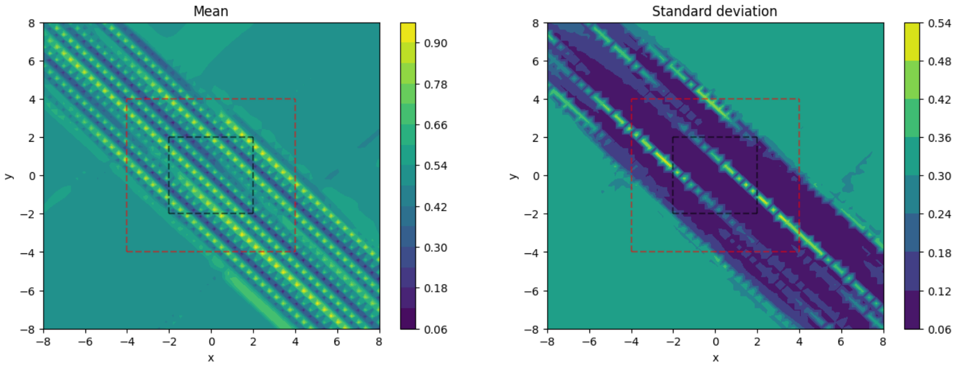


(b) TETNP (top plot is the model prediction and bottom is the ground truth GP)

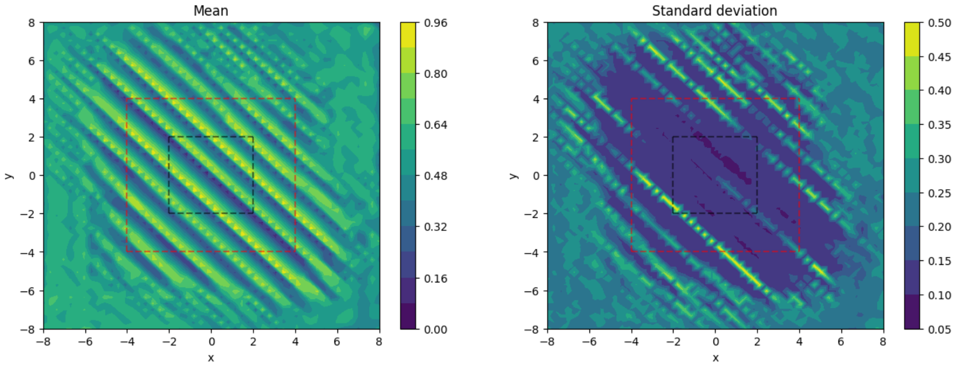
Figure 6.3.1: Samples from ConvNP and TETNP on a frequency 2D Gaussian Process using 64 context points.

6.3.2 Restricted Sawtooth and Rotational Equivariance

As previously stated the restricted sawtooth dataset is a subset of the full sawtooth dataset which restricts the 'direction of travel' of the sawtooth function to the line of $x_1 = x_2$ or $x_1 = -x_2$. Training the models on this dataset will highlight the generalization ability of the models.



(a) TETNP



(b) ConvNP

Figure 6.3.2: Samples from TETNP and ConvNP on the Restricted Sawtooth dataset. The region inside the black dotted box is the region the models were trained on and the region outside the black dotted box is the region the models were not trained on. 64 context points were used.

Figure 6.3.2 shows that the ConvNP performs excellently on this dataset, which is expected as CNNs use filters to learn features and patterns in the data explicitly, allowing them to perform excellently on extrapolation tasks (outside the black dotted box region). The TETNP on the other hand struggles to generalize fully within the target region (red dotted box) and outside the target region. Instead, it learns to extrapolate the sawtooth along one axis, but not the other. This brings to light a limitation of the Transformer architecture - it has very little interpretability which can result in unexpected behavior.

Is this TETNP able to generalize to the full sawtooth dataset? To answer this question we will simply run the TETNP on a rotated version of the restricted sawtooth dataset which is the full sawtooth dataset.

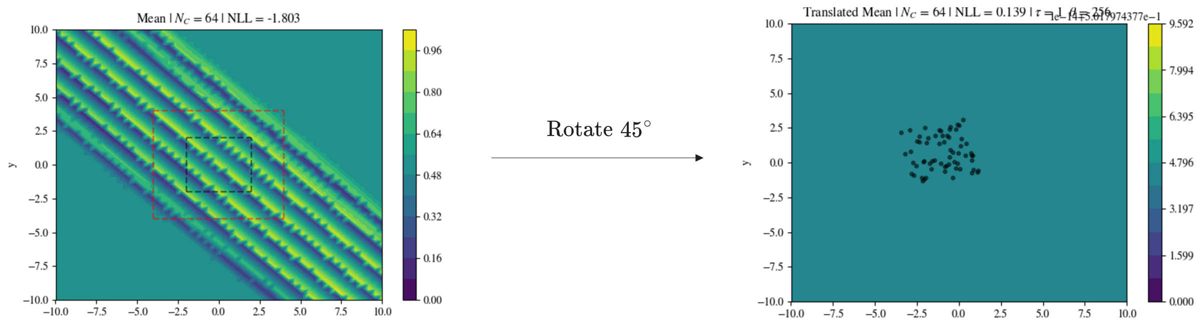


Figure 6.3.3: Generalization of the TETNP. The context points on the left are rotated by 45 degrees and the inference is performed on this rotated set giving us the right plot.

In Figure 6.3.3 we explicitly reduced the target and context region size to allow for the TETNP to cover the full target region. The bottom plot illustrates the predictions when we rotate the context points by 45 degrees, clearly the TETNP completely fails, predicting only a constant mean and standard deviation. It struggles to generalize to rotations, which brings us to the question: *Could introducing rotational equivariance to the TETNP help it generalize to the full sawtooth dataset?*

Rotational Equivariance

Introducing rotational equivariance (RE) is relatively straightforward. In our formulation of the Translation Equivariance Attention, we use a pairwise-difference matrix $\boldsymbol{\Delta}$ which contains the differences between the $\boldsymbol{x}$ values of all the data points.

$$\text{Not RE}: \quad \boldsymbol{\Delta}_{ij} = \boldsymbol{x}_i - \boldsymbol{x}_j \tag{6.3.1}$$
$$\text{RE}: \quad \boldsymbol{\Delta}_{ij} = \|\boldsymbol{x}_i - \boldsymbol{x}_j\|_2 \tag{6.3.2}$$

To introduce rotational equivariance we can simply take the L2 norm of the $\boldsymbol{\Delta}$ matrix which will give us the distance between all the data points. Distances are invariant to rotation, hence the TETNP should be rotationally equivariant. Using this new $\boldsymbol{\Delta}$ matrix we can train the TETNP on the restricted sawtooth dataset and observe if it can generalize to the full sawtooth dataset.

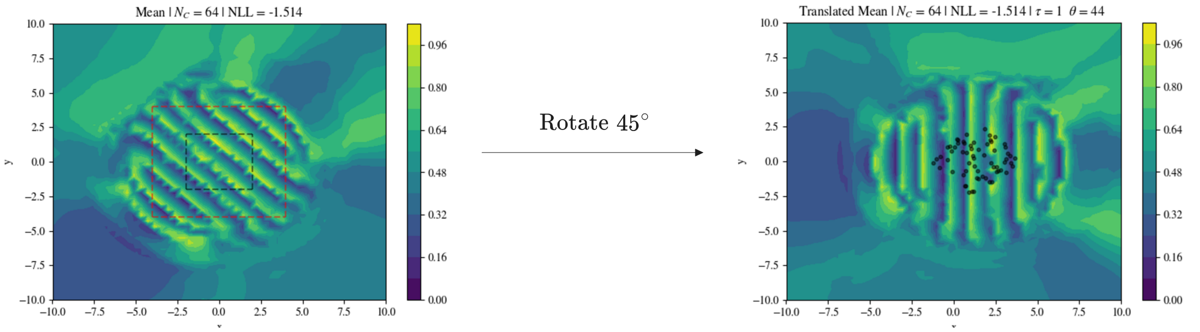


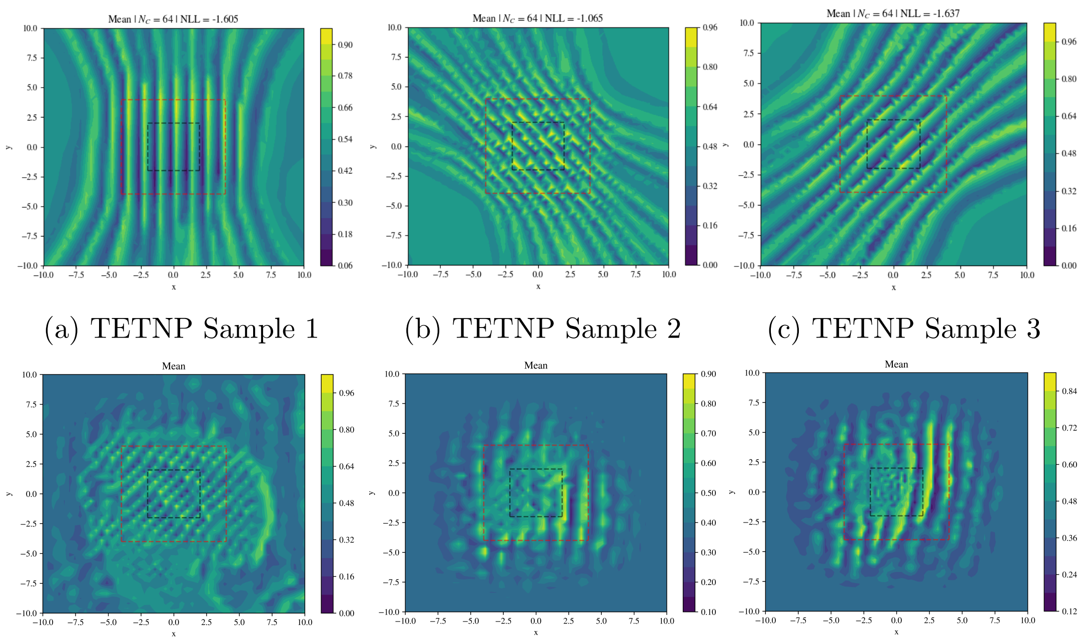
Figure 6.3.4: Generalization of the RE-TETNP. The context points on the left are rotated by 45 degrees and the inference is performed on this rotated set giving us the right plot.

Figure 6.3.4 demonstrates a massive improvement in the TETNP's ability to generalize to the full sawtooth dataset. It has learned to extrapolate the sawtooth in all directions, which is a significant improvement over the non-RE TETNP. Hence, we can conclude that RE is beneficial for the case when the inherent structure of the data is rotationally invariant. This illustrates a massive benefit of the Transformer architecture, as **it is very easy to introduce inductive biases to the Transformer model**, which is not the case for CNNs.

However, as we will see in the next section, if the data given to the model contains samples from many directions, the model will learn to be RE.

6.3.3 Full Sawtooth

The full sawtooth dataset is the sawtooth function which is not restricted to any direction of travel. We observe that the TETNP is able to generalize to the full sawtooth dataset without the need for rotational equivariance Figure 6.3.5.



(a) TETNP Sample 1 (b) TETNP Sample 2 (c) TETNP Sample 3

(d) ConvNP Sample 1 (e) ConvNP Sample 2 (f) ConvNP Sample 3

Figure 6.3.5: Samples from TETNP and ConvNP on the full Sawtooth dataset after training for 7 hours using 1 million parameters models and 64 context points. Context region is inside the black dotted box and target region inside the red dotted box. Out of the target region is the extrapolation region. Standard deviation is omitted for clarity.

The ConvNP performs terribly on the full sawtooth dataset, which may seem surprising at first, but it is not since CNNs are not rotationally equivariant. We hypothesize that the CNN will need to learn filters for the sawtooth in all directions, which is a very difficult task, especially with limited parameters. The ConvNP was able to perform well on the restricted sawtooth dataset as it only needed to learn filters for the sawtooth in *two directions*. Perhaps with longer training times and more parameters, the ConvNP could learn to generalize to the full sawtooth dataset, but this was not considered in this work.

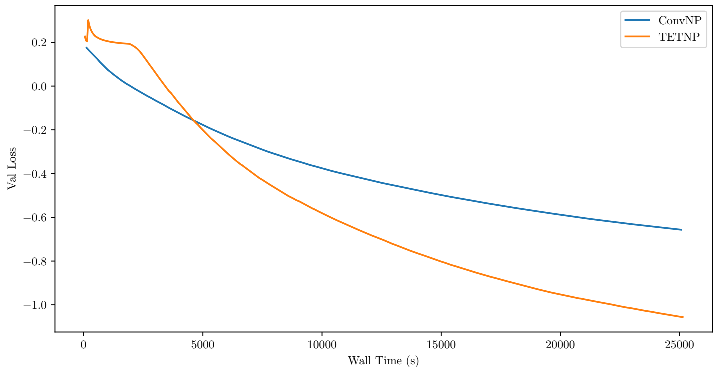


Figure 6.3.6: Validation Loss of ConvNP and TETNP on the 2D Sawtooth Dataset. Lower validation loss is better.

Figure 6.3.6 demonstrates that the TETNP outperforms the ConvNP by a significant margin on the full sawtooth dataset. Overall the TETNP is the clear winner and highlights the power of the Transformer architecture in learning complex patterns in the data.

6.4 Computational Complexity

N_c	N_t	ConvNP Memory (MB)	TETNP Memory (MB)
10	10	24	13
100	10	29	23
1000	10	257	984
5000	10	1273	24082
10	1000	224	26
100	1000	268	113
1000	1000	378	985
5000	1000	1273	24083

Table 6.4.1: Memory usage of ConvNP and TETNP on a 2D dataset with 1 million parameter models. N_c is the number of context points and N_t is the number of target points.

Table 6.4.1 clearly demonstrates that the ConvNPs memory requirement has increased by a factor of 2^3 compared to the 1D case Table 5.4.1, which agrees with the theoretical complexity of the ConvNP. On the other hand, the TETNP retains the same memory requirements as the 1D case, although it remains higher than the ConvNP. When scaling systems to higher dimensions (3D, 4D, etc.), the TETNP will be more memory-efficient than the ConvNP and will likely be the preferred choice.

Chapter 7

Linear Runtime Models

7.1 Introduction

From Table 5.4.1 and Table 6.4.1, it is clear that the Transformer models are memory-intensive due to their quadratic complexity in terms of the dataset size. This is a significant bottleneck for scaling up the Transformer models to larger datasets with limited compute resources. To address this issue, several methods have been proposed to reduce the memory complexity of the Transformer models which we will explore in this chapter.

7.2 Pseudotokens

Pseudotokens (also known as Inducing Points) are a set of tokens that are used to approximate the full set of tokens, it can be thought of as a lower dimensional representation of the data set. They have been widely used in the context of Gaussian Processes (GPs) to reduce the complexity of the model with great success [Hensman, Fusi, and Lawrence 2013]. The original tokens $\boldsymbol{X} \in \mathbb{R}^{N \times D}$ are projected onto into a lower-dimensional space $\boldsymbol{I} \in \mathbb{R}^{M \times D}$ where $M << N$ through some translation equivariant network [Ashman et al. 2024] giving us the pseudotokens $\boldsymbol{I}$ which are translation equivariant to the original tokens $\boldsymbol{X}$. We then perform cross-attention between the pseudotokens $\boldsymbol{I}$ and the original tokens $\boldsymbol{X}$, hence reducing the memory complexity to $\mathcal{O}(MN_c + MN_t)$.

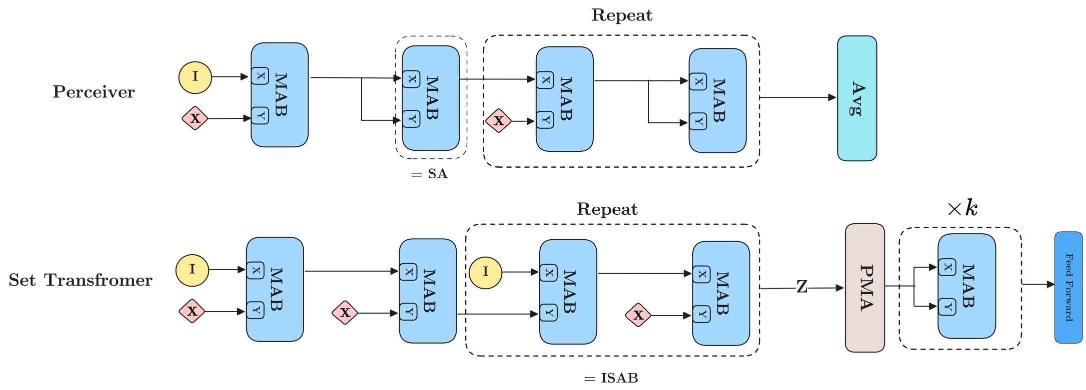


Figure 7.2.1: Perceiver vs Set Transformer. MAB is equivalent to Multi-Head Cross Attention, refer to [Jaegle et al. 2021; Lee et al. 2019] for more details.

We consider two implementations of a pseudotokens based Transformer, one is the Set Transformer [Lee et al. 2019] and the other is the Perceiver [Jaegle et al. 2021]. Both models are very similar and differ in the way they implement the cross-attention mechanism between the original and pseudo tokens Figure 7.2.1. Feng et al. 2023 implemented the Perciever model in the context of NPs creating the 'Latent Bottled Attention Neural Process' (LBANP) model. [Ashman et al. 2024] implemented the Set Transformer model into a NP creating the 'Inducing Set Transformer' (IST) model. Ashman et al. 2024 found both models to perform very similarly. Both models will be encompassed under the 'PT-TETNP' term in the results' section since they are very similar in terms of performance.

7.3 Linear Transformer

Katharopoulos et al. 2020 introduces a kernelized form of the attention mechanism that allows the model to be linear with the number of tokens. The output of an attention head $\boldsymbol{H}$ is computed as follows:

$$\boldsymbol{H}_i = \frac{\phi(\boldsymbol{Q}_i)^T \sum_{j=1}^{N} \phi(\boldsymbol{K}_j)\boldsymbol{V}_j^T}{\phi(\boldsymbol{Q}_i)^T \sum_{j=1}^{N} \phi(\boldsymbol{K}_j)} \tag{7.3.1}$$

Where ϕ is a function that introduces non-linearities, the authors use the ELU function [Clevert, Unterthiner, and Hochreiter 2016]. We only compute $\sum_{j=1}^{N} \phi(\boldsymbol{K}_j)$ and $\sum_{j=1}^{N} \phi(\boldsymbol{K}_j)\boldsymbol{V}_j^T$ once for all the queries $\boldsymbol{Q}_i$, hence reducing the complexity to $\mathcal{O}(N)$. This can simply replace the transformer in the original TNP model, giving us the 'Linear Transformer NP' (LinearTNP).

7.4 HyperMixer

Is attention required for a Transformer model to be effective? The majority of parameters in Transformers are in the MLPs and not the attention mechanism. Tolstikhin et al. 2021 proposes the 'MLP-Mixer' - a Transformer model that replaces the attention mechanism with MLPs across rows and columns of the input. It can be viewed as 'mixing' the features across tokens, thus learning patterns in a way akin to attention. However, the MLP-Mixer requires a **known and fixed input size** which breaks the flexibility of the model if we apply it to a NP.

To overcome this limitation in flexibility, the 'HyperMixer' model [Mai et al. 2023] was proposed which is a variant of the MLP-Mixer that is designed to work with variable input sizes. The model uses hypernetworks [Ha, Dai, and Le 2016] to generate the weights of the MLPs. The hypernetworks $h_k, h_q : \mathbb{R}^{n \times d} \to \mathbb{R}^{n \times p}$ are applied on the queries $\boldsymbol{Q}$ and keys $\boldsymbol{K}$ of the input (row wise) to generate the weights:

$$\boldsymbol{W_k} = h_k(\boldsymbol{K}) = \begin{bmatrix} \mathrm{MLP}_k(\boldsymbol{k}_k) \\ \vdots \\ \mathrm{MLP}_k(\boldsymbol{k}_N) \end{bmatrix} \quad \boldsymbol{W_q} = h_q(\boldsymbol{Q}) = \begin{bmatrix} \mathrm{MLP}_q(\boldsymbol{q}_k) \\ \vdots \\ \mathrm{MLP}_q(\boldsymbol{q}_N) \end{bmatrix} \tag{7.4.1}$$

where $\mathrm{MLP}_k, \mathrm{MLP}_q : \mathbb{R}^d \to \mathbb{R}^p$ transforms the dimension of the input to a lower dimension p. The output of the model is computed as follows:

$$\boldsymbol{H} = \boldsymbol{W_k}\sigma(\boldsymbol{W_q^T V}) \tag{7.4.2}$$

With σ being an activation function, the authors use the GELU function [Hendrycks and Gimpel 2023]. Cross attention is performed by generating the queries from one input and the keys and values from another input. We simply replace attention in the original TNP model with the HyperMixer giving us the 'HyperMixNP' model.

Lack of Translation Equivariance

Both the LinearTNP and HyperMixNP models are not translation equivariant. To introduce TE without using pseudotokens, we require the use of the pairwise differences matrix $\boldsymbol{\Delta}$ (Equation 4.3.3), which would increase the complexity of the model to $\mathcal{O}(N^2)$. It is possible to use pseudotokens in conjunction with these model to achieve translation equivariance however this was not explored in this work.

7.5 Experimental Results

Model	Linear	GP Loss	Sawtooth Loss
HyperMixNP	✓	1.144	-
LinearTNP	✓	1.141	-
PT-TETNP	✓	1.148	-
TETNP	✗	**1.134**	**-1.407**
ConvNP	✗	1.168	-0.8701

Table 7.5.1: Comparison of validation loss on 2D datasets using $N_c = 64$ $N_t = 128$ with 1 million parameter models. All models were trained for 4 hours. Fields with '-' indicate that the model was unable to learn the dataset.

Table 7.5.1 shows the validation loss of the models on the 2D datasets. All the linear models perform similarly to each other and outperform the ConvNP in the GP dataset. However on the Sawtooth dataset, none of the linear models are able to learn the dataset, even with hyperparameters tuning. Such a result is surprising, and potentially indicates that loss of expressive power by simplifying the attention mechanism to be linear. Ultimately, the TETNP massively outperforms all the models.

7.5.1 Computational Complexity

		Linear			Quadratic	
N_c	N_t	HyperMixNP	LinearTNP	PT-TNP	ConvNP	TETNP
10	10	16	16	19	24	13
100	10	16	16	19	29	23
1000	10	23	31	50	257	984
5000	10	69	109	190	1273	24082
10	1000	19	25	51	268	26
100	1000	19	31	51	268	113
1000	1000	24	32	51	268	985
5000	1000	70	109	191	1273	24083

Table 7.5.2: Memory usage of the models in MB under inference using N_c context points and N_t target points on the 2D datasets.

Table 7.5.2 demonstrates the drastic improvement in memory usage of the linear models compared to the quadratic models. The linear models are able to scale to larger datasets with a much smaller memory footprint, making them suitable for large-scale applications where data is smooth, and the quadratic models are infeasible to use.

Chapter 8

Conclusion

In this report we have presented the Neural Process, a general purpose model for uncertainty aware meta-learning. This report highlights the intuition behind the NP and how it achieves all the desired properties of an uncertainty aware predictor.

The standard NP model based on DeepSets was shown to have limited representational capacity, leading to suboptimal performance on complex, high-dimensional datasets. This motivates the need to explore more powerful backbone architectures that have widely been successful in learning representations of data across various domains.

This report introduces two models that extend the NP using CNNs and Transformers, highlighting the intuition behind these models. CNNs are known for their ability to learn spatial structure in data and have been widely successful in image processing tasks. Transformers, on the other hand, have been successful in learning long-range dependencies in sequential data, gaining ubiquity in many Natural Language Processing (NLP) tasks. As of recent Transformers have been applied to a range of tasks beyond NLP, including image processing and reinforcement learning, showing promising results in these domains.

Transformer based NP model (TNP) lacks certain properties that potentially affect its ability to model data with strong stationary patterns or periodicity. Learning these patterns is crucial for modelling many real world tasks such as:

- **Climate modelling**: parameters such as temperature, humidity, and pressure have strong spatial dependencies which exhibit periodic patterns. Accurately modelling these patterns is crucial for predicting weather patterns.
- **Time series forecasting**: Many real world time series data exhibit periodic patterns, such as stock prices, energy consumption, and traffic data which have daily, weekly, and yearly periodicity.
- **Image processing**: Images have strong spatial dependencies, such as textures, edges and blobs which play a large role in characterizing the underlying structure of the data.

The Convolutional Neural Process (ConvNP) model has no issues with learning these properties, due to the convolutional layers learning the spatial structure of the data in a *translation equivariant* manner. Translation equivariance is a key property of CNNs which allows them to learn patterns irrespective of their position in the input data, making them ideal for stationary and periodic data.

This motivates the need for TNP model that is translation equivariant. section 4.3 proposed a novel method for introducing Translation Equivariance into the Transformer model by modifying the attention mechanism to only depend on the relative position of the tokens. This gives us the Transformer Equivariant Neural Process (TETNP) model.

Investigation into the TETNP model on the 1D dataset showed a substantial improvement in performance over the TNP model for the sawtooth dataset, highlighting the importance of translation equivariance when modelling structured data with discontinuities. As a result the TETNP model out performs the ConvNP in the 1D dataset.

Taking the experiments to 2D revealed the TETNP is able to massively outperform the ConvNP, especially on the sawtooth dataset. We highlight that with a simpler 'restricted' version of the Sawtooth dataset, the TETNP does not learn the proper underlying structure of the 'unrestricted' sawtooth dataset. Such behavior the limitation of the Transformer to learn structure without access to the full dataset. On the other hand, the ConvNP performs excellently on the restricted dataset, since the CNNs learn the filters for this simple dataset. On the full dataset, the ConvNP fails, we hypothesize that the CNN struggles to learn the sawtooth filter in all directions, leading to poor performance.

A major drawback of the TNP models is the $\mathcal{O}(N^2)$ complexity of the attention mechanism. We explored several linear runtime approximations to the quadratic attention mechanism. One of them being a pseudotoken models which uses a lower dimensional representation of the input data to reduce the complexity of the attention mechanism.

The other models focused on developing efficient attention mechanisms giving us the Linear Transformer and HyperMixer. While these linear models demonstrated significantly improved computational efficiency, they were unable to match the TETNP's performance on discontinuous datasets, potentially indicating a trade-off between expressively and efficiency.

We conclude that the TETNP model is an excellent choice for training on datasets where we lack little to no prior knowledge of the underlying structure of the data. Though the model is computationally expensive, the performance gains are significant. In higher dimensions, the ConvNP becomes infeasible due to it scaling cubically with dimensionality. The TETNP model does not suffer from this issue, making it a viable choice for high-dimensional data.

Bibliography

Agarap, Abien Fred (2019). *Deep Learning using Rectified Linear Units (ReLU)*. arXiv: 1803.08375 [cs.NE].

Ashman, Matthew, Cristiana Diaconu, Junhyuck Kim, Lakee Sivaraya, Stratis Markou, James Requeima, Wessel P. Bruinsma, and Richard E. Turner (2024). "Translation-Equivariant Transformer Neural Processes". In: *Forty-first International Conference on Machine Learning*. URL: https://openreview.net/forum?id=pftXzp6Yn3.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E. Hinton (2016). *Layer Normalization*. arXiv: 1607.06450 [stat.ML].

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei (2020). *Language Models are Few-Shot Learners*. arXiv: 2005.14165 [cs.CL].

Clevert, Djork-Arné, Thomas Unterthiner, and Sepp Hochreiter (2016). *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*. arXiv: 1511.07289 [cs.LG].

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova (2019). *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. arXiv: 1810.04805 [cs.CL].

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby (2021). *An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale*. arXiv: 2010.11929 [cs.CV].

Feng, Leo, Hossein Hajimirsadeghi, Yoshua Bengio, and Mohamed Osama Ahmed (2022). "Efficient Queries Transformer Neural Processes". In: *Sixth Workshop on Meta-Learning at the Conference on Neural Information Processing Systems*. URL: https://openreview.net/forum?id=_3FyT_W1DW.

— (2023). *Latent Bottlenecked Attentive Neural Processes*. arXiv: 2211.08458 [cs.LG].

Garnelo, Marta, Dan Rosenbaum, Chris J. Maddison, Tiago Ramalho, David Saxton, Murray Shanahan, Yee Whye Teh, Danilo J. Rezende, and S. M. Ali Eslami (2018). *Conditional Neural Processes*. arXiv: 1807.01613 [cs.LG].

Garnelo, Marta, Jonathan Schwarz, Dan Rosenbaum, Fabio Viola, Danilo J. Rezende, S. M. Ali Eslami, and Yee Whye Teh (2018). *Neural Processes*. arXiv: 1807.01622 [cs.LG].

Gordon, Jonathan, Wessel P. Bruinsma, Andrew Y. K. Foong, James Requeima, Yann Dubois, and Richard E. Turner (2020). *Convolutional Conditional Neural Processes*. arXiv: 1910.13556 [stat.ML].

Ha, David, Andrew Dai, and Quoc V. Le (2016). *HyperNetworks*. arXiv: 1609.09106 [cs.LG].

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun (2015). *Deep Residual Learning for Image Recognition*. arXiv: 1512.03385 [cs.CV].

Hendrycks, Dan and Kevin Gimpel (2023). *Gaussian Error Linear Units (GELUs)*. arXiv: 1606.08415 [cs.LG].

Hensman, James, Nicolo Fusi, and Neil D. Lawrence (2013). *Gaussian Processes for Big Data*. arXiv: 1309.6835 [cs.LG].

Jaegle, Andrew, Felix Gimeno, Andrew Brock, Andrew Zisserman, Oriol Vinyals, and Joao Carreira (2021). *Perceiver: General Perception with Iterative Attention*. arXiv: 2103.03206 [cs.CV].

Katharopoulos, Angelos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret (2020). *Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*. arXiv: 2006.16236 [cs.LG].

Kim, Hyunjik, Andriy Mnih, Jonathan Schwarz, Marta Garnelo, Ali Eslami, Dan Rosenbaum, Oriol Vinyals, and Yee Whye Teh (2019). *Attentive Neural Processes*. arXiv: 1901.05761 [cs.LG].

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton (2012). "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger. Vol. 25. Curran Associates, Inc. URL: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Lee, Juho, Yoonho Lee, Jungtaek Kim, Adam R. Kosiorek, Seungjin Choi, and Yee Whye Teh (2019). *Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks*. arXiv: 1810.00825 [cs.LG].

Mai, Florian, Arnaud Pannatier, Fabio Fehr, Haolin Chen, Francois Marelli, Francois Fleuret, and James Henderson (2023). *HyperMixer: An MLP-based Low Cost Alternative to Transformers*. arXiv: 2203.03691 [cs.CL].

Nguyen, Tung and Aditya Grover (2023). *Transformer Neural Processes: Uncertainty-Aware Meta Learning Via Sequence Modeling*. arXiv: 2207.04179 [cs.LG].

Rasmussen, Carl Edward and Christopher K. I. Williams (2006). *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, pp. I–XVIII, 1–248. ISBN: 026218253X.

Simonyan, Karen and Andrew Zisserman (2015). *Very Deep Convolutional Networks for Large-Scale Image Recognition*. arXiv: 1409.1556 [cs.CV].

Tolstikhin, Ilya, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy (2021). *MLP-Mixer: An all-MLP Architecture for Vision*. arXiv: 2105.01601 [cs.CV].

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin (2017). *Attention Is All You Need*. arXiv: 1706.03762 [cs.CL].

Zaheer, Manzil, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola (2018). *Deep Sets*. arXiv: 1703.06114 [cs.LG].

Appendix A

Risk Assessment

Hi