Unifying Transformers and Convolutional Neural Processes

Lakee Sivaraya (ls914)

May 17, 2024

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Desirable Properties . . . 1
1.3 Aims and Objectives . . . 3

2 Neural Processes **4**
2.1 Introduction . . . 4
2.2 Architecture . . . 4
2.2.1 Conditional Neural Processes . . . 4
2.2.2 Latent Neural Processes . . . 6
2.3 Neural Processes vs Gaussian Processes . . . 8
2.4 Performance of Vanilla NP . . . 8

3 Convolutional Neural Processes **9**
3.1 Motivation . . . 9
3.2 Model . . . 10

4 Transformer Neural Processes **12**
4.1 Transformers . . . 12
4.1.1 Introduction . . . 12
4.1.2 Embedding . . . 12
4.1.3 (Self-)Attention . . . 13
4.1.4 Multi-Head Self-Attention . . . 14
4.1.5 Encoder . . . 15
4.2 Vanilla Transformer Neural Process . . . 16
4.2.1 Model Architecture . . . 16
4.2.2 Performance . . . 17
4.3 Efficient Transformer Neural Process . . . 18
4.4 Translation Equivariant Transformer Neural Process . . . 18

5 Experimenation on 1D Datasets **20**
5.1 Datasets . . . 20
5.1.1 Gaussian Process . . . 20
5.1.2 Sawtooth . . . 21

5.2 Relative Attention Function . . . 21
5.3 Optimizing Hyperparameters . . . 22
5.4 TNP vs ConvNP . . . 24
5.4.1 Computational Complexity . . . 25

6 Experimenation on 2D Datasets **27**
6.1 Datasets . . . 27
6.1.1 Gaussian Process . . . 27
6.1.2 Sawtooth . . . 27
6.1.3 Restricted Sawtooth . . . 27
6.2 Post or Pre Relative Attention Function . . . 28
6.3 ConvNP vs TETNP . . . 29
6.3.1 Gaussian Process . . . 29
6.3.2 Restricted Sawtooth and Rotational Equivariance . . . 32
6.3.3 Full Sawtooth . . . 34
6.4 Computational Complexity . . . 36

7 Linear Runtime Models **37**
7.1 Pseudotokens . . . 37
7.1.1 LBANP . . . 37
7.1.2 IST . . . 37
7.1.3 Experimental Results . . . 37
7.2 Linear Transformer . . . 37
7.3 HyperMixer . . . 37

8 Conclusion **38**

Bibliography **39**

Chapter 1

Introduction

1.1 Motivation

Machine learning models have been immensely successful in variety of applications to generate predictions in data-driven driven domains such as computer vision, robotics, weather forecasting. While the success of these models is undeniable, we lack the ability to understand the uncertaincy in the predictions made by most of the State-of-the-Art models. This is a major drawback in the deployment of these models in real-world applications, for example, in weather forecasting, it is important to know the uncertaincy in the prediction of the weather as this information is arguably as valuable as the prediction itself. In this work, we aim to implement a model is **Uncertainty-Aware** whilst also posessing further desirable properties.

1.2 Desirable Properties

Ontop of being uncertaincy aware, we would like to insert some desirable inductive biases that help the model to generalize better and be more interpretable. We would like the model to be able to:

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 work on it. This allows the model to be used in a variety of applications and domains.

Scalable: *The model should be able to work on large datasets.* The model can be trained on large datasets and should be able to make predictions on large datasets. There should be no limit on the size of the dataset that the model can work on which is not the case with many traditional models such as LLMs. Another aspect of scalability is the ability to work on high-dimensional data and large data sets 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 datapoint 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 using 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.* For example, in the case of a weather forecasting model, if the data from the weather stations is shifted by a constant amount, the model should be able to predict the same shift in the weather forecast.

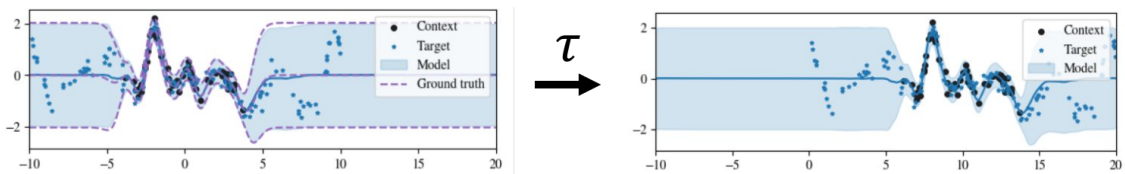


Figure 1.2.1: The Translation Equivariant property. Consider the given prediction on the black datapoints on the left. If the input is shifted by a constant amount, the prediction should also shift by the same amount (right).

TODO

Clearer figure

Figure 1.2.1 illustrates the Translation Equivariant (TE) property on a simple 1D dataset. If the model, f is TE then the following holds:

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

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

Where $\boldsymbol{x}$ is the input and $\hat{\boldsymbol{y}}$ is the output and $\boldsymbol{c}$ is a constant shift in the input.

Off-the-Grid Generalization: *The model should be able to work on off-the-grid data points.* Off-the-grid data points are the data points that are not in a regular gridded structure, for example, images that are missing pixel values are off-the-grid. Traditional models like Convolutional Neural Networks (CNNs) are not able to work 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 different types of data and easily handle missing data points. It also allows the model to generalize to regions of the input space that it has not seen during training. Tasks like image inpainting, where the model is required to fill in missing pixels in an image, can benefit from this property.

TODO

Add a figure to illustrate off-the-grid generalization

1.3 Aims and Objectives

Neural Processes (NPs) [Gar+18b] are a class of models that satisfy the above properties. The framework undermining NPs is general purpose and thus it can be modified with a variety of neural network architectures.

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). We aim to compare the two models on a variety of tasks and datasets to see how they perform in terms of generalization, scalability, and uncertainty estimation.

Furthermore we investiage how the TNP can be optmized to achieve better performance. We then investigate new Transformer architectures that have better computational efficiency compared to the original Transformer architecture.

Our end goal is to better understand the properties of these models and how they can be used in real-world applications and figure out the best practices for using these models in different scenarios.

Chapter 2

Neural Processes

2.1 Introduction

Neural Process (NP) is a meta-learning framework introduced in [Gar+18a; Gar+18b] that can be 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), whilst we will discuss the differences between the two in this chapter, we will focus on the CNP for the majority of the project and hence we will implicitly refer to CNP as NP.

The main idea behind Neural Processes is to learn a distribution over the input locations we want to predict 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 set of context-target pairs and then tested on a new set of context-target pairs to see how well it can generalize to new tasks.

2.2 Architecture

2.2.1 Conditional Neural Processes

Conditional Neural Processes (CNPs) [Gar+18a] was one of the two original Neural Processes introduced by Garnelo et al. in 2018. The general framework for a CNP requires us to take a context set $\mathcal{C} = \{\mathcal{C}_i\}_{i=1}^{N_c}$ where each context point $\mathcal{C}_i$ is a input-output pair $\mathcal{C}_i = (\boldsymbol{x}_i^{(c)}, \boldsymbol{y}_i^{(c)})$ and a target set $\mathcal{T} = \{\boldsymbol{x}_i^{(t)}\}_{i=1}^{N_t}$ where each target point $\boldsymbol{x}_i^{(t)}$ is an input point we want to predict the output for.

We firstly encode each data point in the context set $\mathcal{C}_i$ into an embedding using an encoder network.

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

Where $\boldsymbol{r}$ is the embedding of the context set $\mathcal{C}_i$ and θ are the parameters of the encoder.

Then we process the embeddings of the context sets to get a global representation of the dataset.

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

We require this 'processing' to 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 give us a posterior distribution over the outputs $\boldsymbol{y_i^{(t)}}$.

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

The overall architecture is shown in Figure 2.2.1.

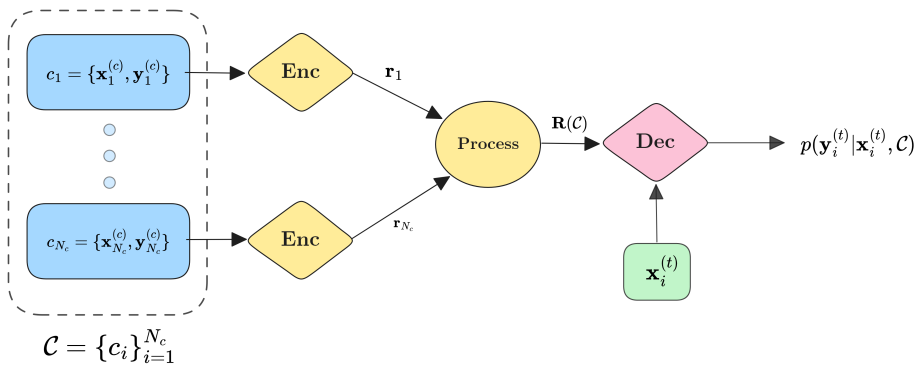


Figure 2.2.1: CNP Architecture: The model vectorizes each individual data point $\mathcal{C}_i$ in the context set $\mathcal{C}$ and then processes/aggregates them to obtain a global representation $\boldsymbol{R}(\mathcal{C})$ which is then used to condition the decoder to predict a distribution over 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, this happens to be an implementation off the 'DeepSet' architecture [Zah+18].

$$\begin{aligned} \text{Enc}_{\theta_e} &= \text{MLP}_{\theta_e} \\ \text{Process} &= \text{Mean} \\ \text{Dec}_{\theta_d} &= \text{MLP}_{\theta_d} \end{aligned}$$

Importantly, CNPs make the strong assumption that the posterior distribution *factorizes* over the target points. This means that the posterior distribution over the target points is independent of each other.

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

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

$$\overset{(c)}{=} \mathcal{N}(\boldsymbol{y}^{(t)}|\boldsymbol{\mu}(\boldsymbol{x}^{(t)}, \mathcal{C}), \boldsymbol{\Sigma}(\boldsymbol{x}^{(t)}, \mathcal{C})) \tag{2.2.6}$$

The benefit of this factorization assumption (a) is that 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) [Gar+18a]. This also adds a further assumption as we have to select a likelihood for the distribution which may not be appropriate for the data we are modeling.

Since the product of Gaussians is a Gaussian (c) the model learns a mean and variance for each target point (though typically we learn the log variance to ensure it is positive), in this way, the model decoder can be interpreted as outputting a function of mean and variance for each target point.

As the likelihood is a Gaussian, the model can be trained using simple maximum likelihood estimation (MLE) by minimizing the negative log-likelihood (NLL) of the target points.

2.2.2 Latent Neural Processes

The 'Latent Neural Processes' is the 2nd variant of NP (introduced in [Gar+18b]) which can generate coherent sample paths and are not restricted to a specific likelihood. They can do this by learning a latent representation of the context set $\mathcal{C}$ which is then used to condition the decoder. We will call this latent representation $\mathbf{z}$ (instead of $\boldsymbol{R}$) to avoid confusion with the non-latent global representation $\boldsymbol{R}$ used in CNPs).

The encoder learns a *distribution* over the latent representation $\mathbf{z}$ of the context set $\mathcal{C}$. Then the decoder learns a distribution over the target outputs $\boldsymbol{y}^{(t)}$ conditioned on the latent representation $\mathbf{z}$ and the target inputs $\boldsymbol{x}^{(t)}$.

$$p(\mathbf{z}|\mathcal{C}) = \mathrm{Enc}_\theta(\mathcal{C})$$
$$p(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(t)}, \mathbf{z}) = \mathrm{Dec}_\theta(\boldsymbol{x}^{(t)}, \mathbf{z})$$

The full model is shown in Figure 2.2.2.

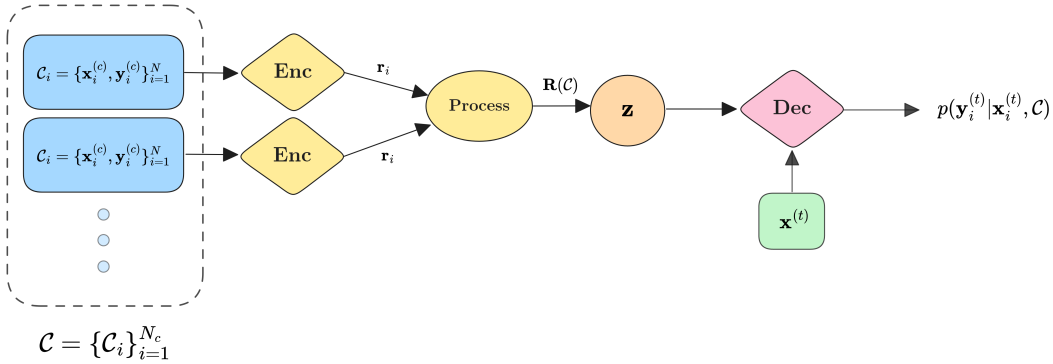


Figure 2.2.2: LNP Architecture

The likelihood of the target points is then given by the marginal likelihood of the latent representation $\mathbf{z}$.

$$p(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(t)}, \mathcal{C}) = \int p(\mathbf{z}|\mathcal{C}) p(\boldsymbol{y}^{(t)}|\boldsymbol{x}^{(t)}, \mathbf{z}) d\mathbf{z} \tag{2.2.7}$$

$$\overset{(a)}{=} \int p(\mathbf{z}|\mathcal{C}) \left(\prod_{i=1}^{N_t} p(y_i^{(t)}|x_i^{(t)}, \mathbf{z}) \right) d\mathbf{z} \tag{2.2.8}$$

$$\overset{(b)}{=} \int p(\mathbf{z}|\mathcal{C}) \left(\prod_{i=1}^{N_t} \mathcal{N}(y_i^{(t)}|\mu_i, \sigma_i^2) \right) d\mathbf{z} \tag{2.2.9}$$

We can see that we express the conditional distribution over the target points conditioned on the latent representation $\mathbf{z}$ as a factorized (a) product of Gaussians (b), this may seem like we are making the same factorization assumption as CNPs. However, the difference is that we are not making this assumption conditioned on the latent variable instead of the context set. This integral models an *infinite mixture* of Gaussians which allows us to model *any* distribution over the target points. The downside is that the likelihood is intractable and we need to use approximate inference methods such as Variational Inference (VI) or Sample Estimation using Monte Carlo (MC) methods to train the model which typically leads to biased estimates of the likelihood with high variance, thus we require more samples to train the model.

Though the LNP has many uses, **in this report we will strictly focus on the CNP** as it is more tractable and easier to train. From this point on, when we refer to Neural Processes we will be referring to the Conditional Neural Process (CNP) unless otherwise stated.

2.3 Neural Processes vs Gaussian Processes

TODO

HERE

2.4 Performance of Vanilla NP

Whilst Neural Processes are very flexible and have the ability to scale, in reality due to the simple archiecture of the encoding and aggregation stage, they are unable to perform effectively in more complicated and higher dimensional data. This is because the model is unable to learn a good representation of the context set using a simple MLP and summation operation.

The benefit of the NP is we can replace the encoder and decoder with more powerful models such as Convolutional Neural Networks (CNNs) or Transformers to learn a better representation of the context set. This allows us to scale the model using well known architectures that have been shown to perform well on a variety of tasks and at scale. Both CNN and Transformer based NPs are bound to have their unique advantages and disadvantages which we will explore in the following chapters by firstly exploring the Convolutional Neural Process (ConvCNP) and then the Transformer Neural Process (TNP).

Chapter 3

Convolutional Neural Processes

TODO

NEED TO DOUBBLE CHECK THIS SECTION, I MIGHT HAVE MISUNDERSTOOD SOME THINGS

3.1 Motivation

When designing models we want them to be able to generalize to unseen data and learn patterns in the data, for example when analyzing 2D climate data, we would want the model to learn relations between the data in the spatio-temporal domain. In other words, we want the model to be able to learn the underlying function of the data without depending on the specific location. A useful property that embodies this requirement is the *translation equivariance* and is a property that is present in many real world problems.

Convolutional Neural Networks (CNNs) are a class of neural networks that are renowned this property, they have been widely used in image processing tasks and have shown to be able to learn spatial patterns in the data via convolutional filters. CNNs are very powerful and have been the state of the art in many image processing tasks making them a desirable backbone for a Neural Process. Hence, the motivation for Convolutional Neural Processes (ConvCNPs) [Gor+20].

3.2 Model

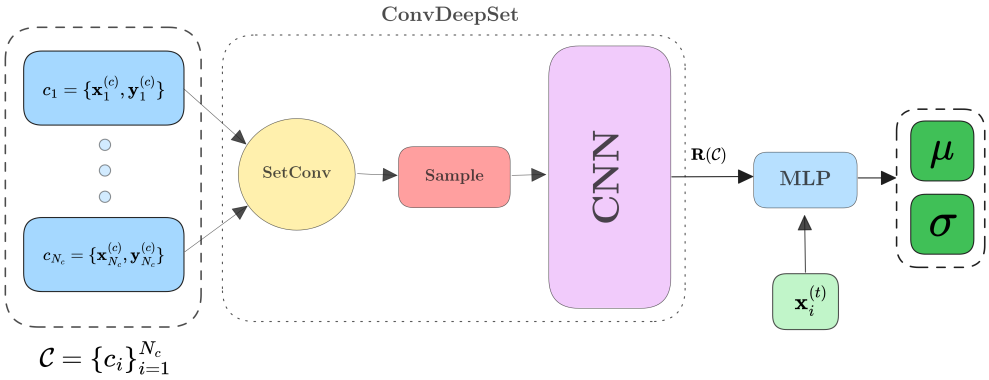


Figure 3.2.1: ConvCNP Architecture

ConvCNPs [Gor+20] Figure 3.2.1 utilize CNNs, more specifically the UNet architecture. However, a few hurdles are preventing us from directly plugging data into a CNN.

CNNs operate on **on-grid data**, meaning that the data is on a regular grid, for example, an image. However, the data we are working with is sometimes **off-grid data**, for example time series dataset with irregular timestamps. Furthermore, to 'bake-in' the TE property, we need to be able to shift the data in the input space and the output space in the same way. This is not trivial when using standard vector representations of the data. [Gor+20] propose a solution to this problem by using **functional embeddings** to model the data. Functional embeddings are a way to represent data as a function that has a trivial translation equivariance property.

These functional embeddings are created using the **SetConv** operation. The SetConv operation is a generalization of the convolution operation that operates on sets of data, it takes a set of input-output pairs and outputs a continuous function. The SetConv operation is defined 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 to the model so that it 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 which means that the model is translation equivariant.
- The summation over the data points naturally introduces **Permutation Invariance** to the model, meaning that the order of the data points does not matter.

In [Gor+20], they refer to the addition of the 1 as the **density channel**, this channel is used to distinguish between observed and unobserved data points.

TODO

Take figure from the ConvCNP paper to show functional embeddings

Now that we have a function representation of the context set, we need to sample it/discretize it at **evenly** spaced points. Thus converting the data into a **on-grid** format. Now it can be fed into a CNN. Now that we have a CNN that operates on the data, we need to convert it back to a continuous function. This is done again 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.

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

The decoder is a simple MLP that takes the output of the encoder and the target set as input and outputs the mean and variance of the predictive distribution. The decoder is defined as follows:

$$\boldsymbol{\mu}(\boldsymbol{x_t}), \boldsymbol{\sigma}(\boldsymbol{x_t}) = \text{MLP}(R(\boldsymbol{x_t})) \tag{3.2.3}$$

Chapter 4

Transformer Neural Processes

4.1 Transformers

4.1.1 Introduction

The Transformer is a deep learning architecture introduced in [Vas+17]. It is a sequence-to-sequence model that uses attention to learn the relationships between the input and output sequences. Transformers have been revolutionary in the field of natural language processing (NLP) leading to the development of models like BERT and GPT. They are also starting tog gain tracking in fields like computer vision and reinforcement learning.

TODO

Add citations to popular transformer models

In this section we will purely focus on the encoder of the transformer model. The encoder is the part of the transformer that processes the input sequence and lairs patterns.

4.1.2 Embedding

Firstly we transform data points it into a vector representation called an *embedding*. Let us denote the embedding of the i-th data point in the input sequence as $\boldsymbol{e}^{(i)} \in \mathbb{R}^{1\times D}$, where D is the feature dimension. The transformer can process these embeddings in **parallel** so we need a way to encode the position of the data in the sequence. In [Vas+17] theyo this by adding a positional encoding $\boldsymbol{p}^{(i)} \in \mathbb{R}^{1\times D}$ to the embedding $\boldsymbol{e}^{(i)}$. The positional encoding is a vector that is unique to the position of the word in the sequence. The positional encoding that is used in [Vas+17] is given by:

$$\boldsymbol{p}_j^{(i)} = \begin{cases} \sin\left(\frac{i}{10000^{j/D}}\right) & \text{if } j \text{ is even} \\ \cos\left(\frac{i}{10000^{(j-1)/D}}\right) & \text{if } j \text{ is odd} \end{cases} \tag{4.1.1}$$

where j is the dimension of the positional encoding. The positional encoding is added to the

embedding as follows:

$$\boldsymbol{x}^{(i)} = \boldsymbol{e}^{(i)} + \boldsymbol{p}^{(i)} \in \mathbb{R}^{1\times D} \tag{4.1.2}$$

These are all stacked together to form the input matrix $\boldsymbol{X} \in \mathbb{R}^{N\times D}$, where $\boldsymbol{X}_{i:} = \boldsymbol{x}^{(i)}$ and N is the number of words in the input sequence. There are other ways to encode the position of the data in the sequence, later on we will concatenate the position and value of the data instead of adding the positional encoding to the embedding.

4.1.3 (Self-)Attention

The attention mechanism is a way to learn the relationships between the input and output sequences. 'Normal' attention infers what the most important word/phrase in an input sentence is which is not very powerful. This is where the idea of *self-attention* comes in. Self-attention is a way to learn the relationships between the data in the input sequence itself (Note when we say attention from now on, we mean self-attention). 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, this gives a powerful mechanism to learn the relationships between the input and output sequences and thus allows the model to learn the translation of the input sequence.

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.

$$\begin{aligned}
\boldsymbol{Q} &= \boldsymbol{X}\boldsymbol{W_q} \in \mathbb{R}^{N\times d_k} \\
\boldsymbol{K} &= \boldsymbol{X}\boldsymbol{W_k} \in \mathbb{R}^{N\times d_k} \\
\boldsymbol{V} &= \boldsymbol{X}\boldsymbol{W_v} \in \mathbb{R}^{N\times d_v}
\end{aligned}$$

Where each row of the matrices is the query, key and value vectors for each word in the input sequence. The query represents the word that we want to compare to the other words in the input sequence, to do this we compare it to the key vectors. The value vectors represent the word that we want to output.

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

$$\boldsymbol{A} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right) \tag{4.1.3}$$

The intuition behind this is that we want to compute the similarity between the query and the key vectors as such we use the 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 follows:

$$\boldsymbol{H} = \boldsymbol{A}\boldsymbol{V} \tag{4.1.4}$$

The overall attention function for a layer is given by:

$$\boldsymbol{H} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{4.1.5}$$

4.1.4 Multi-Head Self-Attention

So far we have only computed the attention matrix $\boldsymbol{A}$ once so the model only learns one attention relationship, however, we can take advantage of using multiple attention 'heads' in parallel to learn many different 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:

$$\boldsymbol{H}^{(i)} = \text{Attention}(\boldsymbol{Q}\boldsymbol{W_q}^{(i)}, \boldsymbol{K}\boldsymbol{W_k}^{(i)}, \boldsymbol{V}\boldsymbol{W_v}^{(i)}) \in \mathbb{R}^{N \times d_v} \tag{4.1.6}$$

Then 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.

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

The figure below summarizes the MHSA operation.

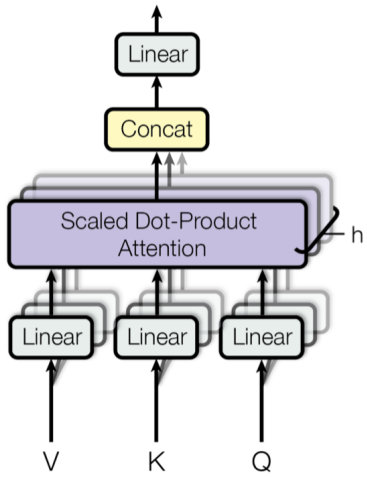


Figure 4.1.1: Multi-Head Self-Attention [Vas+17]

4.1.5 Encoder

Now that we have covered the MHSA block, we can move on to the encoder of the transformer.

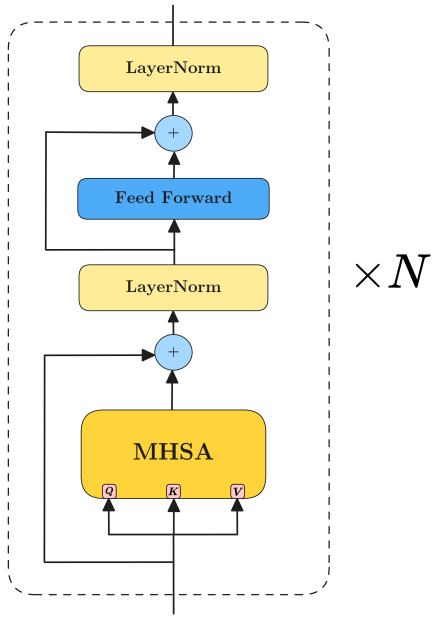


Figure 4.1.2: Transformer Encoder [Vas+17]

Figure 4.1.2 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 $\text{LayerNorm}(x + \text{Sublayer}(x))$ where x is the input to the sub-layer. To dissect this, we first add the input from the sub-layer to the

output of the sub-layer, this is called a *residual* connection. It allows the information from the input to bypass the sublayer and be passed to the next layer, thus preventing the network from losing information about the input. This is then passed through a layer normalization layer. The layer normalization layer normalizes the output (so each row) of the sub-layer to have a mean of 0 and a standard deviation of 1.

The final output of the encoder is the output of the last layer which shall be denoted as $\boldsymbol{Y} \in \mathbb{R}^{N \times D}$. 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.

4.2 Vanilla Transformer Neural Process

An Attention based encoder for the Neural Process was investigated in [Kim+19], though the results were impressive, the model fails to perform at larger scale and 'tends to make over-confident predictions and have poor performance on sequential decision-making problems' [NG23]. It is natural to think that the Transformer [Vas+17] could be used to improve the performance of the Neural Process, as Transformers have been proven to effectively model large scale data using attention mechanisms. [NG23] 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 done by using a simple Multi-Layer Perceptron (MLP) to encode the data points into vector tokens with a configurable token dimension, D_{em}.

Where the TNP differs is that it also encodes the target points which is padded with zeros to represent the lack of value for the target data points, then both context and target tokens are passed into the transformer at the same time instead of computing a context representation and then passing the target points as in the standard NP.

A flag bit is introduced into the tokens to indicate whether the token is a context or target token. Consider 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:

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

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

Now that the data points are tokenized we can pass them through the 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

TODO

ADD FIGURE OF MASKING

With the correct masking in place, we can now pass the tokens through the Transformer encoder layer several times to learn the relationships between the context and target points and generate a vector output. The output of the Transformer encoder is then passed through a Multi-Layer Perceptron (MLP) to generate the mean and variance of the predictive distribution of the target points.

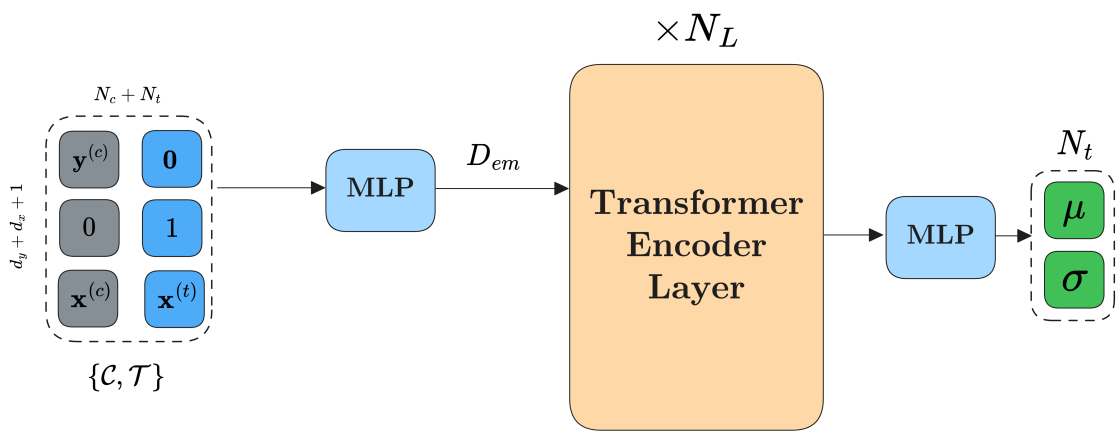


Figure 4.2.1: Vanilla Transformer Neural Process Architecture. All the context and target sets are tokenized and passed through the Transformer Layer. The output of the Transformer Layer is passed through an MLP to generate the mean and variance of the predictive distribution.

4.2.2 Performance

One of the key benefits of Transformers (and attention) is the ability to have a global view of the data, analogous to an 'infinite receptive field' in Convolutional Neural Networks. This allows the model to learn very 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 allow 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 positions of the data do not matter as much as the relative relationships between the data points, e.g. in image classification the position of the object in the image does not matter as much as the features of the object itself. Transformers lack this property, causing them to generate random predictions when the data is shifted out of context as shown in Figure 4.2.2.

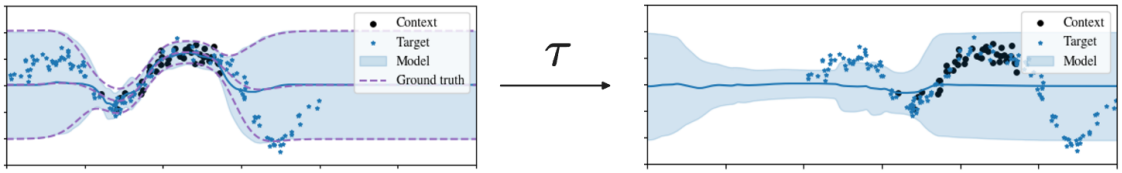


Figure 4.2.2: Vanilla Transformer Neural Process failing to generalize in out of context data. The TNP successfully models the data in the context region (left) but fails to make predictions when the data is shifted (right).

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.

4.3 Efficient Transformer Neural Process

TODO

Add a section on the Efficient Transformer Neural Process

4.4 Translation Equivariant Transformer Neural Process

Recall, 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:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\boldsymbol{E}\right)\boldsymbol{V} \tag{4.4.1}$$

$$\boldsymbol{E_{ij}} = \frac{\boldsymbol{q_i} \cdot \boldsymbol{k_j}}{\sqrt{d_k}} \tag{4.4.2}$$

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:

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}, \boldsymbol{X}) = \text{softmax}\left(\boldsymbol{E} + F(\boldsymbol{\Delta})\right)\boldsymbol{V} \tag{4.4.3}$$

$$\boldsymbol{E}_{ij} = \frac{\boldsymbol{q}_i \cdot \boldsymbol{k}_j}{\sqrt{d_k}} \tag{4.4.4}$$

$$\boldsymbol{\Delta}_{ij} = \boldsymbol{x}_i - \boldsymbol{x}_j \tag{4.4.5}$$

Where F is some function that introduces non-linearity into the attention weights and 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. It is clear to see that if all the input locations are shifted by a constant $\boldsymbol{\tau}$, the relative term will remain the same $\boldsymbol{F}(\boldsymbol{\Delta}_{ij}) = F(\boldsymbol{x}_i + \boldsymbol{\tau} - \boldsymbol{x}_j - \boldsymbol{\tau}) = F(\boldsymbol{x}_i - \boldsymbol{x}_j)$ and the attention weights will remain the same.

Importantly since 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. The tokenization of the data points is now:

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

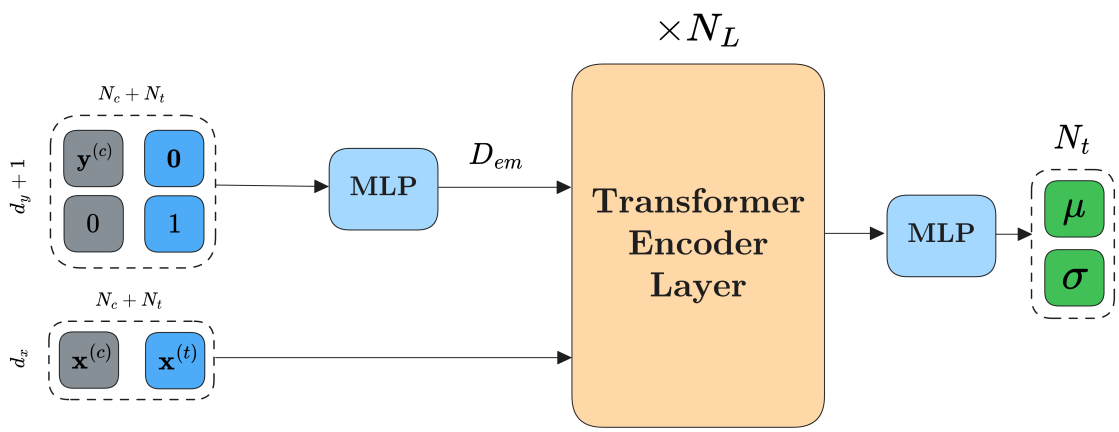


Figure 4.4.1: Translation Equivariant Transformer Neural Process Architecture. All the outputs of the context and target sets are tokenized and passed through the Transformer Layer, whilst the input locations are passed in raw to the attention mechanism to enforce the Translation Equivariance property. The output of the Transformer Layer is passed through an MLP to generate the mean and variance of the predictive distribution.

A benefit of removing the $\boldsymbol{x}$ values from the tokenization is that the tokens only need to encode the $\boldsymbol{y}$ values, reducing the dimensionality of the tokens whilst also separating the encoding for $\boldsymbol{x}$ and $\boldsymbol{y}$ values which could be beneficial for the model as in the vanilla TNP the model loses distinction between the $\boldsymbol{x}$ and $\boldsymbol{y}$ values since they are embedded together.

Chapter 5

Experimenation on 1D Datasets

We will use 1D datasets to firstly optimize the hyperparameters of the TNP models and use this to compare the performance of the TNP and ConvNP models. The metric used to compare the models is the validation loss of the models on unseen data. Furthermore, we will look at the plots of the predictions to observe the behavior of the models especially in the regions outside the training data.

5.1 Datasets

5.1.1 Gaussian Process

We will use samples from a Gaussian Process to generate the data. The aim will be for our model to learn the underlying function of the Gaussian Process and fit the data just like the original Gaussian Process. The Gaussian Process is defined as:

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

Where we use the squared exponential kernel:

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

With length scales being sampled from a uniform distribution $l \sim \mathcal{U}(\log(-0.101), \log(0.101))$. We choose to sample N_c context points from the Gaussian Process and use these as the training data for our models. We then sample N_t target points from the Gaussian Process and use these as the validation data for our models. Depending on the specific task we will either fix or vary the number of context points N_c and target points N_t.

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. We will use the sawtooth function for this purpose. The sawtooth function with period T is defined as:

$$f(x) = x - T \left\lfloor \frac{x}{T} \right\rfloor + n \tag{5.1.3}$$

Where n is some random noise which is sampled from a normal distribution $n \sim \mathcal{N}(0, 0.1)$. We will sample N_c context points from the sawtooth function and use these as the training data for our models. We then sample N_t target points from the sawtooth function and use these as the validation data for our models. Depending on the specific task we will either fix or vary the number of context points N_c and target points N_t.

5.2 Relative Attention Function

As mentioned in section 4.4 we need to pass the matrix of differences ($\mathbf{\Delta}$) between x values through a function F to apply non-linearities to the differences acting as hyperparameter of our mode. We will investigate using a simple linear function with no bias and gradient 1 ('identity') as a baseline. For non-linear functions, we will consider 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 $\text{MLP}(\Delta)$ is a 2-layer MLP with ReLU activation functions.

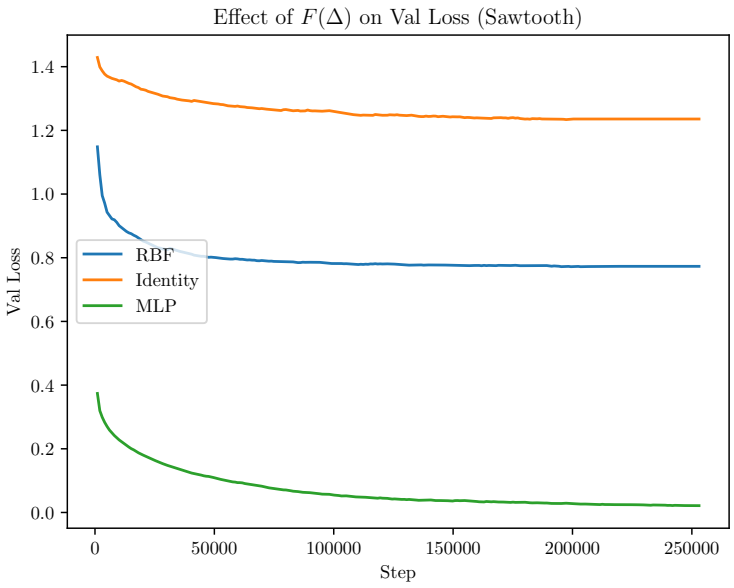
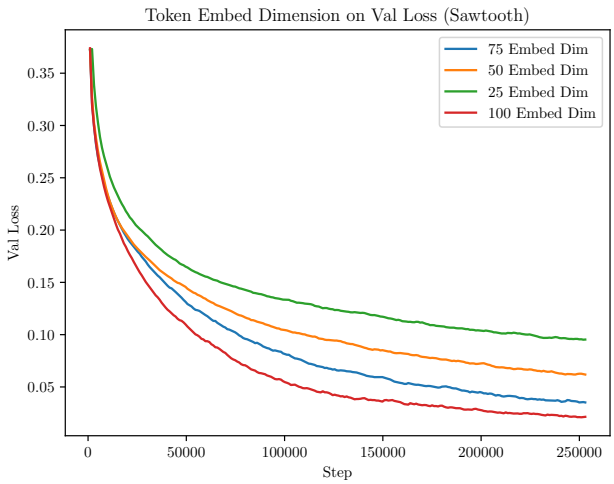


Figure 5.2.1: Relative Attention Functions on Validation Loss for the TETNP on the 1D Sawtooth Dataset. Lower validation loss is better.

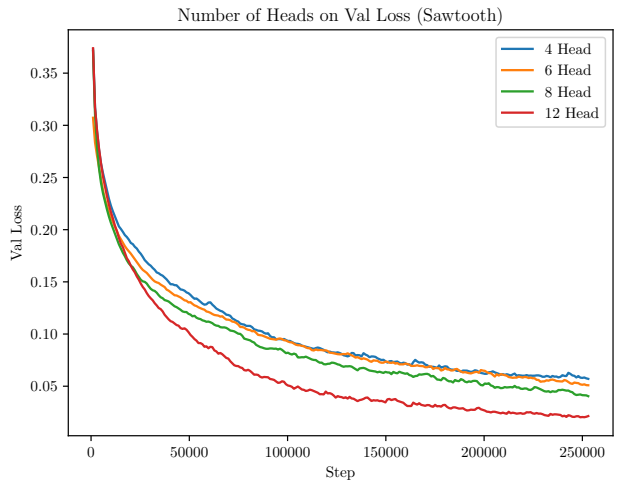
Figure 5.2.1 shows the validation loss curves for the TNP with different relative attention functions. We can see that the MLP function performs significantly better than the other two functions. This is likely due to the MLP being able to **learn** whilst the other two functions are fixed. We can also see that the RBF function performs better than the identify function since the effect of adding the raw difference affects the dot product of the attention mechanism. So we can conclude that the MLP function is the best function to use for the TNP. The computational cost of using the MLP function is also not significantly higher than the other two functions, since the MLP is very small.

5.3 Optimizing Hyperparameters

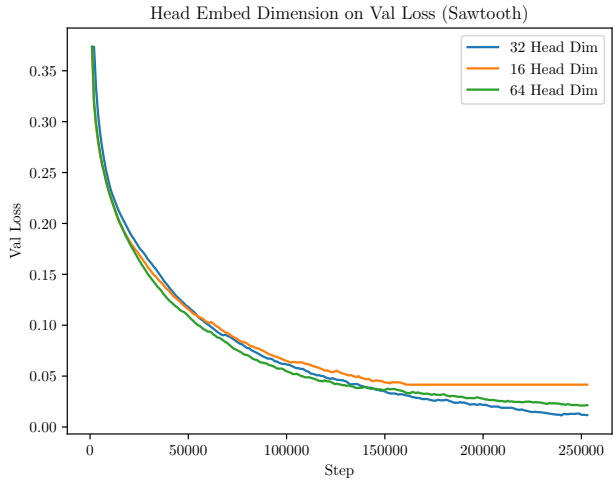
The multi-headed attention mechanism in the Transformer Encoder has three hyperparameters that we will investigate. These are the token embedding dimension of the data (D_{em}), the number of attention heads (N_h) and the embedding dimension of the attention heads (D_h). We will investigate how changing 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

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 even when using low one-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.

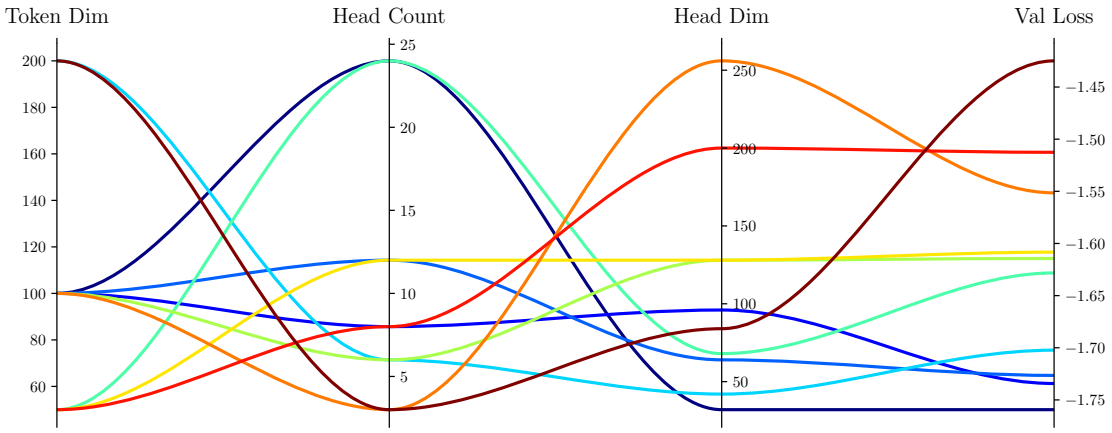


Figure 5.3.2: Constant Parameter Budget Hyperparameter Selection

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. We can see that 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

Finally using our '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.

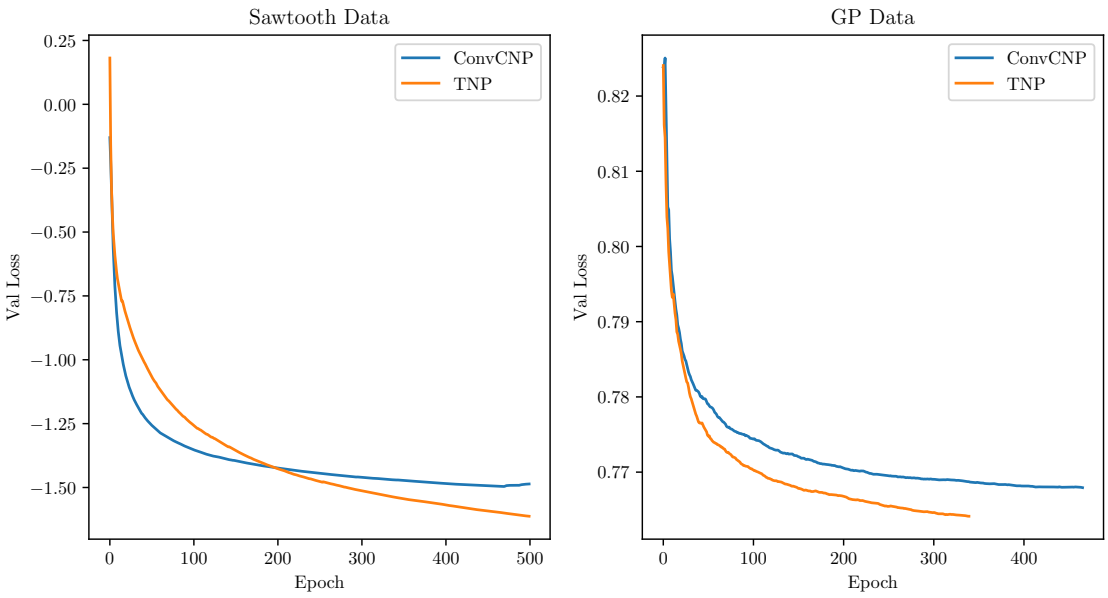
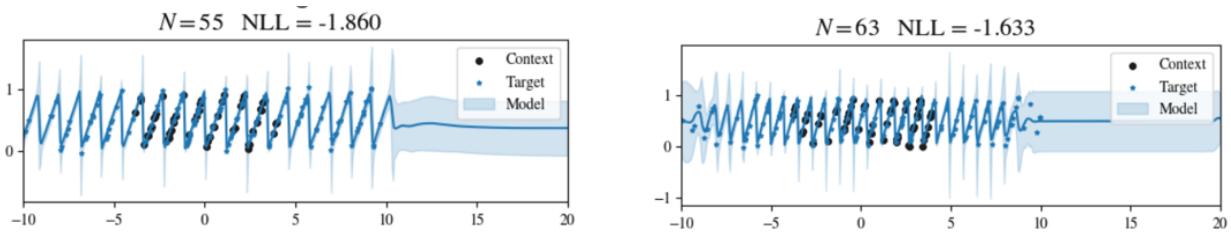


Figure 5.4.1: ConvNP vs TNP on Sawtooth and GP Data

Figure 5.4.1 shows the validation loss curves for the ConvNP and TNP on sawtooth and GP data. We can see that 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

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

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

N_c	N_t	ConvNP Memory (MB)	TETNP Memory (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.

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

Experimenation 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.

The following plots show some samples from the GP dataset.

TODO

Add plots of GP dataset

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.

The following plots show some samples from the Sawtooth dataset.

TODO

Add plots of Sawtooth dataset

6.1.3 Restricted Sawtooth

By accident when generating the 2D Sawtooth dataset, we ended up restricted the 'direction of travel' of the sawtooth function to the line of $x_1 = x_2$ or $x_1 = -x_2$. This was not intentional but when training both models on this dataset, we found very interesting results. As the models only learn a subset of the 'full sawtooth' function, we can see how well the models can generalize to samples from the full sawtooth function.

The following plots show some samples from the Restricted Sawtooth dataset.

6.2 Post or Pre Relative Attention Function

TODO

Highlight MLP

In our original formulation of the TETNP (section 4.4) we pass the matrix of differences ($\boldsymbol{\Delta}$) between x values through a function F to apply non-linearities then add it to the dot product attention Equation 4.4.3, 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 Relative Attention Function'.

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

$$\text{Pre:} \quad \boldsymbol{E}_{ij} = \boldsymbol{D}_{ij} + \text{MLP}(\boldsymbol{\Delta}_{ij}) \tag{6.2.2}$$

$$\text{Post:} \quad \boldsymbol{E}_{ij} = \text{MLP}(\texttt{cat}[\boldsymbol{D}_{ij}, \boldsymbol{\Delta}_{ij}]) \tag{6.2.3}$$

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 Relative Attention Function' compared to the original 'Pre Relative Attention Function'. We choose to use the Sawtooth dataset as it is more difficult to learn than the Gaussian Process dataset and will show the differences between the two models more clearly.

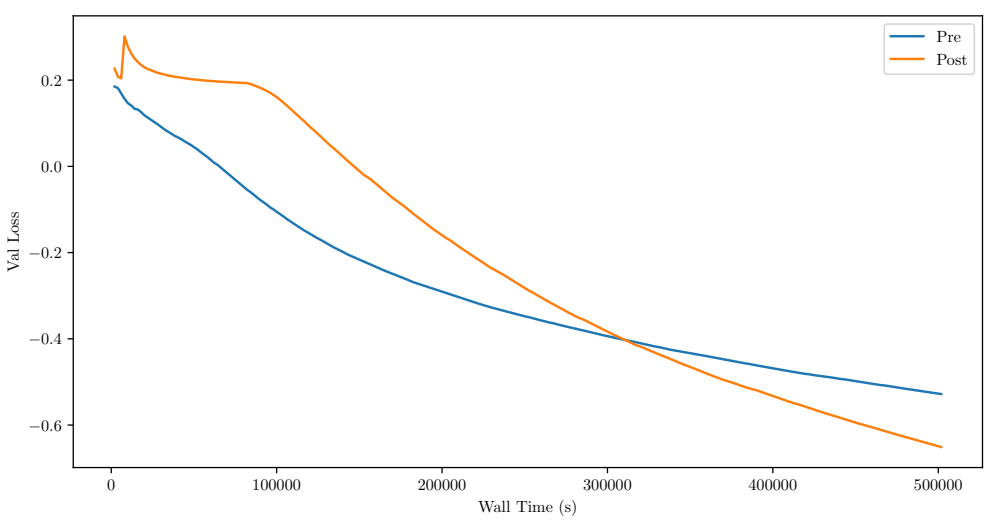


Figure 6.2.1: Validation Loss of TETNP with the 'Post Relative Attention Function' and 'Pre Relative Attention Function' on the 2D Sawtooth Dataset. Lower validation loss is better.

The results show that the TETNP with the 'Post Relative Attention Function' outperforms the TETNP with the 'Pre Relative Attention Function' by quite a large margin. Trivially this makes a lot of sense as 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 $\mathbf{\Delta}$ matrix. The computational complexity of these two functions are not too different as the MLP are small and applied to the same size matrices.

6.3 ConvNP vs TETNP

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.

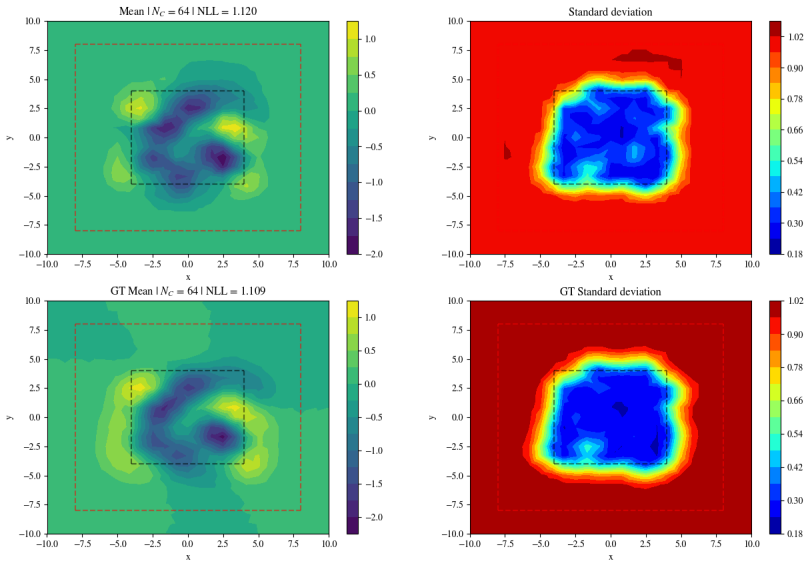
6.3.1 Gaussian Process

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.

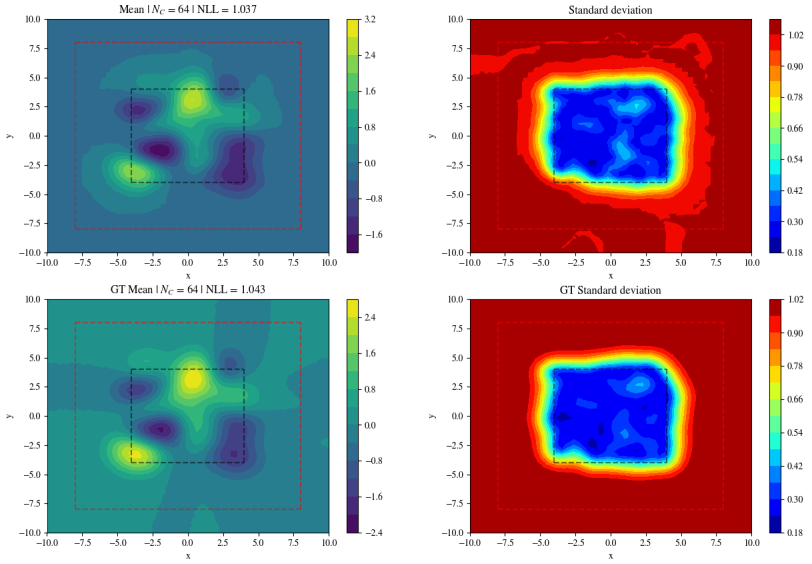
Model	Validation Loss
ConvNP	1.168
TETNP	1.134

Table 6.3.1: Validation Loss of ConvNP and TETNP on the 2D Gaussian Process dataset after training for 3 hours using 1 million parameters models and 64 context points. Lower is better.

Observing the samples from the ConvNP and TETNP for low frequency Figure 6.3.1 and high frequency functions Figure 6.3.2, we can see both models are able to generate very similar predictions to the ground truth GP. The TETNP performs slightly better than the ConvNP in the low frequency case, but the ConvNP performs better in the high frequency case for these samples. However overall the TETNP performs better than the ConvNP on the Gaussian Process dataset.

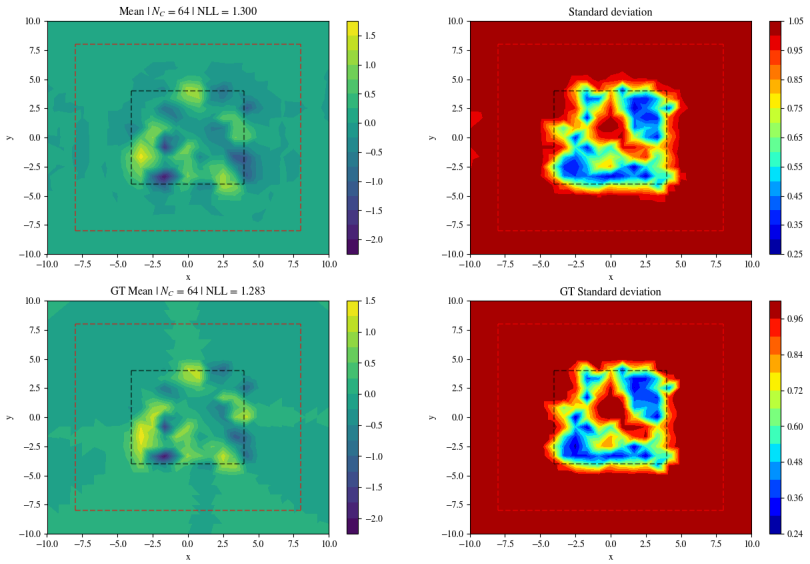


(a) ConvNP (top plot is the model prediction and bottom is the ground truth GP)

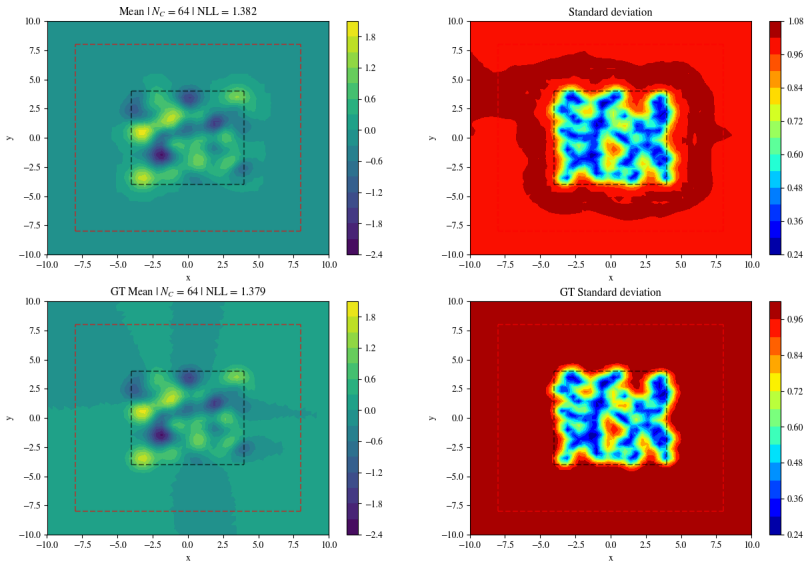


(b) TETNP (top plot is the model prediction and bottom is the ground truth GP)

Figure 6.3.1: Samples from ConvNP and TETNP on a low frequency 2D Gaussian Process.



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

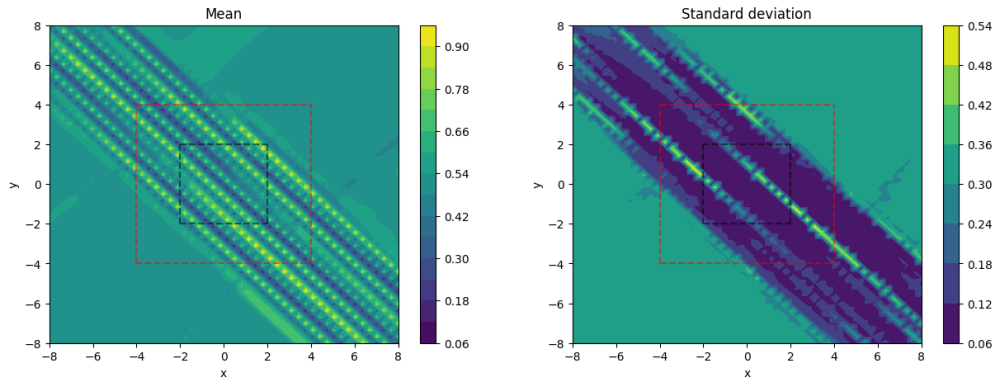


(b) TETNP (top plot is the model prediction and bottom is the ground truth GP)

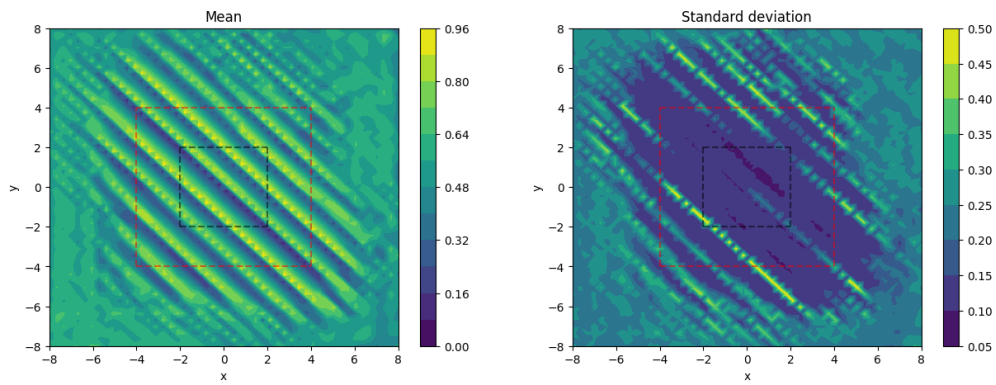
Figure 6.3.2: Samples from ConvNP and TETNP on a high frequency 2D Gaussian Process.

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 $x_1 = -x_2$. By training both models on this dataset we can see how well the models can generalize.



(a) TETNP



(b) ConvNP

Figure 6.3.3: 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.

Figure 6.3.3 shows that the ConvNP performs excellently on this dataset, which is expected as CNNs have filters which learn features and patterns in the data explicitly, thus performs well 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 clearly highlights a limitation of the Transformer architecture which has very little interpretability hence producing weird results.

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.

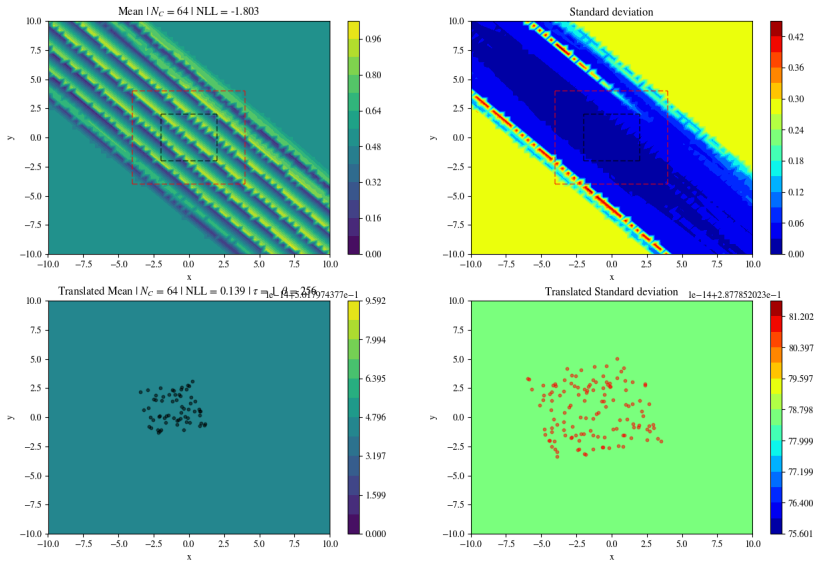


Figure 6.3.4: Samples from TETNP on the full Sawtooth dataset. The top plot is the model prediction on the restricted sawtooth dataset and the bottom plot is the model prediction on the full sawtooth dataset which is a rotated version of the restricted sawtooth dataset. The region inside the black dotted box is the region the model was trained on and the region outside the black dotted box is the region the model was not trained on.

In Figure 6.3.4 we explicitly reduced the target and context region size to allow for the TETNP to cover the full target region. The bottom plot shows the predictions when we rotate the context points by 256 degrees, clearly the TETNP completely fails, it just predicts a constant mean and standard deviation. *Could introducing rotational equivariance to the TETNP help it generalize to the full sawtooth dataset?*

Rotational Equivariance

To introduce rotational equivariance is fairly simple, in our formulation of the Translation Equivariance Attention, we use a matrix $\boldsymbol{\Delta}$ which is the difference 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 see if it can generalize to the full sawtooth dataset.

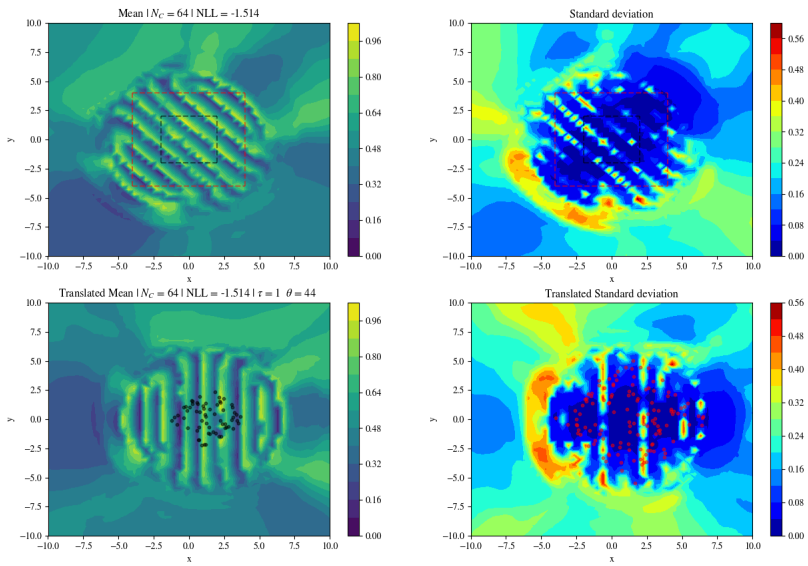


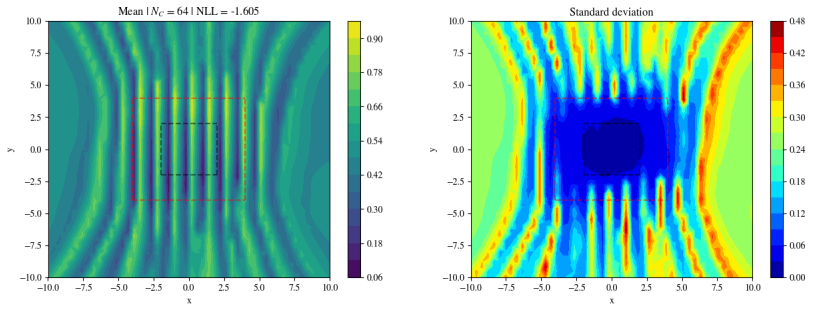
Figure 6.3.5: Samples from TETNP on the full Sawtooth dataset with Rotational The region inside the black dotted box is the region the model was trained on and the region outside the black dotted box is the region the model was not trained on.

Figure 6.3.5 demonstrates a massive improvement in the TETNP's ability to generalize to the full sawtooth dataset. 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.

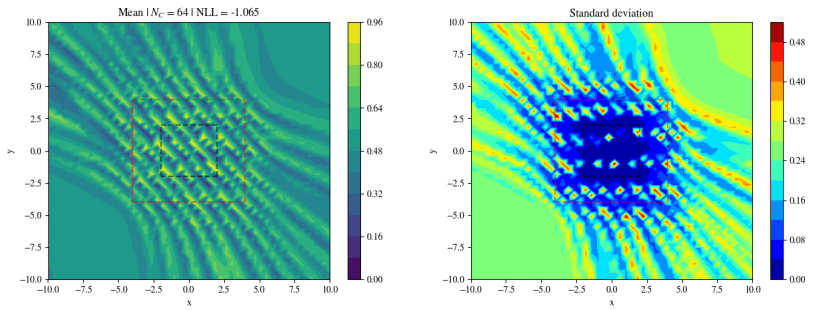
However, as we will see in the next section, if the data given to the model contains samples from many directions, the model will learn to be RE.

6.3.3 Full Sawtooth

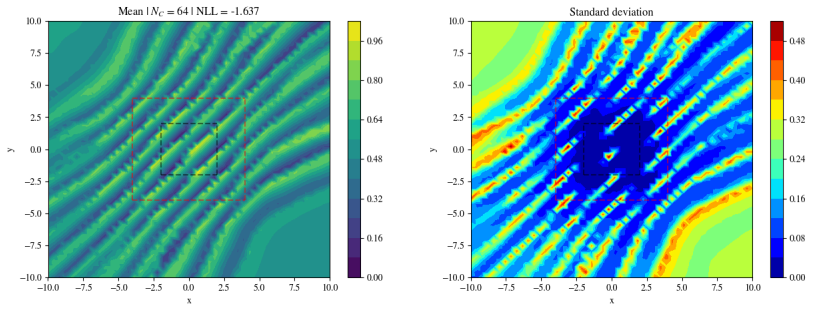
The full sawtooth dataset is the full sawtooth function which is not restricted to any direction of travel. We observe that the TETNP is able to generalize to the full sawtooth dataset without the need for rotational equivariance Figure 6.3.6.



(a)



(b)



(c)

Figure 6.3.6: Samples from TETNP on the full Sawtooth dataset. The region inside the black dotted box is the region the model was trained on and the region outside the black dotted box is the region the model was not trained on.

TODO

GET PLOTS OF CONVNP ON FULL SAWTOOTH AND COMPARE RESULTS!!!!!!!!!!!!
!
!
!
!

6.4 Computational Complexity

TODO

Run the code for this
!
!
!
!

Chapter 7

Linear Runtime Models

7.1 Pseudotokens

7.1.1 LBANP

7.1.2 IST

7.1.3 Experimental Results

7.2 Linear Transformer

7.3 HyperMixer

Chapter 8

Conclusion

Bibliography

[Gar+18a] Marta Garnelo et al. *Conditional Neural Processes*. 2018. arXiv: 1807.01613 [cs.LG].

[Gar+18b] Marta Garnelo et al. *Neural Processes*. 2018. arXiv: 1807.01622 [cs.LG].

[Gor+20] Jonathan Gordon et al. *Convolutional Conditional Neural Processes*. 2020. arXiv: 1910.13556 [stat.ML].

[Kim+19] Hyunjik Kim et al. *Attentive Neural Processes*. 2019. arXiv: 1901.05761 [cs.LG].

[NG23] Tung Nguyen and Aditya Grover. *Transformer Neural Processes: Uncertainty-Aware Meta Learning Via Sequence Modeling*. 2023. arXiv: 2207.04179 [cs.LG].

[Vas+17] Ashish Vaswani et al. *Attention Is All You Need*. 2017. arXiv: 1706.03762 [cs.CL].

[Zah+18] Manzil Zaheer et al. *Deep Sets*. 2018. arXiv: 1703.06114 [cs.LG].