Unifying Transformers and Convolutional Neural Processes

Lakee Sivaraya (ls914)

May 14, 2024

Contents

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.

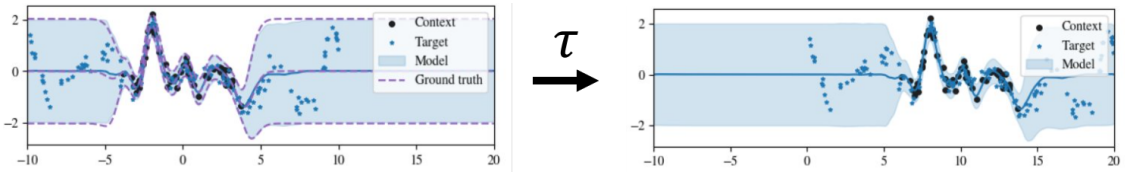

Figure 1.2.1: The Translation Equivariant property. Consider the given prediction on the black datapoints on the left. If the input is shifted by a constant amount, the prediction should also shift by the same amount (right).

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+18b; Gar+18a] that can be used for few-shot uncertaincy 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 target points conditioned on the context points. Context points are the input-output pairs that the model is trained on and is used to condition the model to predict the target points, which are the input locations we want to predict the output for. The model is trained on a sets 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

A meta-dataset is split into two sets, the context set and the target set (also called the query set) where the sets are disjoint $\mathcal{D} = \mathcal{C} \cup \mathcal{T}$ and $\mathcal{C} \cap \mathcal{T} = \emptyset$. The model is trained on the context set and then tested on the target set to see how well it can generalize to new tasks. In essence, our task is to predict the outputs for the target conditioned on the training of the context set.

To achieve this we used a neural network (usually MLP) to encode the context set into embeddings.

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

Where $\boldsymbol{r}$ is the embedding of the context set $\mathcal{C}_i$ and θ are the parameters of the encoder. Then we aggregate the embeddings of the context sets to get a global representation of the

dataset.

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

Typically the aggregation function 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 $\mathcal{T} = \{\boldsymbol{x}_i^{(t)}\}$ 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.

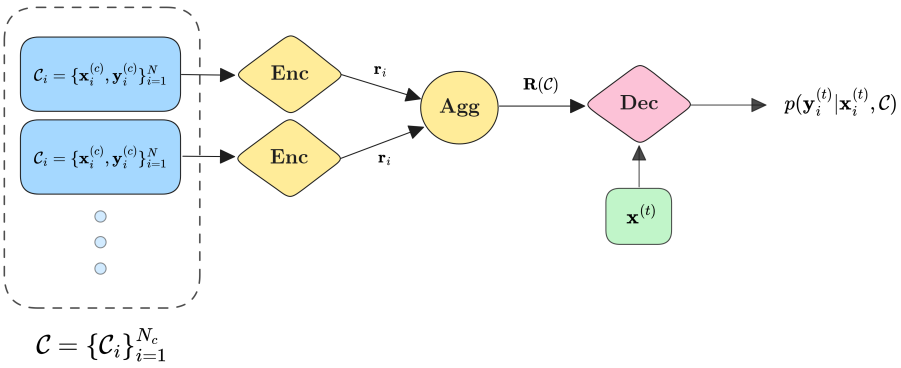

Figure 2.2.1: CNP Architecture

2.2.1 Conditonal Neural Processes

Conditional Neural Processes (CNPs) [Gar+18a] was one of the two original Neural Processes introduced by Garnelo et al. in 2018. Using the architecture framework described above, CNPs can be described as follows.

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

The encoding and summation is an implementation of the 'DeepSet' architecture [Zah+18] which is a neural network that is **Permutation Invariant** (PI) due to the summation operation. This means that the order of the input does not matter.

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

Neural Processes better named 'Latent Neural Processes' are an extension of CNPs that 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 $\boldsymbol{x}^{(t)}$.

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

The full model is shown in Figure 2.2.2.

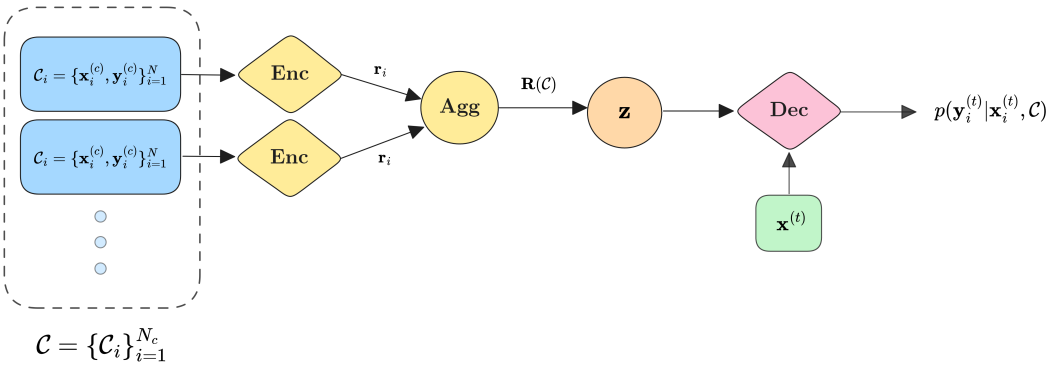

Figure 2.2.2: LNP Architecture

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.

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

3.1 ConvCNP

3.1.1 Background

When training AI models we want them to be able to generalize to unseen data and learn patterns in the data, for example when analysing 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. This property is called *translation equivariance* and is a property that is present in many real world problems. Translation Equivariance (TE) states that if our inputs are shifted in the input space, the output should be shifted in the output space in the same way. CNPs do not have this property, the ConvCNP aims to add this property to the CNP by utilizing the TE property of CNNs.

3.1.2 Model

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

Encoder

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**, meaning that the data is not on a regular grid, 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, as the data is not on a regular grid. [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:

$$\text{SetConv}(\{x_i, y_i\}_{i=1}^N)(x) = \sum_{i=1}^N [1, y_i]^T \mathcal{K}(x - x_i) \tag{3.1.1}$$

Where $\mathcal{K}$ is a kernel function that measures the distance between the queryable x and the datapoint x_i.

This operation has some key properties:

- We append 1 to output 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 datapoint of $y_i = 0$, then if we did not append the 1, the model would not be able to distinguish between an observed datapoint and an unobserved datapoint (as both would be 0).
- The 'weight' of the Kernal depends only on the relative distance between points on the input space which means that the model is translation equivariant.
- The summation over the datapoints naturally introduces **Permutation Invariance** to the model, meaning that the order of the datapoints does not matter.

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

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.

$$R(x_t) = \text{SetConv}(\text{CNN}(\{\text{SetConv}(\mathcal{C})(x_d)\}_{d=1}^D))(x_t) \tag{3.1.2}$$

3.1.3 Decoder

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:

$$\mu(x_t), \sigma^2(x_t) = \text{MLP}(R(x_t)) \tag{3.1.3}$$

Chapter 4

Transformer Neural Processes

4.1 Transformers

4.1.1 Introduction

Figure 0.2.1 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.

In this report we will follow the notation that is used in [Vas+17] where embeddings are represented as rows $\in \mathbb{R}^{1\times D}$ and all matrix multiplications are right multiplications. Subscripts that are not bolded and lowercase are used to index vectors. Superscripts that are surrounded by parenthesis are used to index the position of a vector/matrix within a sequence, layer or head.

4.1.2 Embedding

A machine is not capable of understanding wordings, hence we need to transform it into a vector representation called an *embedding*. Let us denote the embedding of the i-th word 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 word in the sequence. We do 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.

Alternative Positional Encoding

There are many ways to go about positional encoding. Another way is to use a learned positional encoding. This is done by adding a learnable vector $\boldsymbol{p}^{(i)} \in \mathbb{R}^{1\times D}$ to the embedding. Alternatively, to achieve translation equivariance, we can use *Relative Positional Encoding* [SUV18; Wu+21]. These positional encoding schemes will be useful when trying to build equivariance into the Transformer model. See [Kaz19] for a comparison of different positional encoding schemes.

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 words in the input sequence itself (Note when we say attention from now on, we mean self-attention). The example sentence `The quick brown fox jumps over the lazy dog` has strong attention between the words 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 words `quick` and `dog` representing the lack of relationship between the two words. Using self-attention we can learn these relationships between the words 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 used 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:

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

Attention Mechanisms

There are many ways to compute the attention matrix $\boldsymbol{A}$. The one that is used in the original transformer paper is called *Scaled Dot-Product Attention*. Other attention mechanisms may be of interest, see [Wen18] for a comparison of different attention mechanisms.

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, $\boldsymbol{W}_{\boldsymbol{q}}^{(i)} \in \mathbb{R}^{d_k \times d_k}$, $\boldsymbol{W}_{\boldsymbol{k}}^{(i)} \in \mathbb{R}^{d_k \times d_k}$ and $\boldsymbol{W}_{\boldsymbol{v}}^{(i)} \in \mathbb{R}^{d_v \times d_v}$ where $i \in [1, h]$ for a head count of h. Then the attention for the particular head is computed as follows:

$$\boldsymbol{H}^{(i)} = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_{\boldsymbol{q}}^{(i)}, \boldsymbol{K}\boldsymbol{W}_{\boldsymbol{k}}^{(i)}, \boldsymbol{V}\boldsymbol{W}_{\boldsymbol{v}}^{(i)}) \in \mathbb{R}^{N \times d_v} \tag{4.1.6}$$

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 $\boldsymbol{W}_{\boldsymbol{O}} \in \mathbb{R}^{hd_v \times D}$ 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.

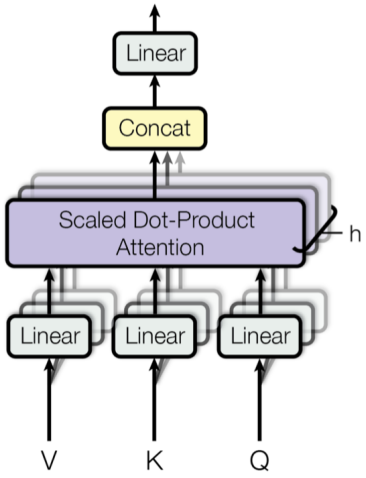

Figure 4.1.1: Multi-Head Self-Attention [Vas+17]

4.1.5 Encoder

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

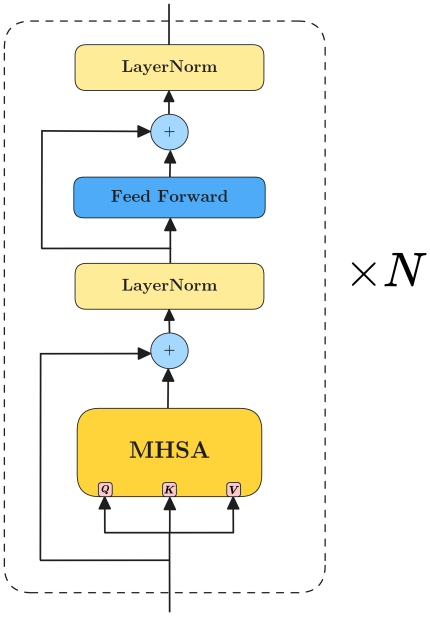

Figure 4.1.2: Transformer Encoder [Vas+17]

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. e.g consider a row of the output of the sub-layer $\boldsymbol{h} \in \mathbb{R}^{1\times D}$, the layer normalization layer will normalize this row as follows:

$$\begin{aligned}
\boldsymbol{h}_i^{(norm)} &= \frac{\boldsymbol{h}_i - \mu}{\sigma} \\
\mu &= \frac{1}{D}\sum_{i=1}^{D} h_i \\
\sigma &= \sqrt{\frac{1}{D}\sum_{i=1}^{D}(h_i - \mu)^2}
\end{aligned}$$

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

4.1.6 Decoder

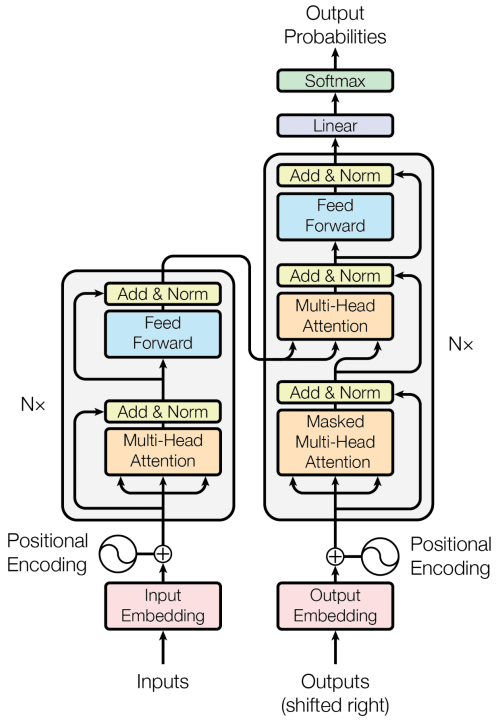

Figure 4.1.3: Transformer [Vas+17]

The transformer architecture is shown in Figure 4.1.3, the decoder is very similar to the encoder, with a few small differences. We now have an additional block called the *Masked Multi-Head Self-Attention* (M-MHSA). The M-MHSA is identical to the MHSA except that the attention matrix $\boldsymbol{A}$ is masked so that the decoder can only attend to previous positions in the sequence. This is done by setting the value of the scaled dot product $\boldsymbol{QK^T}/\sqrt{d_k}$ to $-\infty$ for all positions in the sequence that are greater than the current position. Then when the softmax is applied, these values will be 0 and thus the decoder will not attend to these positions. An example of this is shown below:

$$\begin{bmatrix} 0.2 & 0.3 & 0.5 & 0.1 \\ 0.1 & 0.2 & 0.7 & 0.0 \\ 0.3 & 0.4 & 0.2 & 0.1 \\ 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0.4759 & 0.3092 & 0.2148 & 0 \\ 0.2824 & 0.3072 & 0.3333 & 0.0771 \end{bmatrix} \tag{4.1.7}$$

This ensures that the decoder can only attend to previous positions in the sequence. The M-MHSA is used in the first sub-layer of the decoder.

The second sub-layer of the decoder is the MHSA, this is identical to the MHSA in the encoder however the keys and values come from the **encoder output** $\boldsymbol{Y}$ and the queries

come from the **output of the M-MHSA sub-layer**. Intutively the encoder learns the attention of the input sequence and the decoder learns how to use query the encoder output to generate the output sequence.

$$\text{MHSA}_{dec} = \text{MHSA}(\boldsymbol{Q_{dec}}, \boldsymbol{K_{enc}}, \boldsymbol{V_{enc}})$$

The output of the MHSA_{dec} is then passed through a feed-forward network and layer normalization layer as per usual. This is repeated N times and the output of the final layer is the output of the decoder. This final output is then passed through a linear layer which transforms the decoder output $\in \mathbb{R}^{N \times D}$ to the output vocabulary size $\in \mathbb{R}^{N \times \mathcal{V}}$ where $\mathcal{V}$ is the vocabulary of the output language i.e the set of all possible words in the output language. The output of this linear layer is then passed through a softmax layer to generate the final predictive probability distribution over the output vocabulary. We select the word with the highest probability as the predicted word.

4.1.7 Training

The transformer is trained using teacher forcing. Teacher forcing is a technique used in sequence-to-sequence models where the model is trained to predict the next token in the sequence given the previous tokens. This is done by feeding the model the previous tokens and then the next token in the sequence. For example if we want to translate the sentence `I live in Paris` to French, we would feed the model the tokens `I` in the encoder and a start token `<start>` in the decoder and see if it can predict the translated token `Je`, we then feed correct token `Je` into the decoder and see if it can predict the next token `vis` and so on. This forces the model to learn the correct translation of the sentence as we use the correct target tokens to generate a loss function ($\mathcal{L}$) which the transformer decoder optimizes.

The table below shows the training procedure for the translation example.

Encoder Input	Decoder Input	Loss
`I`	`<start>`	$\mathcal{L}$(`pred1`, `Je`)
`live`	`Je`	$\mathcal{L}$(`pred2`, `vis`)
`in`	`vis`	...
...	...	...

4.1.8 Inference

After we have trained our model, we can use it for inference. The methodology is similar to the training, however, the decoder output receives the previously predicted token as input instead of the correct target token. This is shown in the table below.

Encoder Input	Decoder Input	Decoder Output
I	<start>	pred1
live	pred1	pred2
in	pred2	...
...	...	...

4.2 Transformer Neural Processes

4.2.1 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.

Model Architecture

Similar to the standard Neural Process architecture we are required to encode datapoints within the context set into a vector representation / tokens. Where the TNP differs is that it also encodes the target points along which is padded with zeros to represent the lack of value for the target data points. We introduce a flag bit 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 datapoint 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. Now that the datapoints 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

4.2.2 Requirements

As NPs are a meta-learning method they should be trained on a context set which is a subset of a larger dataset. In this example, say we sample N $x-y$ pairs from a process $\mathcal{F}$, we designate the first N_c pairs as the context set and the remaining $N-N_c$ pairs as the target set, or mathematically $\mathcal{C}=\{(x_i,y_i)\}_{i=1}^{N_c}$ and $\mathcal{T}=\{(x_i,y_i)\}_{i=N_c+1}^{N}$. The objective function is the maximum likelihood estimation of the target set given the context set autoregressively, or mathematically on the target set.

$$\mathcal{L}(\theta)=\mathbb{E}_{(x,y)\sim\mathcal{F}}\left[\log p(y_{N_c+1:N}|x_{N_c+1:N},\mathcal{C};\theta)\right] \tag{4.2.1}$$

Where θ are the parameters of the model and $y_{N_c+1:N}$ and $x_{N_c+1:N}$ are the target set. Since we pass the target set through the model autoregressively, the objective function can be rewritten as:

$$\mathcal{L}(\theta)=\mathbb{E}_{(x,y)\sim\mathcal{F}}\left[\sum_{i=N_c+1}^{N}\log p(y_i|x_i,\mathcal{C},x_{N_c+1:i-1},y_{N_c+1:i-1};\theta)\right] \tag{4.2.2}$$

Where each conditional is modeled as a Gaussian distribution with mean and variance predicted by the model.

There are two important properties of TNPs that we need to build upon:

- **Context Invariance**: If we permute the context set, the model should not change its predictions
- **Target Equivariance** If we permute the target set, the model should not change its predictions, e.g. we shift the target set by one point to the left, and the model's predictions should also shift by one point to the left.

4.2.3 Autoregressive Transformer Neural Processes (TNP-A)

We can not implement the traditional Transformer as the inclusion of the positional encoding breaks the context invariance property since the positional encoding is dependent on the position of the points, but we need to allow the model to learn the relationships between x and y **pairs**. To do this, [NG23] concatenates the x and y pairs together into a single vector for the context set and target set. They then add auxiliary tokens from the target set but with $y=0$ such that the dataset looks like

$$(x_1,y_1),...,(x_N,y_N),(x_{N_c+1},0),...,(x_N,0)$$

As we can see the x values from the target set effectively appear twice, the auxiliary tokens are used to represent the target set and then the corresponding y values are used to train the model since those tokens are not auxiliary. To make this work, we must employ a masking scheme to allow:

- Context tokens to attend to themselves
- Target tokens to attend to context and previous target tokens
- Auxillary tokens to attend to context and previous target tokens

This model is referred to as the Autoregressive Transformer Neural Process (TNP-A) in [NG23].

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].

[Kaz19] Amirhossein Kazemnejad. "Transformer Architecture: The Positional Encoding". In: *kazemnejad.com* (2019). URL: https://kazemnejad.com/blog/transformer_architecture_positional_encoding/.

[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].

[SUV18] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. *Self-Attention with Relative Position Representations*. 2018. arXiv: 1803.02155 [cs.CL].

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

[Wen18] Lilian Weng. "Attention? Attention!" In: *lilianweng.github.io* (2018). URL: https://lilianweng.github.io/posts/2018-06-24-attention/.

[Wu+21] Kan Wu et al. *Rethinking and Improving Relative Position Encoding for Vision Transformer*. 2021. arXiv: 2107.14222 [cs.CV].

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