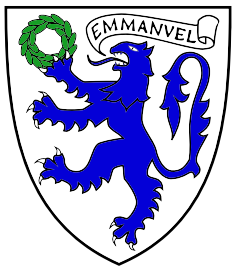
Unifying Transformers and Convolutional Neural Processes

Lakee Sivaraya (ls914)

October 19, 2023

Emmanuel College



Contents

1 Transformers

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.

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.

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:

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

where j is the dimension of the positional encoding. The positional encoding is added to the embedding as follows:

$$\boldsymbol{x}^{(i)} = \boldsymbol{e}^{(i)} + \boldsymbol{p}^{(i)} \in \mathbb{R}^{1\times D} \tag{2}$$

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.

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:

$$\boldsymbol{A} = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right) \tag{3}$$

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:

$$\boldsymbol{H} = \boldsymbol{A}\boldsymbol{V} \tag{4}$$

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:

$$\boldsymbol{H} = \text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}}\right)\boldsymbol{V} \tag{5}$$

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:

$$\boldsymbol{H}^{(i)} = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_{\boldsymbol{q}}^{(i)}, \boldsymbol{K}\boldsymbol{W}_{\boldsymbol{k}}^{(i)}, \boldsymbol{V}\boldsymbol{W}_{\boldsymbol{v}}^{(i)}) \in \mathbb{R}^{N \times d_v} \tag{6}$$

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.

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

The figure below summarizes the MHSA operation.

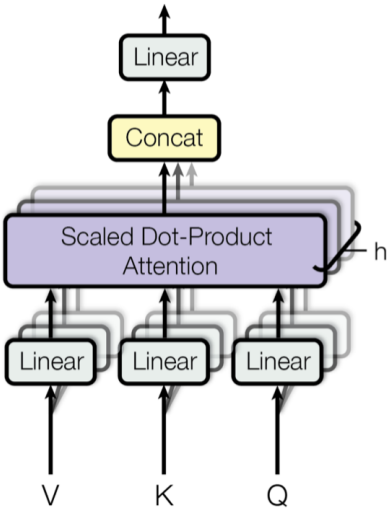


Figure 1.1: Multi-Head Self-Attention [Vas+17]

1.5 Encoder

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

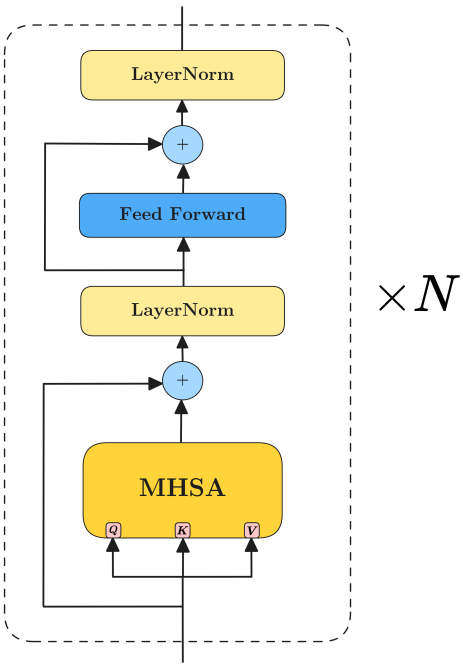


Figure 1.2: Transformer Encoder [Vas+17]

Figure 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:

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

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

1.6 Decoder

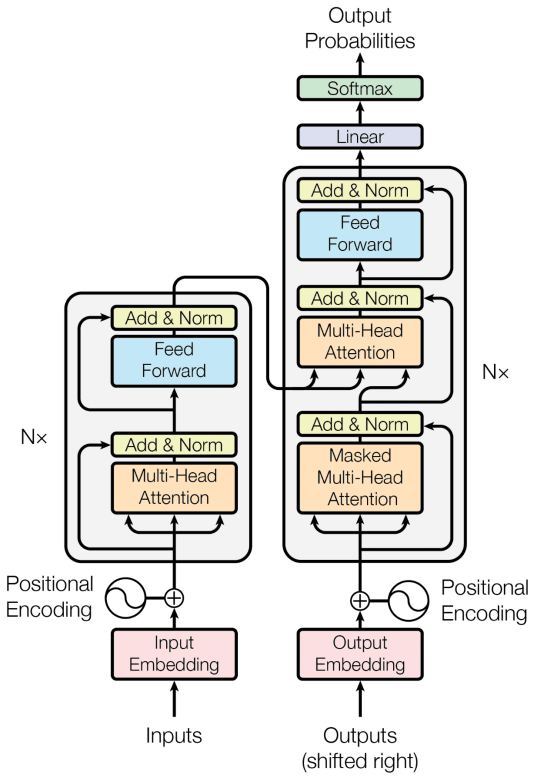


Figure 1.3: Transformer [Vas+17]

The transformer architecture is shown in Figure 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:

$$\begin{bmatrix} 0.2 & 0.3 & 0.5 & 0.1 \\ 0.1 & 0.2 & 0.7 & 0.0 \\ 0.3 & 0.4 & 0.2 & 0.1 \\ 0.1 & 0.2 & 0.3 & 0.4 \end{bmatrix} + \begin{bmatrix} 0 & -\infty & -\infty & -\infty \\ 0 & 0 & -\infty & -\infty \\ 0 & 0 & 0 & -\infty \\ 0 & 0 & 0 & 0 \end{bmatrix} \xrightarrow{\text{softmax}} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 0 \\ 0.4759 & 0.3092 & 0.2148 & 0 \\ 0.2824 & 0.3072 & 0.3333 & 0.0771 \end{bmatrix} \tag{7}$$

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.

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`	...
...	...	...

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`	...
...	...	...

2 Neural Processes

Neural Processes are an expansion of Gaussian Processes using Neural Networks introduced in [Gar+18b] they are trained on a context dataset which is. Neural processes are a meta-learning algorithm that can be used for few-shot learning. They are trained on a context dataset which is a set of input-output pairs $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^{N}$.

2.1 Meta Learning

Meta-Learning is a method of ML where the model is trained on a set of tasks and then tested on a new task. The goal is to learn a model that can learn new tasks quickly'', i.e. 'learning to learn'. These come under the

terminology of few-shot learning where the model is trained on a small number of examples (more specifically, we can say N-way-K-shot learning where N is the number of classes and K is the number of examples per class).

Consider a set of datasets $\mathcal{D} = \{\mathcal{D}_i\}_{i=1}^{N}$ where each dataset $\mathcal{D}_i$ is a set of input-output pairs $\mathcal{D}_i = \{(x_{ij}, y_{ij})\}_{j=1}^{K_i}$. The goal of meta-learning is to learn a model f that can learn a new task $\mathcal{D}_{N+1}$ with a small number of examples. The data used to train the model is called the context set $\mathcal{C}$ which is a set of datasets $\mathcal{C} = \{\mathcal{D}_i\}_{i=1}^{N}$. The data used to query the model is called the target set, which has the support of the inputs only. Our task is to predict the outputs for the target set given the training on the context set.

2.2 Conditonal Neural Processes

Conditional Neural Processes (CNPs) [Gar+18a] was introduced to integrate neural networks and Gaussian Process into a model that can generate a stochastic representation of the training dataset efficiently with low computational complexity.

CNPs are unable to generate coherent sample paths, they are only able to product a stochastic representation of the training dataset.

2.3 (Latent) Neural Processes

Neural Processes or better named 'Latent Neural Processes' are an extension of CNPs that are able to generate coherent sample paths. They are able to do this by using a latent variable z which is created by the encoder. The sampled latent variable is then used by the decoder to generate coherent sample paths.

3 Transformer Neural Processes

3.1 Introduction

Transformer Neural Processes (TNPs) [NG23] were introduced to overcome the limitations of the Latent Neural Process (LNP) [Gar+18b] which have the following limitations:

- LNPs have intracatable likelihoods hence they have to be trained using variational inference by optimizing for the ELBO. It has been shown that the ELBO can be a poor approximation of the true likelihood hence the model can be poorly trained.
- LNPs tend to underfit the data and are unable to capture the complexity of the data.

TNPs overcome these limitations by using the Transformer [Vas+17] to utilize attention to learn the relationships between the context points. This allows the model to capture the complexity of the data and also allows the model to be trained using maximum likelihood estimation (MLE) instead of variational inference as the likelihood is tractable.

Attention has already been implemented before [NG23] in [Kim+19] however according to [NG23] the attention mechanism used in [Kim+19] tends to make overconfident predictions and have poor performance on sequential decision-making problems.

3.2 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.

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{F}}\left[\log p(y_{N_c+1:N}|x_{N_c+1:N}, \mathcal{C}; \theta)\right] \tag{8}$$

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:

$$\mathcal{L}(\theta) = \mathbb{E}_{(x,y)\sim\mathcal{F}}\left[\sum_{i=N_c+1}^{N} \log p(y_i|x_i, \mathcal{C}, x_{N_c+1:i-1}, y_{N_c+1:i-1}; \theta)\right] \tag{9}$$

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.

3.3 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].

4 ConvCNP

4.1 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.

4.2 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.

4.2.1 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:

$$\text{SetConv}(\{x_i, y_i\}_{i=1}^N)(x) = \sum_{i=1}^{N} [1, y_i]^T \mathcal{K}(x - x_i) \tag{10}$$

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.

$$R(x_t) = \text{SetConv}(\text{CNN}(\{\text{SetConv}(\mathcal{C})(x_d)\}_{d=1}^{D}))(x_t) \tag{11}$$

4.3 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:

$$\mu(x_t), \sigma^2(x_t) = \text{MLP}(R(x_t)) \tag{12}$$

References

[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].