Multi-dimensional Positional Self-Attention With Differentiable Span

Meena Alfons, David W. Romero, and Michael Cochez

Vrije Universiteit Amsterdam, Netherlands
meena.kerolos@student.vu.nl,
{d.w.romeroguzman, m.cochez}@vu.nl

Abstract. Attentive models are inherently permutation equivariant and need positional encoding to incorporate positional information, crucial to structured data like time series, images and other inputs with a higher number of positional dimensions. Attentive models are suitable to represent long-distance relations, although they come with a substantial computational burden that local self-attention is designed to solve. This work presents a flexible way to produce positional features that naturally extends to multiple dimensions. We use sinusoidal representation networks (SIREN) to implicitly represent positional encoding. Our approach uses a relative positional encoding that integrates with the attentive model in a way that keeps it translation equivariant. SIREN-based positional encoding gives comparable results to models depending on fixed sinusoidal features. We also introduce a differentiable span, a data-driven approach that limits the attention span according to a locality feature inferred from the data. Using local self-attention with a differentiable span increases the model's accuracy under specific conditions. It also has the potential to reduce the computation costs of attention when the implementation makes use of the learned span to limit the computation of attention scores.

Keywords: Self-attention · Multi-dimensional · Positional Encoding · Local self-attention · Differentiable · Transformer · Classification.

1 Introduction

Attentive models have become ubiquitous building blocks in deep learning architectures and achieved state-of-the-art results in many domains, including Natural Language Processing (NLP) and Computer Vision. One of the main advantages of attention is its ability to model the dependencies between different parts of the input regardless of the distance between them. However, attention models face two main challenges. First, the attention operation treats the input as a set of tokens ignoring the spatial or temporal structure of the input that is crucial for adequately interpreting structured data like time series, images and videos. Here comes the importance of positional encoding, a set of features that reflects the positional structure of the input. Attention models are usually augmented

with positional encoding to treat structured inputs properly. For example, the original Transformer[21] presents a kind of positional encoding that depends on fixed sinusoidal features with different frequencies that produce different representations for each position in the input. That positional encoding is then added to the input to insert the positional information in the representation of the input tokens. That positional encoding is designed to encode the absolute position of tokens in a 1-dimensional sequence. Since then, positional encoding has seen further development in multiple directions to enhance its modelling capacity and make it more parameter efficient.

In this work, we present a way to let the network learn the positional encoding of relative positions from the data by implicitly modelling it with sinusoidal representation networks (SIREN). The relative nature of this positional encoding allows it to be integrated with the attention operation providing new positional information at each attention layer and keeping the attention operation translation equivariant. Furthermore, the presented approach can produce positional encoding for inputs with multiple positional dimensions, capturing the actual multi-dimensional structure of the input. We show that SIREN-based positional encoding produces comparable results to models depending on fixed sinusoidal features for the 1D setting. We also show that it can capture the multi-dimensional structure of the input.

The second challenge is the substantial computation costs of the attention operation. For self-attention to model long-distance relations, it computes the relations between all pairs of input tokens. The complexity of this operation is quadratic in the size of the input. One way to reduce the resources needed for the attention operation is to limit a token's attention span to the local neighbourhood around that token, called local self-attention. Such models depend on choosing the attention span beforehand as a hyperparameter. Finding the optimal values for the attention span for each attention layer presents a new challenge in the design space.

This work presents a differentiable parameterisation for the attention span that captures the locality feature for inputs with multiple positional dimensions. This approach allows the attention span to be inferred from the data and adapts to the information density at each attention layer. This approach removes the demanding task of hyperparameter tuning for the attention span of local self-attention models. We show that using a differentiable span with a number of positional dimensions matching the input's structure increases the model's accuracy by focusing on the most relevant tokens of the input and discarding the noise coming from tokens outside the local neighbourhood. Furthermore, the learned span can be used to reduce the computation costs of the attention operation by limiting the computed attention scores to the local neighbourhood instead of discarding the unneeded attention scores after being computed. We present two implementations that attempt to limit the attention operation using the learned span. However, both are less efficient than computing all attention scores and discarding those outside the learned attention span.

We are interested in the following research questions:

- How does SIREN-based positional encoding affect the accuracy of a self-attention-based model compared to a baseline positional encoding using fixed sinusoidal features followed by a linear layer?
- How does using a differentiable span affect the accuracy of a self-attention-based model?

The following sections are organised as follows. Section 2 covers the related work regarding self-attention, positional encoding, and local self-attention. Section 3 presents the algorithms and parameterisation of the modules we introduce in addition to the overall network architecture used in our experiments. Section 4 covers the experimental design and the implementation details of the experiments. Section 5 presents the results of the conducted experiments with their interpretations. Section 6 includes a discussion reflecting on the outcome and methodology used in this research. Finally, section 7 concludes this work with summarised answers to the research questions and future work.

2 Related Work

Self-Attention. The Transformer[21] is the first model that depends entirely on self-attention. Self-attention-based models, including the Transformer and its variants [3,15,11], have achieved the state of the art in many domains, including natural language processing (NLP), computer vision, and others [2][6].

Positional Encoding. Positional encoding (PE) is essential for self-attention models. Without positional encoding, self-attention becomes permutation equivariant and ignores the input's structure, which is crucial for many kinds of inputs like images. Positional encoding can be broken down into the following set of factors which have seen various developments: absolute or relative, where PE is merged with the attentional model (location), what operation is used for merging (operation), what part of the data it is merged with (augmented with), the representation of PE, having learnable parameters or not, being discrete or continuous, and being shared across heads or not. Table 1 compares positional encoding approaches incorporating various combinations of those factors.

Absolute positional encoding[5,21,13] produces a representation for each position in the input. This absolute representation then gets merged with the input via a point-wise addition operation. As a result, the absolute positional encoding will affect the self-attention layer's queries, keys and values. Relative positional encoding[19,8,16,3,22] considers the relative position of each key to each query and produces a representation for each unique relative position. For example, if the input is a sequence of N tokens, relative positions are in the range $[-(N-1), N-1]$. Another way to produce relative positional encoding is to subtract the absolute positional encoding for two positions[22]. Relative positional encoding is applied inside each self-attention layer. This makes the model translation equivariant and provides new positional information at each self-attention layer. Relative positional encoding can be merged with keys[8,16,3], with keys and values[19], or with attention scores[22]. The merge operation could be point-wise addition or concatenation[22].

Table 1: **Comparison: Positional Encoding.** Displayed in chronological order. a=absolute, r=relative, d=discrete, c=continuous, outside=once before the first attention layer, inside=inside each attention layer, N=sequence length or number of unique positions.

Work	a/r	location	merge operation	merge with	representation	learnable params	d/c	shared across heads	content & position bias
[5]	a	outside	add	inputs	embedding vectors	O(N)	d	no	no
[21]	a	outside	add	inputs	fixed sin features	no	c	no	no
[19]	r	inside	add	keys & values	embedding vectors	O(N)	d	yes	no
[8]	r	inside	add	keys	embedding vectors	O(N)	d	no	no
[16]	r	inside	add	keys	fixed sin features	no	c	-	no
[3]	r	inside	add	keys	fixed sin features + linear layer	O(1)	c	-	yes
[22]	r	inside	concat	attention scores	normalized position + linear layer	O(1)	c	-	no
[13]	a	outside	add	inputs	learned sin features + 1-layer GeLU	O(1)	c	-	no
Baseline	r	inside	add	keys	fixed sin features + linear layer	O(1)	c	no	yes
Ours	r	inside	add	keys	normalized position + SIREN	O(1)	c	no	yes

The representation for either absolute or relative positional encoding is a function that takes the position and returns a vector representing this position. This representation can be discrete [5,19,8] by learning the representation vectors for the input positions seen during training. However, this approach cannot produce representations for positions not seen during training. This kind of representation requires $O(N)$ learnable parameters where N is the number of unique positions of the input. Another representation approach is to have a transformation applied to the position to generate the representation vector. Such transformation can be parameter-free or incorporate learnable parameters. Positional encodings that depend on such transformations are called continuous because they can produce representations for positions outside the range seen during training. If this transformation has learnable parameters, they need $O(1)$ parameters that do not depend on the input size. The Transformer [21] uses a parameter-free transformation function that uses sinusoidal features with different frequencies to construct different representations for different positions. Transformer-XL [3] adds a linear layer with learnable parameters after those fixed sinusoidal features. A learnable version of the sinusoidal features where the frequencies are not fixed is introduced in [13], which also augments the transformation function with a 2-layer feed-forward network with A GeLU activation for the hidden layer. Another model that does not depend on sinusoidal features is [22], where the normalized position is fed into a linear layer that provides the position representation. In most models, the representation vector size is equal to

Table 2: **Comparison: Multi-dimensional positional encoding.**

Work	factorized	merge operation
[16]	yes	concat
[7]	yes	add
[22]	yes	concat
[13]	no	-
Ours	no	

the content vector size of the inputs or the keys, which gives each attention head different positional information. Sharing the positional encoding across attention heads is possible, as shown in [19].

Transformer-xl introduces a new parameterisation to incorporate relative positional encoding inside the self-attention layer. This parameterisation provides clear differentiation among content attention, positional attention, content bias, and positional bias. We will be building on this parameterisation for the baseline and our new approach.

Multi-dimensional positional encoding. Inputs with multiple positional dimensions are natural in cases like images with two spatial dimensions or videos with an additional temporal dimension. One way to extend positional encoding to multi-dimensional spaces is to produce positional encoding for each dimension separately and then merge them to get a representation for each multi-dimensional position. The merge operation could be point-wise addition [7] or concatenation [16,22]. Another way to produce positional encoding for a multi-dimensional space is to construct a function that takes the multi-dimensional coordinate of a position as input and gives a representation vector as an output [13]. The latter can capture more complex positional relations that cannot be factorized into relations in orthogonal dimensions. Table 2 compares different approaches to produce multi-dimensional positional encoding.

Sinusoidal Representation Networks. Implicit neural representation can be used to implement the transformation function needed to generate positional encoding vectors from position coordinates. Using periodic activation functions in implicit neural representations proves to be superior in representing spatial and temporal signals with fine details [20]. Such networks with periodic activation functions are called sinusoidal representation networks or SIRENs. SIRENs have been successfully used to model continuous convolution kernels for sequential data of arbitrary length [18].

Local Self-Attention The ability of self-attention to model long-distance relations in one layer comes with a considerable computational burden where each token needs to attend to each other token in the input. This results in a quadratic complexity in FLOPs needed to carry out self-attention. Local self-attention has been mentioned as future work in the Transformer[21], where each query con-

siders a limited neighbourhood of keys centred around the respective query. Another way to achieve local attention is to chunk the input sequence into non-overlapping blocks of queries. Each query only attends to keys inside its block[14]. This approach can be extended to allow each query block to have a wider keys block[15], which results in overlapping key blocks. In both cases, the attention cost per block becomes constant. Consequently, the cost of the attention layer becomes linear in the size of the input. While local attention limits the span of attention per layer, a multi-layer self-attention network can still model long-distance relations.

Differentiable span. In this work, we introduce a differentiable attention span, where the local neighbourhood size is parameterised and adapted to the data during training. This idea has been implemented by Flexconv[17] in convolutional neural networks, where the kernel's size is learned during training.

3 Architecture

This section presents a formal introduction to the ideas incorporated in our approach, including multi-head attention, relative positional self-attention, the baseline positional encoding, the SIREN-based positional encoding, multi-dimensional positional encoding, and differentiable span. The overall network architecture is also presented for the classification tasks with which we experiment.

3.1 Transformer-based Classifier

The Transformer is the first model that solely depends on self-attention. It consists of an encoder and a decoder stack to be used for tasks like machine translation. A multi-head attention layer is used in both the encoder and decoder blocks. However, classification tasks like those we focus on in this work do not need the decoder part of the architecture. Figure 1 shows the overall architecture of a transformer-based classifier where a stack of encoder blocks is used. The representation of the last token at the last layer is used for classification. It is fed into a feed-forward neural network that gives the probabilities for the output classes.

3.2 Multi-head Self-attention

The multi-head attention layer is a crucial part of this architecture. Inside encoder blocks, the multi-head attention layer gets its queries, keys and values from the same pool of tokens. Hence, the operation is called self-attention. We collect the entire operation in a layer called Multi-Head Self-Attention. Algorithm 1 defines the inputs, outputs, parameters and hyperparameters of the Multi-Head Self-Attention layer alongside how the self-attention operation is carried out. Notice that it internally uses the $MultiHeadAttention$ operation defined in Algorithm 2.

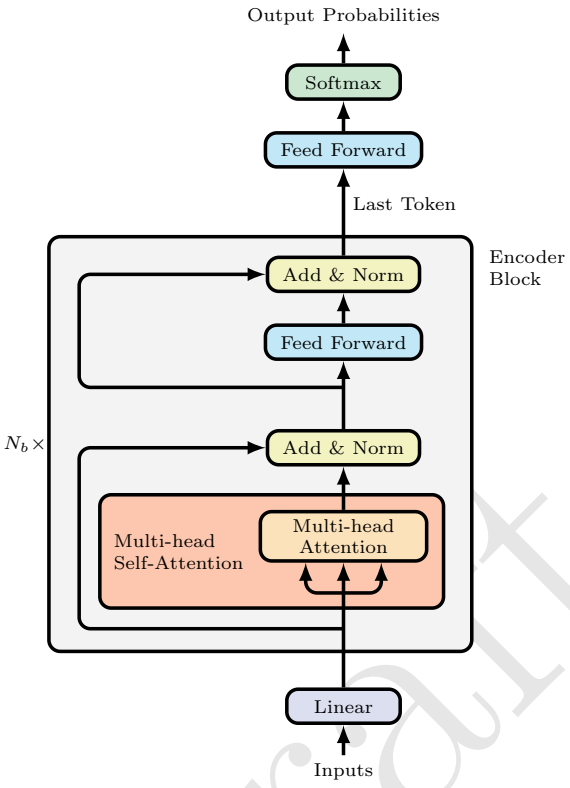


Fig. 1: Transformer-based classifier

3.3 Relative Positional Self-Attention

Transformer-XL[3] proposed a different parameterisation for computing the attention scores that incorporate relative positional encoding inside the self-attention layer. Formula 1 shows an adapted version of that parameterisaiton to fit the purposes of this work. That formula uses the Einstein notation[1], where subscripts refer to the tensor dimensions. Scaling and softmax operations are omitted to focus on the four terms contributing to the attention scores.

Algorithm 1: Multi-head Self-Attention

input : $X \in \mathbb{R}^{N \times d}$ where N is the number of input tokens.
output : $Y \in \mathbb{R}^{N \times d}$ where N is the number of output tokens (same as input tokens).
parameters : $W_q, W_k, W_v, W_o \in \mathbb{R}^{d \times d}$
hyperparameters: d is the number of features for each token.
$Q, K, V \leftarrow XW_q, XW_k, XW_v \qquad Q, K, V \in \mathbb{R}^{N \times d}$
$O \leftarrow MultiHeadAttention(Q, K, V) \qquad O \in \mathbb{R}^{N \times d}$
$Y \leftarrow OW_o$

$$A_{qk} = \underbrace{(Q_{qd}K_{kd})_{qk}}_{\text{content attention}} + \underbrace{(Q_{qd}P_{qkd})_{qk}}_{\text{positional attention}} + \underbrace{(u_d K_{kd})_k}_{\text{content bias}} + \underbrace{(v_d P_{qkd})_{qk}}_{\text{positional bias}} {}^{*} \tag{1}$$

where q is the query dimension ($= N$)
k is the key dimension ($= N$)
d is the feature dimension
u_d, v_d are learnable parameters
P_{qkd} is the relative positional encoding for all keys respective to all queries
* Einstein notation

The attention scores in Formula 1 consist of the following terms:

- **content attention** is a score representing the similarity between a query and a key in the feature space.
- **positional attention** is a score representing the similarity between the query and the relative positional encoding of a key respective to that query.
- **content bias** is a score that depends solely on the content of the key, providing bias according to the content of the key.
- **positional bias** is a score that depends solely on the relative positional encoding of the key respective to the query.

In these terms, the similarity is measured by a dot-product in the feature space represented by the d dimension.

3.4 Positional Encoding

The tensor P is the only piece in Formula 1 that is yet to be defined. Before generating relative positional encoding and constructing the tensor P, let us introduce some definitions.

N is the number of unique positions in the input. For inputs with one positional dimension, N is the sequence length. This will be extended later to accommodate multiple dimensions.

Algorithm 2: MultiHeadAttention

input : $Q, K, V \in \mathbb{R}^{N \times d}$ where N is the number of input tokens.
output : $O \in \mathbb{R}^{N \times d}$
hyperparameters: d is the number of features for each token. H is the number of heads.

Q, K, and V are split into H heads where the number of features d is split into $H * d_H$, and d_H is the number of features in one head.

$d_H \leftarrow d/H$

The following operations are broadcasted on the heads which means these operations are effectively applied separately to each head. We'll use Q_h, K_h, V_h, O_h to refer to one head h extracted from the respective tensors Q, K, V, Y where $Q_h, K_h, V_h, O_h \in \mathbb{R}^{N \times d_H}$.

$Q_h \leftarrow Q_{:,\, hd_H:(h+1)d_H}$
$K_h \leftarrow K_{:,\, hd_H:(h+1)d_H}$
$V_h \leftarrow V_{:,\, hd_H:(h+1)d_H}$

$A_h \leftarrow Q_h {K_h}^T / \sqrt{d_H} \quad A_h \in \mathbb{R}^{N \times N}$

$S_h \leftarrow softmax(A_h)$
where softmax is applied to each row to normalize the scores of keys respective to a specific query.

$O_h \leftarrow S_h V_h \quad O_h \in \mathbb{R}^{N \times d_H}$

N_r The number of relative positions. For inputs with one positional dimension, the relative position of keys respective to queries can take values from $-(N-1)$ to $(N-1)$. Therefore, the total number of unique relative positions $N_r = 2N - 1$. Notice that relative positions are, in fact, the relative distances between keys and queries. The definition of relative positions will be extended for multiple positional dimensions in section 3.5

Generating the tensor P holding relative positional encoding for all keys respective to all queries can be split into two steps, as shown in Figure 2.

Step 1. The first step is to generate positional encoding vectors for all unique relative positions. This is represented by $E \in \mathbb{R}^{N_r \times d}$, where N_r is the number of relative positions and d is the number of features in each positional encoding vector. Various approaches can be used to generate those positional encodings, as listed in Table 1. This section will explain the baseline and our new approach, SIREN-based positional encoding.

Baseline positional encoding. Algorithm 3 is the baseline we use to generate the positional encoding for relative positions E. It is similar to the one used in Transformer-XL. First, a matrix of sinusoidal features S is generated for all

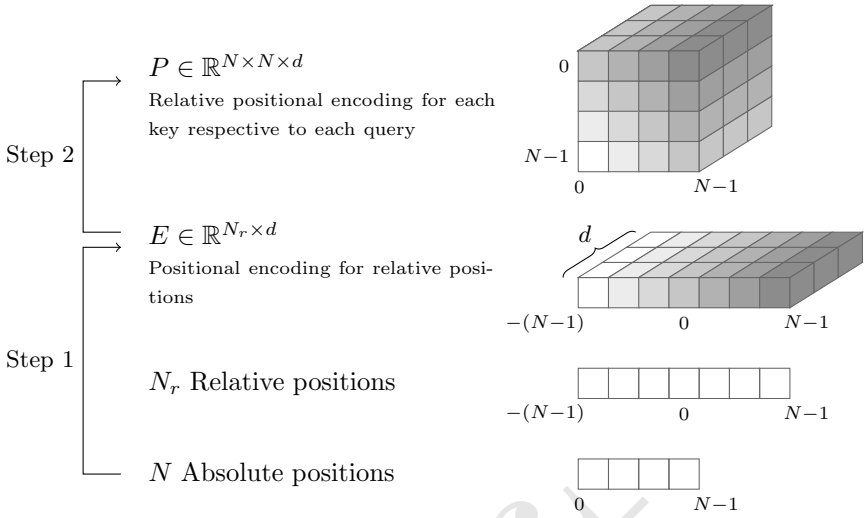


Fig. 2: **Generating Relative Positional Encoding.** N is the number of absolute positions, N_r is the number of relative positions, E holds the positional encoding vectors for all relative positions, and P is the relative positional encoding for all keys respective to all queries.

relative positions. That matrix S is then fed into a linear layer with weights W and b (Transformer-XL does not have the bias b).

SIREN-based positional encoding. SIRENs are proven superior in representing spatial and temporal signals with fine details [20]. Therefore, they are a good fit for generating positional encoding vectors for each relative position. Algorithm 4 shows the implementation of a SIREN network with L layers that generates such positional encodings. First, the coordinates of relative positions are normalized in the range $[-1, 1]$. Then, the value of the normalized coordinate is fed into this feedforward network to produce the positional encoding for that coordinate. All hidden layers use the sine activation function, and the last layer stays without activation. There are two critical hyperparameters for this network, ω_0 and $\omega_{0,initial}$ serve as the base frequencies for the sine activation function. The value of $\omega_{0,initial}$ proves necessary as it interacts with the frequency spectrum of the input signal. Furthermore, the value of ω_0 helps boost the gradients to the weight matrix W by the factor ω_0. We found that using $\omega_{0,initial}$=ω_0=10 works best for the task and dataset we experiment with.

Step 2. The second step in Figure 2 is to spread the values in E into $P \in R^{N \times N \times d}$, where the first index refers to the query and the second index refers to the key. P_{ij} is the positional encoding vector for key j relative to query i. Therefore, $P_{ij} = E_{j-i}$ for all $i, j \in [0, N)$. For an efficient algorithm to construct P from E, see Appendix A.1.

Algorithm 3: Relative Positional Encoding - Baseline

input : N the number of input positions.
output : $E \in \mathbb{R}^{N_r \times d}$
parameters: $W \in \mathbb{R}^{d \times d}, b \in \mathbb{R}^d$
$N_r \leftarrow 2N - 1$
Create meshgrid $x \in \mathbb{N}^{N_r \times 1}$ of integer positions
$x_i \leftarrow i$ for all $i \in [0, N_r)$
Create a matrix of sinusoidal features $S \in \mathbb{R}^{N_r \times d}$

$$S_{i,j} = \begin{cases} sin(x_i e^{-j * log(10000)/d}), & \text{if } j \text{ is even} \\ cos(x_i e^{-(j-1) * log(10000)/d}), & \text{if } j \text{ is odd} \end{cases}$$

for $i \in [0, N_r)$ and $j \in [0, d)$
$E \leftarrow SW^T + b^T$

Algorithm 4: Relative Positional Encoding - SIREN - 1D

input : N the number of input positions.
output : $E \in \mathbb{R}^{N_r \times d}$
parameters : $W_1 \in \mathbb{R}^{d \times 1}, b_1 \in \mathbb{R}^d$, and $W_l \in \mathbb{R}^{d \times d}, b_l \in \mathbb{R}^d$ for $l \in [2, L]$
hyperparameters: $\omega_0, \omega_{0,initial} \in \mathbb{R}$, L number of layers
$N_r \leftarrow 2N - 1$
Create meshgrid $x \in \mathbb{R}^{N_r \times 1}$ of normalized positions in $[-1, 1]$
$x_i \leftarrow \frac{i-(N-1)}{(N-1)}$ for all $i \in [0, N_r)$

$z_1 = x * W_1^T + b_1^T$
$h_1 = sin(w_{0,initial} z_1)$
For $l \in [2, L-1]$ do:
    $z_l = h_{l-1} * W_l^T + b_l^T$
    $h_l = sin(w_0 z_l)$
$E \leftarrow h_{L-1} * W_L^T + b_L^T$

3.5 Multi-dimensional Positional Encoding

Positional dimensions are ones along which the input tokens are laid out in an ordered structure that is meaningful for the input domain. Spatial and temporal dimensions are the most famous positional dimensions. Inputs with multiple spatial and temporal dimensions are very common. Examples include images with two spatial dimensions, videos with two spatial dimensions and one temporal dimension (a total of 3 positional dimensions), and other tasks with more dimensions [13].

Such structured inputs are treated as an unordered set of tokens for self-attention because the attention operation is permutation equivariant and does not care about the positions of tokens. However, such models' accuracy suffers when the input tokens' positional structure is significant to interpret the input and solve the task at hand correctly. Therefore, adding some positional hint to the attention operation is crucial for such tasks. That positional hint

is represented by the positional encoding, which needs to consider the input's multi-dimensional structure.

StructureSize. In this context, we define $StructureSize \in \mathbb{N}^n$, a vector holding the size of the input along each positional dimension where n is the number of positional dimensions. That means $StructureSize_0$ is the number of absolute positions along the first positional dimension attributed to input tokens. For an image of size 32×32, $StructureSize$=$[32\ 32]^T$. Notice that the number of input tokens N=$prod(StructureSize)$, which is 1024 in this case.

RelativeSize. Similarly, $RelativeSize \in \mathbb{N}^n$ holds the number of unique relative positions along each dimension, which is calculated by $RelativeSize$= $2StructureSize-1$. For an image with $StructureSize$=$[32\ 32]^T$, $RelativeSize$= $[63\ 63]^T$. There are 63 relative positions in the first dimension in the range $[-31, 31]$. The total number of relative positions N_r= $prod(RelativeSize)$= 3969, which are spread in a two-dimensional grid ranging from $(-31, -31)$ to $(31, 31)$.

Extended SIREN-based positional encoding. SIREN-based positional encoding can easily be extended to consider multi-dimensional positions according to the $StructureSize$. In the 1D version, the SIREN network takes one value $x_i \in \mathbb{R}$ as an input representing the coordinate of the position i. In the multi-dimensional version, the SIREN network takes the multi-valued coordinate $x_i \in \mathbb{R}^n$ representing the position i in n-dimensional space. The result $E \in \mathbb{R}^{RelativeSize \times d}$ of the extended algorithm is the positional encoding for each relative position is a multi-dimensional grid of size $RelativeSize$. Here we took the liberty of using $\mathbb{R}^{RelativeSize \times d}$ to refer to $\mathbb{R}^{RelativeSize_0 \times ... \times RelativeSize_{n-1} \times d}$. Algorithm 5 shows the extended algorithm.

Algorithm 5: Relative Positional Encoding - SIREN

input : $StructureSize \in \mathbb{N}^n$ the number of input positions along each dimension where n is the number of positional dimensions of the input.
output : $E \in \mathbb{R}^{RelativeSize \times d}$
parameters : $W_1 \in \mathbb{R}^{d \times n}, b_1 \in \mathbb{R}^d$, and $W_l \in \mathbb{R}^{d \times d}, b_l \in \mathbb{R}^d$ for $l \in [2, L]$
hyperparameters: $\omega_0, \omega_{0,initial} \in \mathbb{R}$, L number of layers.

$RelativeSize \leftarrow 2StructureSize - 1$
Create meshgrid $x \in \mathbb{R}^{RelativeSize \times n}$ of normalized positions in $[-1, 1]$
$x_{...,i_j,...,j} \leftarrow \frac{i_j - (StructureSize_j - 1)}{(StructureSize_j - 1)}$ for all $i_j \in [0, RelativeSize_j)$ and $j \in [0, n)$
Notice that i_j indexes the jth dimension of x

$z_1 = x * W_1^T + b_1^T$
$h_1 = sin(w_{0,initial} z_1)$
For $l \in [2, L-1]$ do:
    $z_l = h_{l-1} * W_l^T + b_l^T$
    $h_l = sin(w_0 z_l)$
$E \leftarrow h_{L-1} * W_L^T + b_L^T$

Formula 1 is still valid for inputs with multiple positional dimensions. Each token can be referred to with a unique id in the range $[0, N)$. These unique ids are the indices of the q and k dimensions. P still holds the relative positional encoding for each key respective to each query extracted from E. The construction of P from $E \in \mathbb{R}^{RelativeSize \times d}$ is shown in Appendix A.1.

3.6 Differentiable Span

The self-attention operation has a global span by default. This means that each query attends to all keys. This allows the model to capture long-distance relationships in one layer. However, it comes with a substantial computational burden that's quadratic in the number of input tokens $O(N^2)$. Local self-attention[14,15] has been introduced to reduce this computational burden to $O(N)$ by limiting the attention operation for each query to a subset of the keys contained in a limited neighbourhood around the query. Local self-attention limits the distance of captured relations in one layer. However, long-distance relations can still be captured in later layers for which the effective receptive field is wider than the defined local neighbourhood for the local attention.

The size of the local neighbourhood is a crucial hyperparameter for those models. It needs to be big enough to include tokens that significantly affect a query. However, the span of this significant neighbourhood is not known a priori, and it may differ across layers. This makes choosing the size of the local neighbourhood more challenging. Therefore, we propose using a differentiable span whose size can be learned from the data during training. This allows the attention span to adapt to the data and the information density in each layer and eventually include the significant tokens in the attention operation.

Gaussian function. In order to implement this differentiable span, we use a Gaussian function with parameterised variance σ^2 in combination with a constant threshold. The Gaussian function is appropriate to model the locality feature because it gives high values for positions around the centre and low values for positions further away. Relatively applying a Gaussian function, centred around each query, models the locality around each query. The attention scores for keys relative to a specific query are multiplied by the values of the Gaussian function centred around this query. This multiplication modulates the attention scores according to the learned locality feature. The constant threshold is used as a cutoff value on the Gaussian function to determine the attention span around a query and limit the attention operation to the keys inside this span.

A multi-dimensional Gaussian function is used to capture the locality in the input's positional space, which can have multiple positional dimensions. The mean of the Gaussian function is always 0, and it is always centred around each query. We need to parameterise the full covariance matrix for a general parameterisation of the variance of a multi-dimensional Gaussian function. However, parameterising the covariance matrix is a bit complex while maintaining its constraint to be positive semi-definite. Instead, we implement a stricter version where the covariance matrix is diagonal (independent dimensions). This means

there is one standard deviation parameter σ_i for each dimension $i \in [0, n)$. Each attention layer has a set of parameters $\sigma \in R^n$ whose size is equal to the number of positional dimensions n.

An un-normalised Gaussian function is used to keep the curve's height at 1. This way, changing the scale of the Gaussian function does not change the curve's height. This is important to use the Gaussian function to modulate the attention scores relative to the query at the centre without attenuating all the scores.

Algorithm 6: Calculate *SpanSize*

input : $StructureSize \in \mathbb{N}^n$ where n is the number of positional dimensions of the input.
output : $SpanSize \in \mathbb{N}^n$
parameters : $\sigma \in \mathbb{R}^n$
hyperparameters: $t \in \mathbb{R}$ the threshold.
$x = \sqrt{-2log(t) * \sigma^2}$
where $x \in \mathbb{R}^n$ is the position (independently for each dimension) at which the Gaussian function equal the threshold t

The meshgrid used to generate the Gaussian function uses the range $[-1, 1]$ for the positional range $[-(StructureSize_i - 1), StructureSize_i - 1]$ for each dimension $i \in [0, n)$. Therefore, the *SpanSize* is calculated as follows:

$SpanSize = 2\lceil x \circ \frac{StructureSize-1}{1} \rceil + 1$
Notice that operator $\circ$ is the Hadamard product.

Limit the *SpanSize* to the *RelativeSize*:
$SpanSize_i = min(SpanSize_i, RelativeSize_i)$ for all $i \in [0, n)$

SpanSize. The $SpanSize \in \mathbb{N}^n$ is a vector holding the size of the learned span along each positional dimension, where n is the number of positional dimensions. The *SpanSize* operates in and limits the space of relative positions defined by *RelativeSize*. The *SpanSize* is calculated using the learned values for σ and the constant threshold t. The changing value of σ moves the cutoff position where the value of the Gaussian function equals the threshold t. These cutoff positions on each dimension define the *SpanSize*. Algorithm 6 shows the calculation of the *SpanSize*. This *SpanSize* is used to limit the calculation of the Gaussian function instead of calculating it for the full *RelativeSize*. It also limits the generation of positional encoding because values outside *SpanSize* are not needed. Algorithm 7 shows the multi-dimensional Gaussian function where the Gaussian function is calculated for relative positions in a grid of size *SpanSize*.

Modulating attention scores. The generated Gaussian values $G \in \mathbb{R}^{SpanSize}$ need to be transformed into $G^* \in R^{N \times N}$, where each key has the Gaussian value of its relative position to the respective query. This operation is similar

Algorithm 7: GaussianFunction

input : $StructureSize, SpanSize \in \mathbb{N}^n$ where n is the number of positional dimensions of the input.
output : $G \in \mathbb{R}^{SpanSize}$
parameters: $\sigma \in \mathbb{R}^n$
Create meshgrid $x \in \mathbb{R}^{SpanSize \times n}$ of normalized positions.
$x_{\ldots,i_j,\ldots,j} \leftarrow \frac{i_j - (StructureSize_j - 1)}{(StructureSize_j - 1)}$ for all $i_j \in [0, SpanSize_j)$ and $j \in [0, n)$
Notice that i_j indexes the jth dimension of x

$G \leftarrow e^{-0.5x \circ \sigma^{-2}}$
Notice that operator $\circ$ is the Hadamard product.

to transforming E into P, as shown in Appendix A.1. G^* is then multiplied element-wise with A to modulate the attention scores, as shown in Algorithm 8. In addition to modulating the attention scores inside the $SpanSize$, all attention scores outside the $SpanSize$ are set to $-\infty$ to remove their effect altogether. Finally, a boolean mask $SpanMask \in \mathbb{B}^{N \times N}$ asks to be generated from the $SpanSize$ to indicate which keys are inside the $SpanSize$ respective to each query. The generation of $SpanMask$ from the $SpanSize$ is shown in Appendix A.1.

Algorithm 8: Modulate Attention scores with Differentiable Span

input : $A \in \mathbb{R}^{N \times N}$ the attention scores. $G^* \in \mathbb{R}^{N \times N}$ relative Gaussian values for each key respective to each query. $SpanMask \in \mathbb{B}^{N \times N}$ boolean mask generated from $SpanSize$.
output: $A^* \in \mathbb{R}^{N \times N}$ the adjusted attention scores.
Modulate attention scores:
$A^*_{ij} \leftarrow A_{ij} \circ G^*$

Remove attention scores outside $SpanSize$:
$$A^*_{ij} = \begin{cases} -\infty & \text{if } SpanMask_{ij} \text{ is } False \\ A^*_{ij} & \text{if } SpanMask_{ij} \text{ is } True \end{cases}$$

Improving attention complexity using differentiable span. The initial goal for local attention is to reduce the complexity of the attention operation from $O(N^2)$ to $O(N)$. We have shown how to learn the attention span and apply it to attention scores to effectively achieve the local attention operation. We will call this Version 1, whose implementation is shown in Appendix A.1. In order to achieve any computation gains, the $SpanSize$ needs to be used to limit the attention scores computed for each query to only include the attention scores for the keys inside the $SpanSize$ around the query. We present two attempts to use

the *SpanSize* to limit the attention operation as Version 2 and 3, whose implementations can be found in Appendix A.2 and Appendix A.3, respectively. Both versions have the same parameters as Version 1 and produce the same output and the same gradients on those parameters. However, both implementations turn out to be less efficient than Version 1 by an order of magnitude for reasons explained in the comparison provided in Appendix A.4. Further work is needed to develop an implementation that efficiently applies local attention using the learned span.

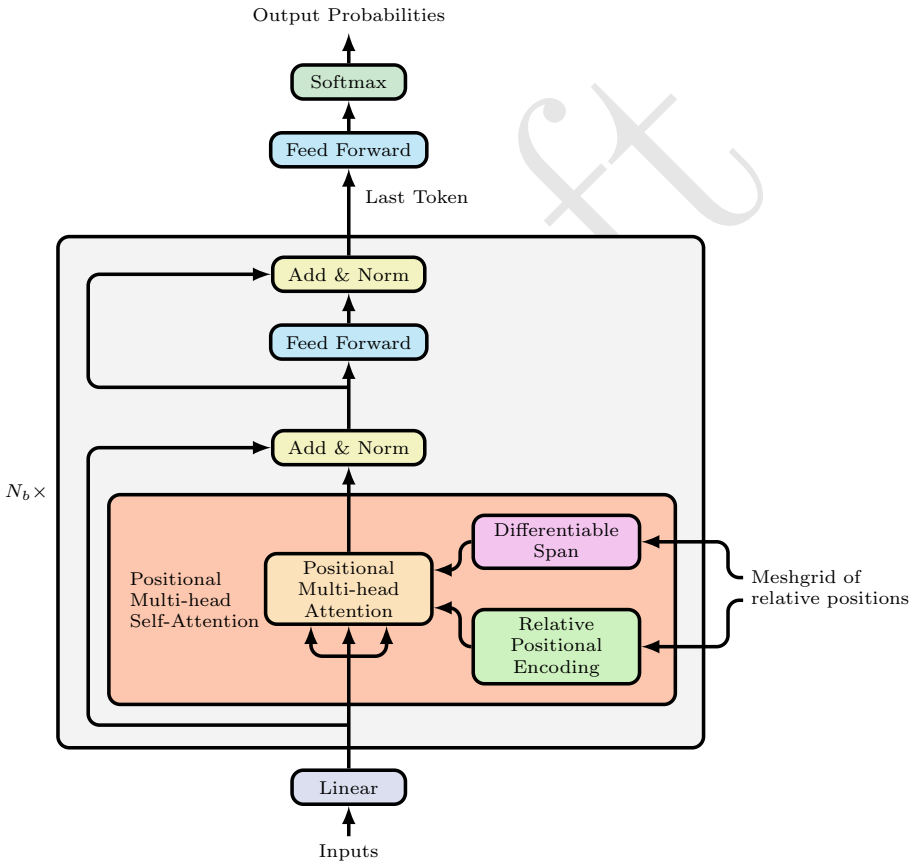


Fig. 3: Transformer-based classifier with Positional Multi-head Self-Attention which includes Relative Positional Encoding and Differentiable Span.

3.7 Positional Multi-Head Self-Attention Block

Figure 3 puts the presented modules into perspective. The Differentiable Span module takes a meshgrid of normalized relative positions and produces the relative Gaussian values. The Relative Positional Encoding module takes a meshgrid of normalized relative positions and produces the relative positional encoding. The results of both modules are fed into the Positional Multi-Head Attention module, which computes the attention scores, including the relative positional encoding as shown in Formula 1 and modulates the attention scores using the Gaussian values. Finally, the whole operation is encapsulated in a new block called Positional Multi-Head Self-Attention.

4 Experiments

4.1 Task & Datasets

In this work, we focus on the image classification task to compare the validation accuracy produced by different models. The MNIST dataset[4] was found to be too easy as an image classification task (even in its sequential form) to compare the models we propose and differentiate between them. Instead, the CIFAR10 dataset[12] is used in an image classification setting to test the different models in this work and the hypotheses around them. CIRAF10 has ten classes with 6K images per class, meaning a random choice algorithm would give an accuracy of 10%. The dataset provides 50K training images and 10K test images. The test images are not used in this work at all. Instead, an 80/20 split is used to randomly split the 50K training images into a *train* dataset of 40K images and a *validation* dataset of 10K images.

4.2 Experimental Setup

The training process of deep learning models depends on pseudo randomness for many aspects, including parameter initialization, dataset shuffling, and the internal operation of some modules (e.g. dropout). A *RandomState* variable is used to control the pseudo randomness in the training process, supporting reproducibility and serving as a block factor in the experiment. A set of 5 *RandomState*s are randomly chosen to serve as block factors in a repeated measure experiment design. Each treatment factor is tested multiple times, once for each block factor. This allows us to capture the variance in the effect of each treatment factor while eliminating any variation caused by the block factor. This results in paired samples, allowing the use of paired tests with more power than their unpaired counterparts. The hypotheses being tested are explained below.

SIREN-based Positional Encoding vs Baseline. In this experiment, we test the difference in validation accuracy between two architectures, one using the baseline positional encoding and the other using SIREN-based positional encoding. Both are being tested on an image classification task where the image is treated as a 1D sequence of pixels where each pixel has three colour channels. No local attention is applied to these experiments.

- H0 is that both approaches have the same modelling capacity, and there is no difference in the resulting validation accuracy.
- H1 is that SIREN-based positional encoding achieves **higher** validation accuracy.

Differentiable Span without threshold. In this experiment we test the effect of multiplying the attention scores with the values of a relative Gaussian function centred around each query. This Gaussian function has the ability to model local attention without applying any threshold because it already modulates the attention scores giving higher values for keys around the query and lower values further away. Two treatment factors are tested, one where the attention scores are not modulated and the other where attention scores are modulated using the values of the relative Gaussian function. In both cases, the SIREN-based positional encoding is being used. This allows us to test the effect of the Gaussian function in 1D and 2D settings because the SIREN-based positional encoding can work with multiple positional dimensions.

- H0 is that modulating the attention scores with the values of a relative Gaussian function does not affect the validation accuracy.
- H1 is that modulating the attention scores with the values of a relative Gaussian function **decreases** the validation accuracy.

Differentiable Span with threshold. In this experiment, we test the effect of applying a threshold to limit the attention span in addition to multiplying the attention scores with the values of a relative Gaussian function. The threshold interacts with the Gaussian function's variance to produce a learned span. All attention scores outside this span are discarded by setting them to $-\infty$. Two treatment factors are tested, one where the threshold is set to 0 (no threshold) and the other where the threshold is set to 0.1. In both cases, the SIREN-based positional encoding is being used. This allows us to test the effect of the differentiable span with threshold in 1D and 2D settings. We hypothesise that limiting the attention scores to the $SpanSize$ will allow the positional encoding to focus more on modelling the relative positions inside the $SpanSize$ and remove noise from tokens outside the $SpanSize$.

- H0 is that discarding the modulated attention scores outside a learned span defined by a threshold is as good as using the modulated attention scores for all the inputs regarding the resulting validation accuracy.
- H1 is that discarding the modulated attention scores outside a learned span defined by a threshold gives **higher** validation accuracy than using the modulated attention scores for all the inputs.

Notice that the computation costs of the model with a differentiable span are not being tested in these experiments because Version 1 of the implementation is being used, which does not make any attempt to use the $SpanSize$ to reduce the costs of the attention operation. Version 2 and 3, which attempt to do so, are less efficient than Version 1, as explained in Appendix A.4.

Statistical testing. A linear model is fitted to the results of each of those experiments. The normality of the residual errors of those models is tested by the Jarque–Bera test [9] and found not to be significantly different from a normal distribution. Given the normality assumption, repeated measures ANOVA can be used to test the effect of the treatment factor on the response variable. A power analysis is conducted on the observed results and found that the repeated measures ANOVA has a power of 66.3% to detect the observed differences at a significance level of 0.05 using 5 points for each sample. This is lower than the usual power of 80% at a significance level of 0.05. Hence, the tests may not give a significant result even if a significant effect is present. Therefore, we are using boxplots to visually present the differences between the models in addition to reporting the p-values for the significance of the treatment effect in a repeated-measures ANOVA.

4.3 Implementation

A cross-entropy loss function is used to provide an error signal that boosts the probability of the correct class for an image. An Adam optimizer[10] is used with parameters β_1=0.9, β_2=0.999 and a learning rate of 0.001. The *train* dataset is shuffled before each training epoch. Each model is trained for 80 epochs to ensure that it has enough training steps to achieve its highest validation accuracy. All experiments are done using one Nvidia Titan RTX GPU, where one training run takes 30-40 hours. The chosen *RandomState* values are $\{9188, 2755, 361, 1321, 833\}$.

Multiple models were initially tested to find a model that is big enough to achieve high validation accuracy but small enough to fit in the 24GB GPU with an appropriate $BatchSize$. We eventually settled on using a model with six encoder blocks, each including the Positional Multi-head Self-attention module, as shown in Figure 3. The model has d=64 features and H=8 attention heads. Gradient accumulation is used with two steps each of $BatchSize$=20, giving an effective $BatchSize$ of 40.

The SIREN-based positional encoding has three layers with ω_0=$\omega_{0,initial}$=10. The differentiable span starts with an initial σ=0.3 for each positional dimension which gives an initial span that includes all tokens. Worth mentioning that each attention block has its own positional encoding and differentiable span and that positional encoding is not shared across attention heads. The hyperparameters used for the final models can be found in Table 3.

In order to make our research reproducible, we open-sourced the code[1] used to run the experiments and get these results. The code includes the implementation of all the modules introduced in this work in addition to a command-line interface to assemble the whole architecture and run those experiments. Furthermore, the configurations for all experiments are included in addition to the statistical analysis carried out on the results.

[1] https://github.com/MeenaAlfons/posattn

Table 3: **Hyperparameter values**

Hyperparameter	value
RandomState	9188, 2755, 361, 1321, or 833
BatchSize	20
Gradient accumulation steps	2
Number of training epochs	80
Learning rate	0.001
N_b Number of encoder blocks	6
d Number of features	64
H Number of heads	8
Dropout	0.1
Hidden dimension in feed-forward networks inside encoder block	128
Hidden dimension in the output feed-forward network	64
SIREN-based Positional Encoding	
L Number of layers	3
ω_0	10
$\omega_{0,initial}$	10
Differentiable Span	
t Threshold	0.1
Initial σ	0.3

5 Results

5.1 SIREN-based Positional Encoding vs Baseline

Table 4 shows the max, mean and standard deviation values for the validation accuracy achieved for each treatment. Figure 4 shows the boxplots of the validation accuracy for each treatment. It is clear that the SIREN-based positional encoding can produce comparable results to the Baseline. However, the Baseline seems more consistent in giving a higher validation accuracy than the SIREN-based one. The results show that SIREN-based positional encoding has a lower validation accuracy of 2.2% than the Baseline. Further investigation is needed to show why the specific choices we use for the SIREN network do not achieve similar validation accuracy while they theoretically can model the Baseline.

5.2 Differentiable Span without threshold

Table 5 shows the max, mean and standard deviation of the validation accuracy achieved by each treatment for the 1D and 2D settings. Figure 6a and 6b show the boxplots for each treatment in the 1D and 2D settings, respectively. The results

Table 4: Validation accuracy achieved by the model for SIREN-based positional encoding vs Baseline.

Treatment	Validation accuracy			Effect	p-value
	max	mean	std		
Baseline	**0.7624**	**0.7493**	0.01065		
SIREN-based	0.7562	0.7269	0.02311	-0.0224	0.08

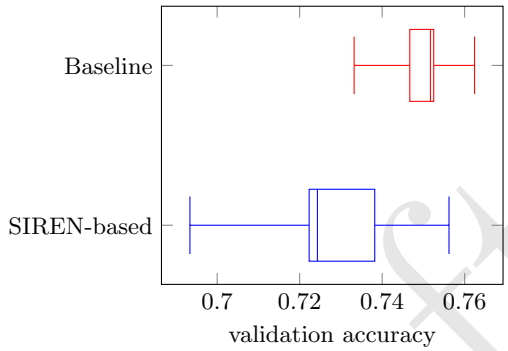


Fig. 4: Validation accuracy achieved by the model for SIREN-based positional encoding vs Baseline.

show that modulating the attention scores with a relative Gaussian function in the 1D setting decreases the validation accuracy. On the other hand, modulating the attention scores with a relative Gaussian function in a 2D setting does not significantly decrease the accuracy. In fact, the box plot for the 2D setting moves closer to the higher end of validation accuracy. This difference in the effect of applying the differentiable span between 1D and 2D settings makes sense because the learned 1D span does not perfectly reflect the actual locality feature in the data and tends to include many tokens outside the local neighbourhood. Figure 5 demonstrates this difference.

5.3 Differentiable Span with threshold

Table 6 shows the max, mean and standard deviation of the validation accuracy achieved by each treatment for the 1D and 2D settings. Figure 6a and 6b show the boxplots for each treatment in the 1D and 2D settings, respectively. Discarding the modulated attention scores outside the learned span seems to have a similar or slightly better effect than including them. That means that including the modulated scores outside the learned span has no positive effect on the validation accuracy and they are better discarded.

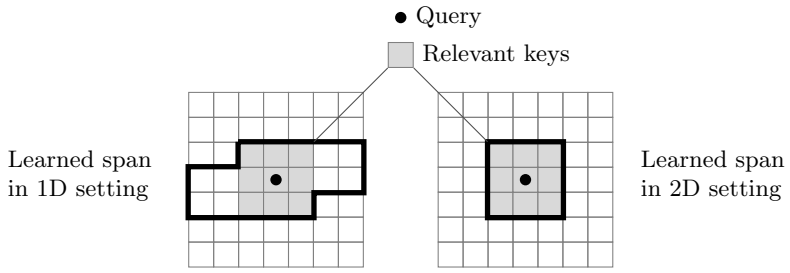


Fig. 5: The learned span in 1D and 2D settings for an input with inherent 2D structure.

Table 5: Validation accuracy achieved by the model for No differentiable span vs Differentiable span without threshold.

	Treatment	Validation accuracy			Effect	p-value
		max	mean	std		
1D	No differentiable span	**0.7562**	**0.7269**	0.02311		
	Differentiable span without threshold	0.7325	0.7094	0.02158	-0.01750	0.1364
2D	No differentiable span	**0.8134**	0.7991	0.008688		
	Differentiable span without threshold	0.8130	**0.8026**	0.01579	0.00346	0.7642

6 Discussion

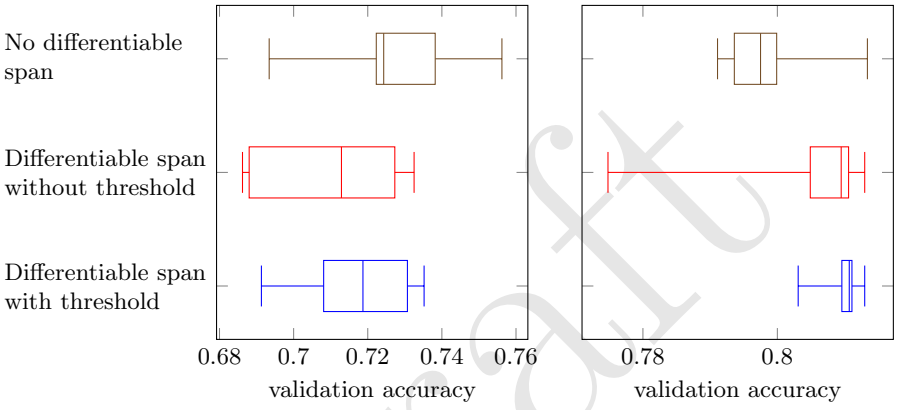
How does SIREN-based positional encoding compare with the Baseline? SIREN-based positional encoding with the hyperparameters we used achieves a lower validation accuracy than the Baseline (by 2.2% in our experiments). However, this does not discredit the approach in general because it can theoretically model the Baseline itself. Further work is needed to investigate the modelling power of both approaches by comparing them in a reconstruction task like the experiments done in [20] to compare SIRENs with other implicit neural representations.

Hyperparameters sensitivity of SIREN-based positional encoding We clearly see the huge effect of the hyperparameters ω_0 and $\omega_{0,initial}$ on the SIREN-based approach. In a selective hyperparameter tuning, we noticed that using $\omega_0 = \omega_{0,initial} \in [0.1, 100]$ results in validation accuracy in the range $[0.46, 0.74]$. In another setting where a constant $\omega_0 = 1$ is used and $\omega_{0,initial} \in [0.1, 100]$, the validation accuracy spread in the range $[0.57, 0.70]$.

How does SIREN-based positional encoding perform in multiple positional dimensions? SIREN-based positional encoding can easily make use

Table 6: Validation accuracy achieved by the model for Differentiable span without threshold vs Differentiable span with threshold.

	Treatment	Validation accuracy			Effect	p-value
		max	mean	std		
1D	Without threshold	0.7325	0.7094	0.02158		
	With threshold	**0.7352**	**0.7168**	0.01776	0.00742	0.1058
2D	Without threshold	0.8130	0.8026	0.01579		
	With threshold	0.8130	**0.8095**	0.003782	0.00694	0.4092



(a) 1D (b) 2D

Fig. 6: Maximum validation accuracy achieved by differentiable span with and without threshold for 1D and 2D settings.

of multiple positional dimensions. When used with multiple positional dimensions that match the positional dimensions of the input, SIREN-based positional encoding indeed captures the relative hints in the multi-dimensional space and boosts the validation accuracy reflecting the inherent structure in the data.

How does Differentiable Span affect the accuracy? Using a differentiable span is about balancing two things. On one side, it tries to use the locality feature in the data and focuses on modelling a limited neighbourhood. On the other side, it needs to keep the receptive field of each layer wide enough so that the effective receptive field of later layers covers the whole input and captures the long-distance relations. The Differentiable Span allows the network to adapt and find the sweet spot to balance between the relations captured in one layer and the relations that will be discovered across multiple layers.

The results show that using a differentiable span with a number of positional dimensions that match the original positional dimensions in the input reflects the actual locality feature in the data and does not reduce the validation

accuracy. However, using a number of positional dimensions smaller than the actual positional dimensions in the input decreases the validation accuracy (in our experiments by 1.75%).

Can Differentiable Span reduce the computation costs of Self-attention? Given that applying the differentiable span does not decrease the validation accuracy (with the conditions mentioned above), this opens the door for using the learned span to reduce the computation costs of self-attention. We tried two different implementations to use the learned span, but both were not computationally efficient. While we identified the bottlenecks for the specific implementations we present, we did not identify any intrinsic bottlenecks that would prevent computation savings from applying the learned span. Further work is needed to investigate and develop a way to efficiently make vectorized operations on selective ranges of the tensors involved.

Statistical significance. Our power analysis shows that using seven samples for each combination of block and treatment factors would have had a power of 87% at a significance level of 0.05. With such higher power, there is a higher probability of detecting a significant effect when it exists. Therefore, using two more samples for each combination of block and treatment factors would have made the results more statistically significant.

7 Conclusion & Future Work

We introduce SIREN-based positional encoding, a simple way to add multi-dimensional positional encoding to attentive models. It is a parameter-based alternative to positional encodings based on fixed sinusoidal features that can easily extend to multiple positional dimensions. However, we have shown that in the 1D setting, fixed sinusoidal features followed by a linear layer (the baseline) consistently outperforms SIREN-based positional encoding. The scope of this result is only limited to the hyperparameters we used for the SIREN-based positional encoding.

We also introduce Differentiable Span, a data-driven approach to local self-attention. We have shown that it boosts the accuracy of the model when the used positional dimensions match the data's inherent structure. However, we failed to develop an implementation to reduce the computation costs of self-attention depending on the learned span.

How to improve SIREN-based positional encoding? comprehensive comparison is needed to get deeper insights into what is holding SIREN-based positional encoding from outperforming the baseline while the former has the theoretical capacity of modelling the baseline. Moreover, the parameterisation for the SIREN-based positional encoding is very sensitive to the values of the hyperparameters. Therefore, a more stable parameterisation is needed to reduce the challenging effort of hyperparameter search.

Differential Span limitations. Besides the possible improvement in accuracy when using local self-attention, the main goal is to reduce the computation

costs of the attention operation. Further work is needed to develop an efficient implementation that translates the (learned) reduced attention span into reduced computation costs.

References

1. Barr, A.H.: The einstein summation notation. An Introduction to Physically Based Modeling (Course Notes 19), pages E **1**, 57 (1991)
2. Chaudhari, S., Mithal, V., Polatkan, G., Ramanath, R.: An attentive survey of attention models. ACM Transactions on Intelligent Systems and Technology (TIST) **12**(5), 1–32 (2021)
3. Dai, Z., Yang, Z., Yang, Y., Carbonell, J., Le, Q.V., Salakhutdinov, R.: Transformer-xl: Attentive language models beyond a fixed-length context. arXiv preprint arXiv:1901.02860 (2019)
4. Deng, L.: The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine **29**(6), 141–142 (2012)
5. Gehring, J., Auli, M., Grangier, D., Yarats, D., Dauphin, Y.N.: Convolutional sequence to sequence learning. In: International Conference on Machine Learning. pp. 1243–1252. PMLR (2017)
6. Ghojogh, B., Ghodsi, A.: Attention mechanism, transformers, bert, and gpt: Tutorial and survey (2020)
7. Ho, J., Kalchbrenner, N., Weissenborn, D., Salimans, T.: Axial attention in multidimensional transformers. arXiv preprint arXiv:1912.12180 (2019)
8. Huang, C.Z.A., Vaswani, A., Uszkoreit, J., Shazeer, N., Simon, I., Hawthorne, C., Dai, A.M., Hoffman, M.D., Dinculescu, M., Eck, D.: Music transformer. arXiv preprint arXiv:1809.04281 (2018)
9. Jarque, C.M., Bera, A.K.: A test for normality of observations and regression residuals. International Statistical Review/Revue Internationale de Statistique pp. 163–172 (1987)
10. Kingma, D.P., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980 (2014)
11. Kitaev, N., Kaiser, Ł., Levskaya, A.: Reformer: The efficient transformer. arXiv preprint arXiv:2001.04451 (2020)
12. Krizhevsky, A., Nair, V., Hinton, G.: Cifar-10 (canadian institute for advanced research) http://www.cs.toronto.edu/~kriz/cifar.html
13. Li, Y., Si, S., Li, G., Hsieh, C.J., Bengio, S.: Learnable fourier features for multidimensional spatial positional encoding. Advances in Neural Information Processing Systems **34** (2021)
14. Liu, P.J., Saleh, M., Pot, E., Goodrich, B., Sepassi, R., Kaiser, L., Shazeer, N.: Generating wikipedia by summarizing long sequences. arXiv preprint arXiv:1801.10198 (2018)
15. Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., Tran, D.: Image transformer. In: International Conference on Machine Learning. pp. 4055–4064. PMLR (2018)
16. Ramachandran, P., Parmar, N., Vaswani, A., Bello, I., Levskaya, A., Shlens, J.: Stand-alone self-attention in vision models. Advances in Neural Information Processing Systems **32** (2019)
17. Romero, D.W., Bruintjes, R.J., Tomczak, J.M., Bekkers, E.J., Hoogendoorn, M., van Gemert, J.C.: Flexconv: Continuous kernel convolutions with differentiable kernel sizes. arXiv preprint arXiv:2110.08059 (2021)
18. Romero, D.W., Kuzina, A., Bekkers, E.J., Tomczak, J.M., Hoogendoorn, M.: Ckconv: Continuous kernel convolution for sequential data. arXiv preprint arXiv:2102.02611 (2021)

19. Shaw, P., Uszkoreit, J., Vaswani, A.: Self-attention with relative position representations. arXiv preprint arXiv:1803.02155 (2018)
20. Sitzmann, V., Martel, J., Bergman, A., Lindell, D., Wetzstein, G.: Implicit neural representations with periodic activation functions. Advances in Neural Information Processing Systems **33**, 7462–7473 (2020)
21. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, Ł., Polosukhin, I.: Attention is all you need. Advances in neural information processing systems **30** (2017)
22. Zhao, H., Jia, J., Koltun, V.: Exploring self-attention for image recognition. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 10076–10085 (2020)

Appendix

A Implementation Details

In this work, we present the possibility and prove the effectiveness of learning the attention span size from the data during training. However, the learned span must be used to limit the operations involved to make any gains in computation costs. The following sections focus on implementing Self-Attention with Relative Positional Encoding and Differentiable Span in vectorized form using tensors. Version 1 takes a standard attention implementation and applies the learned span as an additional step, thus making no computation gains. Versions 2 and 3 try to use the learned span to limit the operations and reduce the computation costs. All three versions have the same parameters, produce the same outputs, and result in the same gradients on the model parameters.

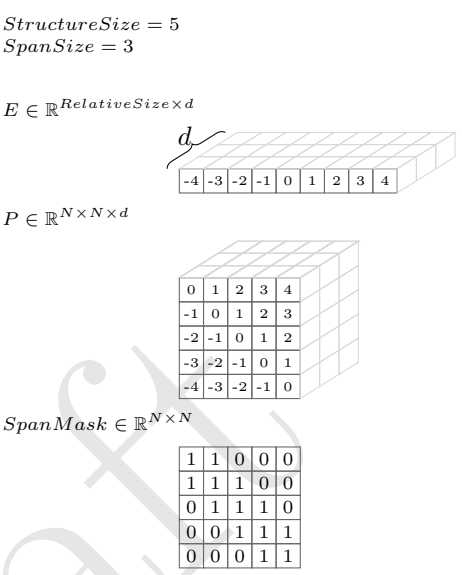


Fig. 7: **Version 1 - Alignment.** This is a 1D example to show the alignment of values in P and $SpanMask$ in an $N \times N$ structure. G^* has the exact alignment as P.

A.1 Version 1

Many tensors are involved in the operations of self-attention with relative positional encoding and differentiable span. Therefore, the layouts of those tensors need to be aligned to allow efficient and straightforward operations. For Version 1, the involved tensors have the following shapes:

- $A \in \mathbb{R}^{N \times N}$
- $P \in \mathbb{R}^{N \times N \times d}$
- $G^* \in \mathbb{R}^{N \times N}$
- $SpanMask \in \mathbb{B}^{N \times N}$
- $N = prod(StructureSize)$

Notice the similar structure $N \times N$ in all these tensors where the first dimension represents the queries and the second dimension represents the keys. Therefore, the values of any of those quantities for keys relevant to a specific query i are laid out in a row with the query index i. Figure 7 depicts an example showing the aligned values. The example uses a 1-dimensional structure where $StructureSize$=5 and $SpanSize$=3.

While the input has a multi-dimensional structure whose size is represented by $StructureSize \in \mathbb{N}^n$, all the input tokens have unique ids in the range $[0, N)$. These ids are assigned according to the row-major convention. Therefore, the queries or keys dimension with size N can be factorized and represented with multiple dimensions $StructureSize_0 \times ... \times StructureSize_{n-1}$ as shown in Formula 2. Going back and forth between the factorized and flattened dimensions will prove helpful in efficiently generating those tensors and operating on them. The same idea applies to $RelativeSize$, flattened into N_r=$prod(RelativeSize)$.

$$A \in \mathbb{R}^{N \times N} \quad \overset{\text{content-wise}}{\equiv} \quad A \in \mathbb{R}^{StructureSize \times StructureSize} \tag{2}$$

Algorithm 9: Version 1: Construct P from E

input : $E \in \mathbb{R}^{RelativeSize \times d}$ the positional encoding for all relative positions in a multi-dimensional region of size $RelativeSize$.

output: $P \in \mathbb{R}^{N \times N \times d}$ the relative positional encoding for each key respective to each query.

1. Unfold patches of size $StructureSize$ from the full volume E of size $RelativeSize$. The features d are extracted for each patch. This results in: $P \in \mathbb{R}^{StructureSize \times StructureSize \times d}$

2. Flip the positional dimensions for queries. $P[..., StructureSize_i - j, ...] = P[..., j, ...]$ for $j \in [0, StructureSize_i)$ and $i \in [0, n)$

3. Flatten each set of positional dimensions $StructureSize$ into N=$prod(StructureSize)$ which results in: $P \in \mathbb{R}^{N \times N \times d}$

In Version 1, the positional encoding vectors are generated for all the relative positions in a multi-dimensional grid of size $RelativeSize$ producing $E \in \mathbb{R}^{RelativeSize \times d}$. The generated positional encoding in E is then used to construct P by extracting the relative positional encoding for each key respective to each query and putting them in the adopted layout. The steps involved in constructing P from E are shown in Algorithm 9.

Similarly, the values of the Gaussian function is generated for all the relative positions in a multi-dimensional grid of size $RelativeSize$ producing $G \in \mathbb{R}^{RelativeSize}$. The generated positional encoding in G is then used to construct G^* by extracting the relative Gaussian values for each key respective to each query and putting them in the adopted layout. G^* construction from G uses the same steps in Algorithm 9 by removing the feature dimension d.

Finally, a boolean mask $SpanMask \in \mathbb{B}^{N \times N}$ is generated to indicate which keys are inside the local neighbourhood of the respective query according to the learned span of size $SpanSize$. The construction of $SpanMask$ from $SpanSize$ is shown in Algorithm 10.

Algorithm 10: Version 1: Construct $SpanMask$ from $SpanSize$

input : $SpanSize \in \mathbb{R}^n$ The size of a relative multidimensional span centred around a query.

output: $SpanMask \in \mathbb{B}^{N \times N}$ a boolean mask marking the keys lying inside the SpanSizerespective to each query.

Start with a tensor $SpanMask \in \mathbb{B}^{StructureSize \times StructureSize}$ and fill it as follows (upper triangle and lower triangle indices can be manipulated to achieve this effect):

$$SpanMask[..., i_j, ..., i_{j+n}, ...] = \begin{cases} False & \text{if } i_j + \frac{SpanSize_j - 1}{2} < i_{j+n} \\ False & \text{if } i_j - \frac{SpanSize_j - 1}{2} > i_{j+n} \\ True & otherwise \end{cases}$$

where $j \in [0, n)$

Flatten each set of positional dimensions $StructureSize$ into N=$prod(StructureSize)$ which results in:
$SpanMask \in \mathbb{B}^{N \times N}$

A.2 Version 2

Version 2 tries to use a tight layout for the values to reduce the memory used for unneeded values outside the $SpanSize$ and remove their computation costs. This requires changing the form of attention scores A mainly to include the needed attention scores. For Version 2, the involved tensors have the following shapes:

- $A \in \mathbb{R}^{N \times N_p}$
- $P \in \mathbb{R}^{N \times N_p \times d}$
- $G^* \in \mathbb{R}^{N \times N_p}$
- $SpanMask \in \mathbb{B}^{N \times N_p}$
- $N_p = prod(PatchSize)$
- $PatchSize = [p_i | p_i = min(SpanSize_i, StructureSize_i) \forall i \in n]^T$

Notice that the keys dimension no longer holds all the N keys. Instead, it holds the N_p keys inside a patch of size $PatchSize$. This $PatchSize$ will equal the $SpanSize$ if the learned span is smaller than the $StructureSize$, hence including a smaller number of keys. However, if the $SpanSize$ is larger than the $StructureSize$, the $PatchSize$ will equal the $StructureSize$, and all the keys will be included. Notice that each dimension's size in $SpanSize$ is independently compared with the corresponding dimension's size in $StructureSize$, producing the corresponding dimension's size of the patch.

In order to carry out the operations in Version 2, an additional tensor $K^* \in \mathbb{R}^{N \times N_p \times d}$ needs to be constructed, holding the relevant keys for each query. K^* is then used with $Q \in \mathbb{R}^{N \times d}$ and $P \in \mathbb{R}^{N \times N_p \times d}$ to calculate the attention scores A, as shown in Formula 3. The steps involved in constructing K^* from the keys tensor $K \in \mathbb{R}^{N \times d}$ are shown in Algorithm 11. The implications of generating this additional tensor on the total computational costs are discussed in the comparison held in Appendix A.4.

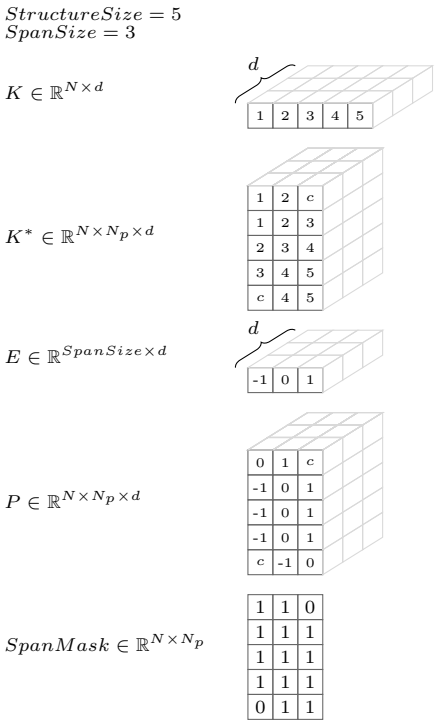


Fig. 8: **Version 2 - Alignment.** This is a 1D example to show the alignment of values in K^*, P, and $SpanMask$ in an $N \times N_p$ structure. G^* has the exact alignment as P. c is an arbitrary constant to fill positions in the tensor that will be discarded, eventually.

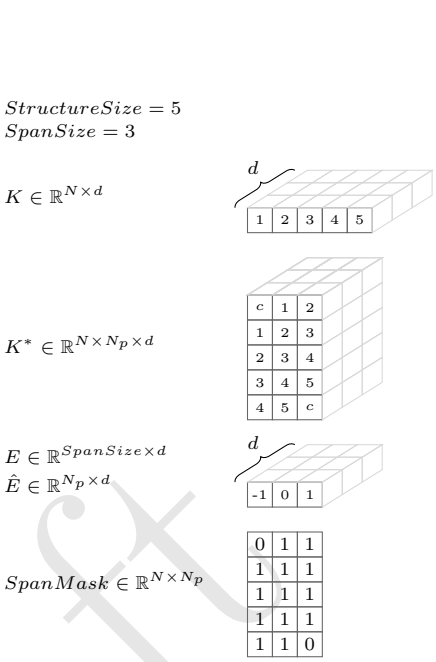


Fig. 9: **Version 3 - Alignment.** This is a 1D example to show the alignment of values in K^*, $\hat{E}$ and $SpanMask$ in an $N \times N_p$ structure. c is an arbitrary constant to fill positions in the tensor that will be discarded, eventually.

Figure 8 contains an example showing the aligned values. The example uses 1-dimensional structure where $StructureSize$=5 and $SpanSize$=3. Algorithm 12 shows the construction of P from E, which also works for constructing G^* from G. Finally, Algorithm 13 shows the construction of $SpanMask$ from $PatchSize$.

$$A_{qk} = \underbrace{(Q_{qd}K^*_{qkd})_{qk}}_{\text{content attention}} + \underbrace{(Q_{qd}P_{qkd})_{qk}}_{\text{positional attention}} + \underbrace{(u_d K_{qkd})_{qk}}_{\text{content bias}} + \underbrace{(v_d P_{qkd})_{qk}}_{\text{positional bias}} {}^*$$

where q is the query dimension ($= N$)
k is the key dimension ($= N_p$)
d is the feature dimension (3)
u_d, v_d are learnable parameters
P_{qkd} is the relative positional encoding for all keys respective to all queries
K^*_{qkd} is the relevant keys for each query
* Einstein notation

Algorithm 11: Version 2: Construct K^* from K

input : $K \in \mathbb{R}^{N\times d}$ the keys. The $PatchSize \in \mathbb{R}^n$ where n is the number of positional dimensions.

output: $K^* \in \mathbb{R}^{N\times N_p\times d}$ the relevant keys for each query lying inside the relative region of $PatchSize$ centred around the respective query where N_p is the total number of relative positions in the region of size $PatchSize$.

1. Reshape K by decomposing the dimension N into its positional structure with dimensions $StructureSize$. This results in:
$K \in \mathbb{R}^{StructureSize\times d}$

2. Unfold patches of size $PatchSize$ from the full volume K of size $StructureSize$. The features d are extracted for each patch. This results in patches for each position in a region of size
$UnfoldedSize = StructureSize - PatchSize + 1$.
$K^* \in \mathbb{R}^{UnfoldedSize\times PatchSize\times d}$

4. Repeat the first and last patch in K^* until the first set of positional dimensions in K^* become $StructureSize$. This results in:
$K^* \in \mathbb{R}^{StructureSize\times PatchSize\times d}$

5. Flatten the first set of positional dimensions $StructureSize$ into $N = prod(StructureSize)$ and the second set of positional dimensions $PatchSize$ into $N_p = prod(PatchSize)$, which results in:
$K^* \in \mathbb{R}^{N\times N_p\times d}$

Algorithm 12: Version 2: Construct P from E

input : $E \in \mathbb{R}^{PatchSize \times d}$ the positional encoding for relative positions in a multi-dimensional region of size $PatchSize \in \mathbb{R}^n$ where n is the number of positional dimensions.

output: $P \in \mathbb{R}^{N \times N_p \times d}$ the relative positional encoding for each key in N_p respective to each query where N_p is the total number of relative positions in region of size $PatchSize$.

1. Pad E with an arbitrary constant c in order to achieve the size $PaddedSize$=$2PatchSize-1$. This results in: $E_{padded} \in \mathbb{R}^{PaddedSize \times d}$

2. Unfold patches of size $PatchSize$ from the full volume E of size $PaddedSize$. The features d are extracted for each patch. This results in: $P \in \mathbb{R}^{PatchSize \times PatchSize \times d}$

3. Flip the positional dimensions for queries. $P[..., StructureSize_i - j, ...] = P[..., j, ...]$ for $j \in [0, StructureSize_i)$ and $i \in [0, n)$

4. Repeat the middle patch in P number of times equal $StructureSize - PatchSize + 1$. This results in: $P \in \mathbb{R}^{StructureSize \times PatchSize \times d}$

5. Flatten the first set of positional dimensions $StructureSize$ into N=$prod(StructureSize)$ and the second set of positional dimensions $PatchSize$ into N_p=$prod(PatchSize)$ which results in: $P \in \mathbb{R}^{N \times N_p \times d}$

A.3 Version 3

Version 3 uses a tight layout that is different from the one used in Version 2. Version 3 uses a simpler approach that assumes that most of the time, the learned $SpanSize$ will be smaller than the $StructureSize$. In general, $SpanSize$ could grow up to $RelativeSize$=$2StructureSize-1$. With this assumption in mind, the $PatchSize$ is taken to always equal $SpanSize$, which is simpler than its counterpart in Version 2. For Version 3, the involved tensors have the following shapes:

- $A \in \mathbb{R}^{N \times N_p}$
- $G^* \in \mathbb{R}^{N \times N_p}$
- $SpanMask \in \mathbb{B}^{N \times N_p}$
- $N_p = prod(SpanSize)$

The chosen layout in Version 3 removes the need for constructing $P \in \mathbb{R}^{N \times N_p \times d}$ because all its rows will have the same content, a flattened version of $E \in \mathbb{R}^{SpanSize \times d}$, which we will call $\hat{E} \in \mathbb{R}^{N_p \times d}$. $\hat{E}$ will be used instead of P in calculating attention scores A, as shown in Formula 4. For the same reason,

Algorithm 13: Version 2: Construct $SpanMask$ from $PatchSize$

input : $SpanSize \in \mathbb{R}^n$ The size of a relative multidimensional span centred around a query.

output: $SpanMask \in \mathbb{B}^{N \times N}$ a boolean mask marking the keys lying inside the SpanSizerespective to each query.

Start with a tensor $SpanMask \in \mathbb{B}^{StructureSize \times PatchSize}$ and fill it as follows (upper triangle and lower triangle indices can be manipulated to achieve this effect):

$$SpanMask[..., i_j, ..., i_{j+n}, ...] = \begin{cases} False & \text{if } i_j + \frac{SpanSize_j - 1}{2} < i_{j+n} \\ False & \text{if } i_j - \frac{SpanSize_j - 1}{2} - StructureSize_j + PatchSize_j > i_{j+n} \\ True & otherwise \end{cases}$$

where $j \in [0, n)$

Flatten the first set of positional dimensions $StructureSize$ into N=$prod(StructureSize)$ and the second set of positional dimensions $PatchSize$ into N_p=$prod(PatchSize)$, which results in: $SpanMask \in \mathbb{B}^{N \times N_p}$

G^* is no longer needed because all its rows will have the same content. Instead, $\hat{G} \in \mathbb{R}^{N_p}$, a flattened version of $G \in \mathbb{R}^{SpanSize}$, is used to modulate the attention scores $A \circ \hat{G}$ by broadcasting the Hadamard product with $\hat{G}$ on all the rows of A.

Similar to Version 2, $K^* \in \mathbb{R}^{N \times N_p \times d}$ needs to be constructed holding the relevant keys for each query. Algorithm 14 shows the construction of K^* from K. Finally, Algorithm 15 shows the construction of $SpanMask$ from $PatchSize$. Figure 9 contains an example showing the aligned values. The example uses 1-dimensional structure where $StructureSize = 5$ and $SpanSize = 3$. Notice that $\hat{E}$ looks the same as E for this 1-dimensional example.

$$A_{qk} = \underbrace{(Q_{qd} K^*_{qkd})_{qk}}_{\text{content attention}} + \underbrace{(Q_{qd} \hat{E}_{kd})_{qk}}_{\text{positional attention}} + \underbrace{(u_d K_{qkd})_{qk}}_{\text{content bias}} + \underbrace{(v_d \hat{E}_{kd})_k}_{\text{positional bias}} \quad ^* \tag{4}$$

where q is the query dimension ($= N$)
k is the key dimension ($= N_p$)
d is the feature dimension
u_d, v_d are learnable parameters
P_{qkd} is the relative positional encoding for all keys respective to all queries
K^*_{qkd} is the relevant keys for each query
* Einstein notation

Algorithm 14: Version 3: Construct K^* from K

input : $K \in \mathbb{R}^{N \times d}$ the keys. The $SpanSize \in \mathbb{R}^n$ where n is the number of positional dimensions.

output: $K^* \in \mathbb{R}^{N \times N_p \times d}$ the relevant keys for each query lying inside the relative region of $SpanSize$ centred around the respective query where N_p is the total number of relative positions in region of size $SpanSize$.

1. Reshape K by decomposing the dimension N into its positional structure with dimensions $StructureSize$. This results in:
$K \in \mathbb{R}^{StructureSize \times d}$

2. Pad K with an arbitrary constant c in order to achieve the size $PaddedSize$=$StructureSize$+$SpanSize$−1. This results in:
$K_{padded} \in \mathbb{R}^{PaddedSize \times d}$

3. Unfold patches of size $SpanSize$ from the full volume K of size $PaddedSize$. The features d are extracted for each patch. This results in:
$K^* \in \mathbb{R}^{StructureSize \times SpanSize \times d}$

4. Flatten the first set of positional dimensions $StructureSize$ into N=$prod$($StructureSize$) and the second set of positional dimensions $SpanSize$ into N_p=$prod$($SpanSize$) which results in:
$K^* \in \mathbb{R}^{N \times N_p \times d}$

A.4 Comparison V1 vs V2 vs V3

Versions 1, 2 and 3 are all similar in their effect. The differences are in the data alignment and the respective operations working on that alignment to give the same effect. Versions 2 and 3 were motivated by reducing the memory and time needed for the involved operations. However, they turn out to be slower in practice. The reason stems from an additional dimension that has been neglected during the previous presentation of all three versions, which is the $BatchSize$. Figure 10 shows the operations in all three versions with the involved tensors sizes at each step, including the $BatchSize$.

A closer look at these operations while taking the $BatchSize$ into account shows that the generation of K^* significantly affects the memory needed for the algorithm. K^* has an order of magnitude more memory than any other tensor in these operations by the magnitude of $BatchSize$. While Versions 2 and 3 try to save memory on the generation of positional encoding P and Differentiable Span G^* and save FLOPs by only computing the needed attention scores, they end up using much more memory by unfolding K into K^*. On the other hand, Version 1 uses an alignment where all keys exist for each query, and adapts P and G^* to that alignment. In doing so, Version 1 uses less memory by not carrying out the massive operation of constructing K^* while increasing the memory needed for P and G^* with a constant factor.

Algorithm 15: Version 3: Construct $SpanMask$ from $SpanSize$

input : $SpanSize \in \mathbb{R}^n$ The size of a relative multidimensional span centred around a query.

output: $SpanMask \in \mathbb{B}^{N \times N}$ a boolean mask marking the keys lying inside the SpanSizerespective to each query.

Start with a tensor $SpanMask \in \mathbb{B}^{StructureSize \times SpanSize}$ and fill it as follows (upper triangle and lower triangle indices can be manipulated to achieve this effect):

$SpanMask[..., i_j, ..., i_{j+n}, ...] =$

$$\begin{cases} False & \text{if } i_j + i_{j+n} < \frac{SpanSize_j - 1}{2} \\ False & \text{if } i_j + i_{j+n} > \frac{SpanSize_j - 1}{2} + StructureSize_j - 1 \\ True & otherwise \end{cases}$$

where $j \in [0, n)$

Flatten the first set of positional dimensions $StructureSize$ into N=$prod(StructureSize)$ and the second set of positional dimensions $SpanSize$ into N_p=$prod(SpanSize)$ which results in: $SpanMask \in \mathbb{B}^{N \times N_p}$

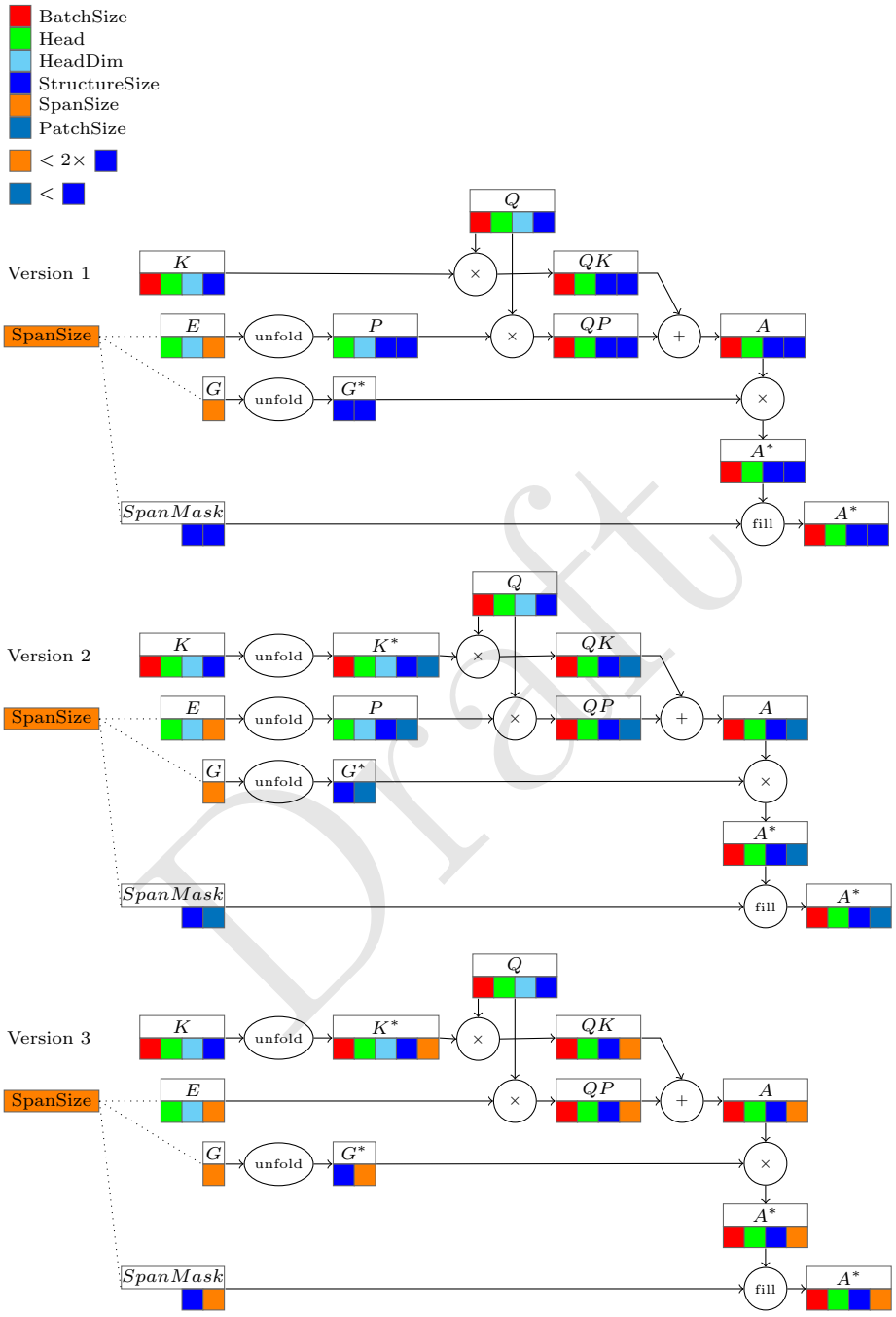


Fig. 10: **Comparison V1 vs V2 vs V3** This is the implementation of three versions for self-attention with relative positional encoding and differentiable span, including the sizes of the involved tensors.