Master Positional Encoding: Part II

We upgrade to relative position, present a bi-directional relative encoding, and discuss the pros and cons of letting the model learn this all for you

Jonathan Kernes · Following
Published in Towards Data Science · 23 min read · Feb 25, 2021

Photo by Sean Stratton on Unsplash

This is Part II of the two-part series “Master Positional Encoding.” If you would like to know more about the intuition and basics of positional encoding, please see my first article.

Whereas the first article discussed the meaning of the fixed sinusoidal **absolute** positional encodings, this article will focus on **relative** positional encodings. We will also discuss the pros and cons of using a learnable positional encoding as well. Along the way I give a proposal for implementing a *bi-directional relative positional encoding*, based on the architecture of Transformer-XL. I haven’t been able to find anyone that discusses this, so please chime in if you can shed some light on whether anyone has pursued this.

If there’s one TL;DR takeaway from this article it’s this:

> You should put positional encoding information directly into the logits.

Here’s the outline. Feel free to skip to whatever interests you!:

Section I (introduction):

- Why study relative encodings?
- How does a model think about relative encodings?

Section II (computing stuff):

- How to build a relative positional encoding given an absolute one.

- An O(n) algorithm for constructing relative positional encodings.
- Code for the aforementioned algorithm, and how you can bi-directionalize it.

Section III (learnable encodings):

- Why you should be injecting positional information directly into logits.
- Pros and cons of learnable vs fixed encodings.
- The Embedding-Position correlation decomposition of logits.

Section I (introduction):

Position in day-to-day language

Quick quiz. How would you respond to the following question:

Where is the nearest bank?

1. Next to the theater.
2. At the corner of First and Main St.
3. At Latitude: 41.559599, Longitude: 2.431680.

Solution: all of those options are correct! That's because in reality, the precise answer of *where* something is, is not a well-defined question. There's no way you could answer that question in three words with enough precision to lead every person on the planet to the exact same location. However, depending on who you're talking to you (a roommate, coworker, news anchor), they might actually find the bank.

Contextual position vs computable position

There are two primary reasons why "Next to the theater" can be an acceptable response. First,

> *"Next to the theater" describes a **relative position**.*

Provided we know the position of the theater, we can **compute** "Next to"+"theater" to arrive at a new location. Sure that seems reasonable, but how do we know the position of the theater? Didn't we just say position is ill-defined? Correct. That brings us to our second point:

> *Every response carries with it **contextual information**, including the current position of both people in the conversation.*

When I answer "Next to the theater", it is implied that the person asking the question knows where I am. This is the **contextual information**. The questioner, in turn, has their own contextual information as well. In day-to-day language, we default to computing positions **relative** to our own position. This imbues position with a well defined meaning: **position is always relative.**

Since we are trying to build machines to understand human logic, we have to somehow instill in them these understandings of position. Let's solve the following problem:

We give a model some paragraph, and ask it to compute the location of the bank. Let's say we feed it the sentence:

"The **bank** is not far from **here**. **It** is **next to** the **theater**."

If you only kept the bold words, you would sound like a caveman, but you would probably figure out where the bank is. A machine doesn't need to conform to societal expectations, so it's free to speak caveman. We can imagine the machine performing the following computation:

1. Look up in a hash table `Location["here"]=0` . Remember, people use relative positional encoding, so the machine has learned that the location of "here" is zero.
2. Look up in a hash table the operation `Operation["next to"]=lambda x: x+1` which just adds a distance of 1 to any position.
3. Look up in a hash table `Location["next to"]=street_pos` .
4. Compute `Location["bank"] = Operation["next to"](Location["theater"]["here"])` .

Step 4 is the key part. Everything else is not difficult to learn,. They are just hash-table values. In fact, you could just hard code these into a model and call it a day. Step 4 however, requires logic, which is much more difficult to hard code. Before getting into this, let's summarize what we have seen using our previous terms defined in bold-face.

Sentence 1 — "The bank is not far from here" — is the **contextual information**. Of all possible values of "theater" in the `Location["theater"]` hash table, it tells us to specifically to use the one located "here".

Sentence 2 — "It [the bank] is next to the theater" — is the actual response the questioner hears. It contains **relative positional information** relating the location of "bank" to the location of "theater".

If this were a Recurrent Neural Network (RNN), we could think of the initial hidden state as providing the **contextual information**, while the actual RNN *computes* the **relative positional information.** This is why stateful models are so much better at extracting long-term dependencies.

From abstract position to actual positions in a sequence

Up until now, we have been discussing position in terms of numbers assigned to abstract things, like a bank, theater, or yourself. That was helpful for driving home the point that in life we only care about relative positions.

We are now going to discuss positions in terms of location in a sequence. In Step 4 of our thought experiment, our model had to figure out how to put together a bunch of numbers to compute location["bank"]. **Grammar** can be thought of as a set of (often broken and inconsistent) rules designed to take a bunch of words, and assign them a meaning based on their position in a sequence. For example, take the sentence:

"I live to eat."

This expresses the speakers love of food. They would die if they couldn't eat (which I guess is not just hyperbolically true but also literally...). The grammar tells us that the verb "eat" is the purpose of living. So, you might imagine a model that learns "When you see the structure 'verb'+ 'to', the word at position index['verb']+2 describes the purpose of verb".

Indeed, let's switch the two verbs to write:

"I eat to live."

We now get the opposite meaning! The speaker implies he/she doesn't love food, and only eats in order continue living. However, this still fits the grammatical model we learned! index['eat']+2 = index['live'] still works, live is the purpose of eating here.

The model has acted human. It took its own position (index['eat']) and treated that as the "absolute zero" position. It then said, "hey, I only care about the thing that is 2 steps ahead of me", which in this case was index['live']. The model only needed **relative position** for computation.

In our previous article, we spent a lot of time dwelling on creating **absolute positional encodings**. However, as we've just argued, we really only need *relative* positions. It's always possible to describe things using absolute position, as our latitude and longitude response showed, but usually that's more info than we need, and internally we probably just find a way to convert that to something relative anyway.

The reason for starting with **absolute positional encodings**, is that they can very easily be turned into **relative positional encodings**. Let's now get into a bit of the mathematics of relative stuff.

Section II (computing stuff):

Relative-ize Your Absolute Encoding: a simple starter example

Let's start off with a simple example, to get a feel for what's going on. Consider a 1D sequence with the following absolute positional encoding vector (the letter A for absolute)

$$\mathbf{A} = (A_0, \dots, A_{n-1}) \equiv (1, \dots, n)$$

If I were to ask you what the position is between any two points *A_i* and *A_j*, represented by a distance function *d(A_i, A_j)*, the answer should hopefully be obvious: *d(A_i, A_j)* = |*A_i-A_j*| = |*i-j*|. Our challenge is not to *compute* relative position, but rather, to *represent* the relative position we computed. Since there are *n* possible positions, there are n^2 possible pairings of positions *A_i* and *A_j*. This suggests that our relative encoding should be an (n, n) matrix. Denote the relative encoding by the matrix **R** (R for relative). Then, we can account for all possible relative positions by filling in the entries of **R** according to the formula

$$R_{i,j} = |A_i - A_j| \equiv |i - j|$$

Remember, when a letter is not bold-faced, it represents just a scalar value, so this is an equation for each of the n^2 entries. The actual matrix now looks like this

$$\mathbf{R} = \begin{pmatrix} 0 & 1 & \cdots & n-1 \\ 1 & 0 & \cdots & n-2 \\ \vdots & \vdots & \ddots & \vdots \\ n-1 & n-2 & \cdots & 0 \end{pmatrix}$$

Notice the zeros on the diagonal, which signify that an index's relative position to itself is zero. This particular matrix is what's known as a Circulant Matrix, which is a special case of a more general class of matrices called a Toeplitz Matrix. Had we relaxed the symmetry condition A_*i-A_j=A_j-A_i* the lower diagonals would be negative, so the matrix would be only Toeplitz. You know you're getting a Toeplitz matrix when each diagonal is constant. The main diagonal is all 0's, the next is all 1's, the next all 2's, etc. Getting a

Toeplitz matrix is highly desirable, as it will greatly simplify the computation to come.

We notice that in the entire (n, n) matrix, there are only *n* unique values. These are (0, 1, …, n-1). The matrix representation is thus highly redundant. We also notice that these unique values are *entirely* represented by the entries of the absolute encoding vector **A**, plus the value 0. We thus postulate that we should be able to determine **R** by only manipulating the matrix **A**, which suggests that we can form **R** in O(n) computations as opposed to $O(n^2)$. This all depends on the fact that our matrix **R** is indeed Toeplitz. However, note that if the sequence values were not equally spaced, e.g. we had something like *A = (1, 2, 5, 6, 9, …, n)*. Then we would NOT have discovered a Toeplitz matrix (compute one diagonal to convince yourself, like A_3-A_2 != A_2-A_1).

This suggests that if we want a Toeplitz relative encoding matrix (which we definitely do) our absolute positional encoding vector must satisfy either of two conditions. The first is that all its values are equally spaced, as was the case in our simple example. But, we know from Part I that our absolute positions will not be simple integers, but instead some complicated d_model-dimensional vector. Satisfying the first conditions is a no-go. The second condition is to demand that each entry of the absolute positional encoding vector actually represents the **fixed relative positions**.

Ok I know that might not make any sense, so consider this explanation. Assume there are 3 positions in the world, called X, Y, and Z. We assume that they are equally spaced, whatever that means, and consecutive, but we can't actually measure their absolute positions. What we can measure, is the output of some distance function *d(X, Y) = NUMBER*. We might find something like *d(X,Y) = 2, d(X,Z) = 6, d(Y,Z)=2,* and the rest 0. The only thing

this distance function needs to do is spit out d(x,x)=0, d(x,y)=d(y,x), and ensure that if (X-Y)>(Z-T), then d(X,Y)>d(Z,T). In other words, to successfully describe positional encoding, we just need to specify the relative positions of each element in the sequence, call this |i-j|, then compute a fixed value R(|i-j|) that signals to whoever reads in the value how close *i, j* are. In this example, if we feed in R(|i-j|)=6, our model learns that 6 implies a separation of 2!

All of that is a lot of words for something that can succinctly be written in an equation definition:

$$R_{i,j} = |A_{i-j}|$$

> This is our defining equation for relative positional encoding.

All we've done is move |i-j| into the index of **A**. It's a tiny change, but it carries with it a larger conceptual change. The final thing we mention in this subsection is that the relative positional encoding matrix is of greater rank than the absolute positional encoding vector that it is was derived from. This is a general feature. **The relative encoding will always be of rank one greater than its absolute encoding.** For the 1D case this means we get a vector and a matrix. Beyond that, we get a rank-*k* tensor and a rank-*(k+1)* tensor. We've now exhausted the usefulness of toy examples, so let's move on to a real-life positional encoding

The Fixed Sinusoidal Relative Position Tensor

Before making a tool, it's usually helpful to know what it's going to be used for. In this case, let's consider the Attention model. The reason we need positional encoding, is to give the attention mechanism a sense of the position of items in the sequence that it attends on. Attention is represented by a matrix **M** of **logits**, where the entry M_{ij} gives the log probability that a given query at index *i* matches with a key at index *j*. Notice that the dimensions of the logits matrix are (seq_len, seq_len). These are precisely the dimensions of **R**, which makes sense. Logits are comparing things at one position to another. It thus makes sense for us to directly inject positional information into the logits of our model, and indeed that will be our aim. We will refer to these positional logits as just that — **positional logits.**

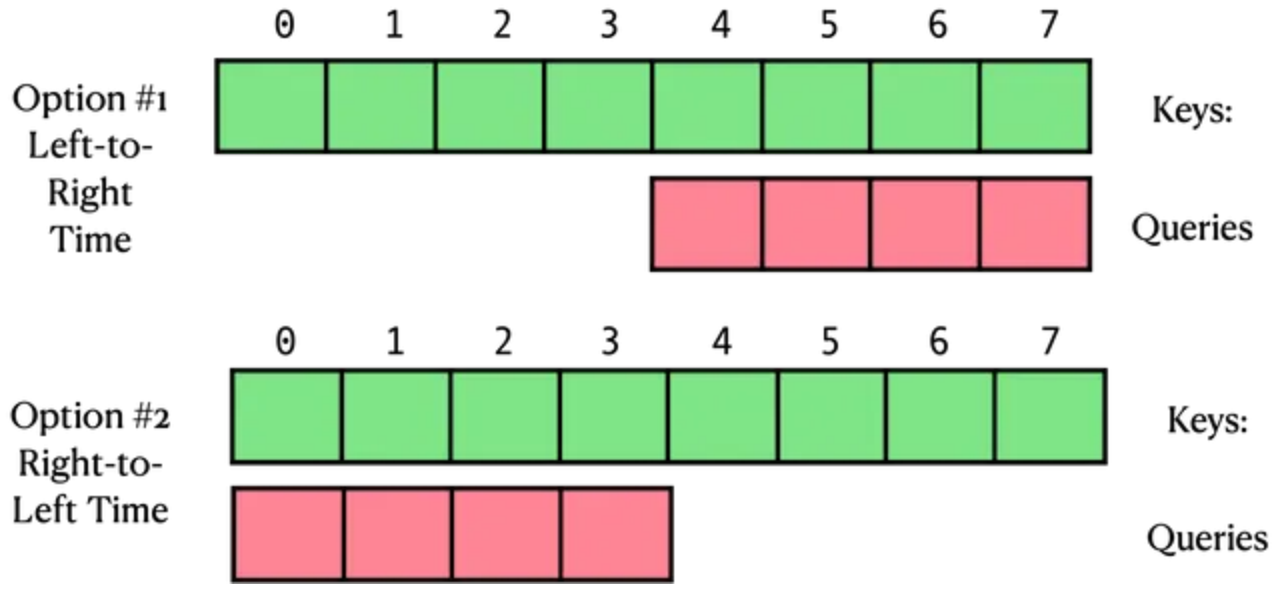
We adopt the language of Attention models and consider two sequences: the query, whose indices are the row indices of **R**, and the key, whose indices are the column indices of **R**. M_{ij} = q_i dot k_j. Without loss of generality we may also assume that the length of the query is less than or equal to that of the key.

As in our starter case, we need a fixed absolute positional encoding to work with. For this, we use the same fixed sinusoidal encoding discussed at length in Part I of this series. This is a matrix **A**, of shape (query_len, d_model), and can be represented as list of vectors **A** = (**a**_0, …, **a**_{n-1}), where **a**_i is the *i*th row of **A**, and of dimension d_model. Writing it this way, we see that the only difference with our simple starter example is that all the elements of **A** are now bold-faced (meaning vectors). This would *suggest* that we can immediately read off the matrix form of **R** as well:

$$\mathbf{R} = \begin{pmatrix} \mathbf{a}_0 & \mathbf{a}_1 & \cdots & \mathbf{a}_{n-1} \\ \mathbf{a}_1 & \mathbf{a}_0 & \cdots & \mathbf{a}_{n-2} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_{n-1} & \mathbf{a}_{n-2} & \cdots & \mathbf{a}_0 \end{pmatrix}$$

This is only true if the query length, m, is equal to the key length n!

However, this is only true if the query length, which we denote by *n,* is equal to the key length *m*. At this point you might be asking, "Why would the query length ever not equal the key length? And couldn't I just use padding to deal with such a case anyway?" To answer that question by example, precisely such a situation arises when implementing the Transformer XL, where one looks at "memory" states, that are further back in the past than the current query length. There are two options for comparing queries and keys of different length:



Source: current author. The integers denote sequence positions.

The first option moves time left-to-right and lines up the end positions of the two sequences. It ensures that the red queries only have access to green keys before it (plus masking on the overlapping parts). Likewise, the second

option lines up the starts of the sequences, and assumes time moves from right to left.

Note that if we were to extend the query length to match the key length, we would retrieve an (n, n) dimensional matrix, whose form is correctly given by the previous equation. Each of the two options are actually *subsets* of this larger matrix. In the first option, we compute the full (n, n) matrix, then remove the first *n-m* rows, leaving an (m, n) matrix. The second option is computed similarly, instead removing the final *n-m* columns. We will now assume that we are working with an (n, n) matrix, then truncate as necessary based on the required direction of time.

We are going to show how the tensor **R** may be computed in O(n) operations by a series of left and right shifts of the tensor **A**_i **1**_j = **R**_*{i,0}*, where the indices are meant to help you understand the tensorial structure. It's just a bunch of repeats of the one unique matrix **A**. The algorithm amounts to the following simple example: find an operation that performs:

$$\begin{pmatrix} 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \\ 0 & 1 & 2 & 3 \end{pmatrix} \xrightarrow{shifts} \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$$

Using right shifts we can correctly get the upper-right diagonal

I will refer to row numbers via zero-indexing. We notice that if we right-shift row one by 1, row two by 2, and row three by 3, then we correctly reproduce the upper-right diagonal. Now suppose we flip all of the positions $i \to -i$. This is equivalent to left-right flipping the left matrix.

$$\begin{pmatrix} 3 & 2 & 1 & 0 \\ 3 & 2 & 1 & 0 \\ 3 & 2 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{pmatrix} \xrightarrow{shifts} \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{pmatrix}$$

Using left shifts we can correctly get the upper-right diagonal

We could then perform the inverse procedure; left-shift row two by 1, row one by 2, then row zero by 3, and we would correctly reproduce the lower-left diagonal. Combining these two operations and masking the incorrect parts of each, we can successfully construct our tensor **R!**

Before coding up this algorithm, let's discuss what it is computing. The lower-left diagonal portion consists of exactly all elements with index $j \leq i$. In other words, the lower-left diagonal represents all key-query pairs where the query attends on keys that occur at *earlier* times. The upper-right diagonal consists of all the opposite, where the query attends on keys at *later* times. Hopefully this helps complement our earlier diagram discussing time flow.

For many neural networks, we not only ignore the upper-right diagonal, but actively mask those values. For attention models, this restriction is referred to as **masked attention**, and is associated with a **decoder** transformer. A decoded sentence cannot attend on future values that it has not yet decoded, so this should make some sense.

If we are implementing a decoder, which the Transformer-XL is, then we need compute only the lower-left diagonal positional logits; the upper-right components are garbage and will be masked anyway. However, their paper does not consider the idea of bi-directionality. Since we know how to compute relative positions for both left and right directions, it's wouldn't be too hard to create a bi-directional Transformer XL.

> One of my motivations for writing this post, was to bring up the possibility of using fixed, bi-directional relative positional encoding in Attention models.

It's also not too hard to compute the relative positional logits for *all* entries, thereby letting queries attend on past, present, and future keys. Bidirectional Transformers are quite useful, BERT being a famous example, and I'm curious to know if there would be any performance increase by using a bidirectional *fixed,* relative positional encoding. If a reader knows, please post a comment! I'm putting the emphasis on *fixed*, because as we'll see, you can incorporate learned bidirectional relative encodings pretty easily, and this is something people have experimented with.

Computing position logits: model examination and setup

It's going to be a lot easier to compute the positional logits if we can first describe what it is exactly we are computing. We start by writing out all of the contributions to a vanilla Attention layer with absolute positional encodings, then examine which contributions need to be "relative-ized". We'll use the following notation

$$\mathbf{E}_i \equiv \text{Embedding of word } i, \quad \mathbf{U}_i \equiv \text{Absolute pos. enc. of word } i$$
$$\mathbf{W}_{k/q,E} \equiv \text{Key/query embedding weight matrix.}$$
$$\mathbf{W}_{k/q,U} \equiv \text{Key/query position weight matrix.}$$

The W weights are the usual weights to transform our embeddings/positions into keys/queries. Per usual, we get an embedding E and positional encoding U, then sum them as the input. We now find for the Attention logits **M:**

$$
\begin{aligned}
M_{ij} \sim \mathbf{E}_i \mathbf{W}_{q,E} \mathbf{W}_{k,E}^T \mathbf{E}_j^T + \\
\mathbf{E}_i \mathbf{W}_{q,E} \mathbf{W}_{k,U}^T \mathbf{U}_j^T + \\
\mathbf{U}_i \mathbf{W}_{q,U} \mathbf{W}_{k,E}^T \mathbf{E}_j^T + \\
\mathbf{U}_i \mathbf{W}_{q,U} \mathbf{W}_{k,U}^T \mathbf{U}_j^T
\end{aligned}
$$

Bold-face indicates the embedding dimension d_model. i and j represent the positional dimension.

We use a similar symbol "~", since there's a normalization factor of sqrt(d_model) that is irrelevant to the current discussion. The terms on each row describe the various types of correlations the model may learn. We will have more to say about this in the next section. For now, we simply request that the matrix element M_{ij} only depends on positional information U_{i-j} = R_{i,j}. This is as simple as just replacing U_j → U_{i-j}.

Next, we observe that lines 2 and 3 are somewhat redundant. We keep line 2 and use it to determine the local correlations between E and U. We then adapt line 3 to use a shared (amongst all keys, aka repeated along each row) vector **u**_i instead of **U**_i **W**_{q,U}. This has the effect of maintaining a **global context vector**, indicating how embeddings are related to positions given all that we've seen in the past. You can think of this as the model being allowed to learn the topic of conversation. If money related words have appeared a lot lately, then this piece may direct the model to interpret the word "bank" as related to financial institutions and not a river bank.

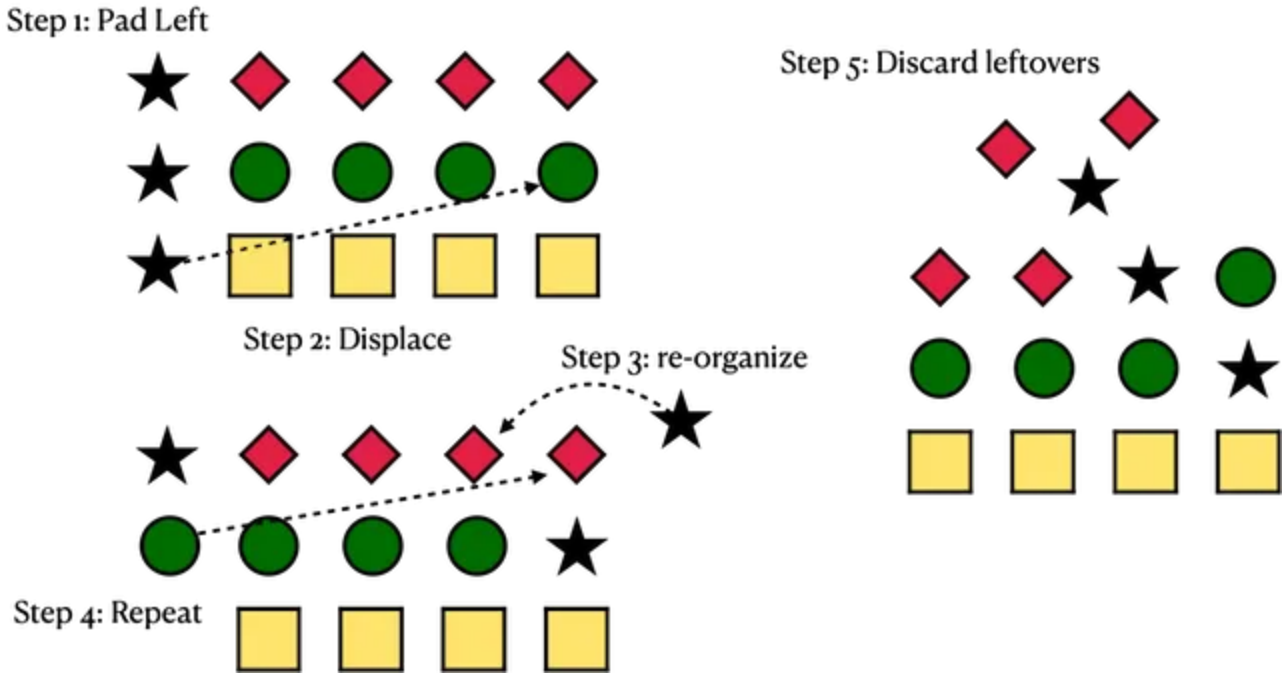
The last line contains position-position correlations. Normally, if these were learned parameters we expect these would just reproduce the fixed positional encodings we already provided. A better use of this term would be to treat it similarly to line 3. We replace v_i instead of **U**_i **W**_{q,U}, which has the role of a **global position vector**, keeping track of our global position. After these substitutions we have:

$$M_{ij} \sim (\mathbf{E}_i\mathbf{W}_{q,E} + \mathbf{u})\mathbf{W}_{k,E}^T\mathbf{E}_j^T + (\mathbf{E}_i\mathbf{W}_{q,E} + \mathbf{v})\mathbf{W}_{k,U}^T\mathbf{U}_{i-j}^T$$

The second term is the thing we are now looking to compute.

Code and algorithm

The algorithm is the same as before. Only now, we have a concrete thing to look at. For our purposes, we can group everything to the left of **U** into a single matrix **q_i**, such that the term we want to compute is **q_i U_{i-j}^T**. We begin with the left-shifts. It's easier to first show the algorithm in a diagram, then try to explain it



I thought it'd be fun to use shapes instead of numbers. The black stars represent padding. They are numerically equivalent to 0. Ultimately, we want to left-shift the 2nd to last row by one, the 3rd to last by two, etc. We begin with a greedy choice. Take the lower left pad token and move it onto the 2nd to last row. This automatically implements a left-shift by forcing all other items in that row to move left by one. However, this in turn moves the first element (a black star) in the 2nd to last row to have nowhere to go, and

also forces one of the green circles to also move since it's now outside the scope of the matrix. As a result, we must now move one black star and one green circle onto the red diamond row. This in turn displaces the red diamonds, leaving three pieces left with nowhere to go! But, we notice that the rest of the matrix is now properly left-shifted. If only we could get rid of those pesky remaining shapes...

We can. And pretty easily too. You just need to reshape the matrix into one that has 3 columns, and remove the first row. Done. If you reshape it back to its proper form, everything will be left shifted. For the following, we assume an input tensor of shape (N, T, Q, K). Here are the algorithm steps:

If time increases left-to-right:

1. Reverse the positions. This can be done as `tf.reverse(U, [2])` or for matrices as `np.flipud(U)` .
2. Perform the dot product **q**_i**U**_j^t.
3. Left pad the result with a column of ones. For a tensor of shape (N, T, Q, K), this means left pad on the fourth and final dimension K.
4. Reshape into size (N, T, K+1, Q).
5. Remove the first row.
6. Reshape into final answer of shape (N, T, Q, K).

If time increases right-to-left:

1. Do not reverse positions. If key length is greater than query length, remove the final (klen-qlen) rows of U. Perform the dot product as in step 2.

2. Right-pad the fourth and final position. Perform the same reshape
3. Truncate the final row instead.
4. If U was originally truncated, left pad as many columns as rows deleted. Reshape to final answer.

If time is bi-directional:

1. Compute a mask by taking the upper and lower diagonals of a (klen, klen) matrix of 1's.
2. Compute the left and right logits respectively, then combine with masks.

Below we present the code in NumPy. For the most part, this can be turned directly into TensorFlow by replacing np →tf, but we thought it would be clearer to avoid the complications of dealing with batching and multi-headed attention dimensions. np.triu and np.tril will convert to tf.linalg.band_part.

I added some additional tests so that you can confirm this does indeed properly compute the positional logits. When I ran the code on my machine I got the output

```
Correct answer:
 [[ 5.18 -2.58  1.54 -2.58  5.18]
 [ 0.18 -2.39  0.91 -1.    0.91]
 [-4.05  3.41  1.86 -0.6  -4.53]]
Shifted algorithm:
```

```
 [[ 5.18 -2.58  1.54 -2.58  5.18]
 [ 0.18 -2.39  0.91 -1.    0.91]
 [-4.05  3.41  1.86 -0.6  -4.53]]
Match? True
```

Section III (learnable encodings and comparison):

Learned embeddings

Up until now, (almost) everything we've done has involved user-defined fixed positional encodings (I say almost because we added those global context/position vectors in Section II, which were learned parameters). This is fine. If we can get good performance with that, it's faster and more stable than letting a model learn these parameters itself. However, if there's one thing deep learning continually reminds us; a computer can probably do it better.

Convolutions have been used for a long time in image processing, however, when building computer vision networks we let the computer decide what kinds of convolutions it will build. In practice, that works much much better. So why not take the same approach here? Why not let the computer learn its own positional encodings? In this section, we discuss some approaches along these lines.

Our starting point is the previous vanilla Transformer Attention logits equation, which we repeat here for convenience

$$\begin{aligned} M_{ij} \sim &\mathbf{E}_i \mathbf{W}_{q,E} \mathbf{W}_{k,E}^T \mathbf{E}_j^T + \\ &\mathbf{E}_i \mathbf{W}_{q,E} \mathbf{W}_{k,U}^T \mathbf{U}_j^T + \\ &\mathbf{U}_i \mathbf{W}_{q,U} \mathbf{W}_{k,E}^T \mathbf{E}_j^T + \\ &\mathbf{U}_i \mathbf{W}_{q,U} \mathbf{W}_{k,U}^T \mathbf{U}_j^T \end{aligned}$$

Bold-face indicates the embedding dimension d_model. i and j represent the positional dimension.

Though these 4 terms were derived as a consequence of adding positional encoding **U** to embedding vector **E,** we are by no means bound to construct logits in this way. Instead, this gives us a clue to the types of contributions to the logit matrices. Principally, there are three types

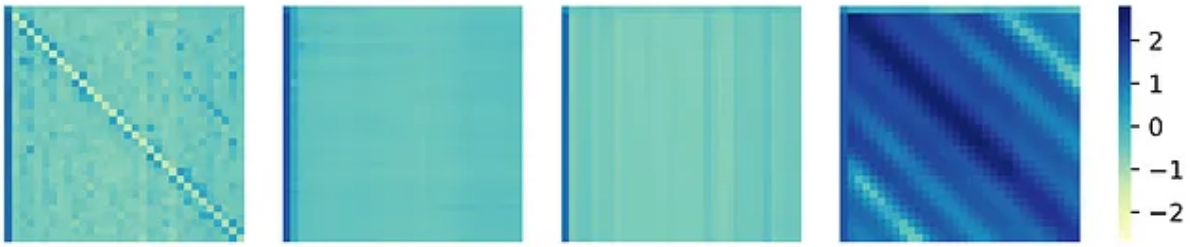
1. Embedding-embedding correlations. Given by line 1.
2. Embedding-position correlations. Given by lines 2 and 3.
3. Position-position correlations. Given by lines 4.

Embedding-embedding correlations are the foundation of attention. We're obviously not getting rid of them, so there's not much more to say about that.

Embedding-position correlations indicate if some words match up with an absolute key/query position. A priori, we would not expect that to be true. If I say the word hello, it shouldn't change meaning based on if today is Feb. 2nd or July 18th. Instead we often assign word meaning base on its relative position. This hints that we may want to replace these terms with something more "relative" friendly. Again, we've already shown how these can be upgraded with fixed relative encodings, but here we're thinking about learnable parameters to add.

Finally, in the last line we have position-to-position correlations. Hopefully, a model will learn to reproduce something like the Toeplitz matrices we've been dealing with manually. That would indicate that the model is learning meaningful positions.

Let's tackle embedding-positional correlations first. The easiest way to determine if these correlations are relevant is to just straight up measure them. In the paper by Guolin Ke, Di He & Tie-Yan Liu (2020), they analyze the learned matrix correlations of a pre-trained BERT model, and split plot the resulting learned correlations. They find the following diagrams,



Source: Guolin Ke, Di He & Tie-Yan Liu (2020). From left to right: E-E, E-U, U-E, U-U correlations.

confirming our intuition that we may drop absolute positional-embedding correlations.

> *As a side note, we remark that this conclusion is reached based on the assumption that key and query sizes are the same. It may be possible in a context like Transformer-XL, that there is* ***global*** *positional or contextual information that could be propagated in the network. In this case it might not be prudent to discard these contributions.*

These remarks lead us to a first guess for learnable positional encoding. We first decide to inject our positional encoding into the network **directly through positional logits.** We then choose the following form for those logits:

$$A_{ij} \sim \mathbf{E}_i \mathbf{W}^{(q)} \mathbf{W}^{(k),T} \mathbf{E}_j^T + \alpha_{i-j}$$

where, as suggested, we erased the positional-embedding stuff and turned the position-position piece into something learnable. This representation was one of the earliest attempts at extending positional encoding to logits, and was presented in the paper Shaw et. al. (2018). There is one slight difference between this and just blindly adding a learnable matrix to the logits. We have constrained the form of the positional logits, here denoted by alpha, to be Toeplitz. As a result, we have effectively halved the available parameters by tying weights. This ensures that the model must treat positions symmetrically; a distance d(x,y) must equal d(y,x). The actual model that Shaw et. al. (2018) implement is a little more complicated, but that's the general idea.

To compare to our fixed positional encoding method, this is not as flexible at dealing with long sequences. Since the positional logits are learned parameters, they cannot be adjusted after the model has trained, and are capped at whatever their size was after training. The authors recognized this issue, and proposed that to overcome this, they implement position clipping. Beyond a certain distance, the relative positions are clipped to maximum value of k.

$$\alpha_{ij}^{(clip)} = \alpha_{\min(|i-j|,k)}$$

The claim is that at long enough lengths, it's enough to know that something is simply "far" away, rather than the precise difference between large numbers. To improve upon the Shaw model, we would like to somehow incorporate more general positional encodings, that might allow for the propagation of global information. Furthermore, despite what we have been constantly saying throughout this article, there may be some use to absolute encodings. For example, a character based language model probably wants

to correlate capital letters very highly with the beginning of a sequence. However, this is only an artifact of a finite sequence length. For models with large sequences like the Reformer, this point becomes irrelevant.

There is another approach we can take, that compromises between retaining all positional encoding logit terms and retaining a restricted class with tied weights. In the paper of Guolin Ke, Di He & Tie-Yan Liu (2020), they propose using the logits

$$A_{ij} \sim \mathbf{E}_i \mathbf{W}^{(q)} \mathbf{W}^{(k),T} \mathbf{E}_j^T + \mathbf{U}_i \mathbf{W}^{(q)} \mathbf{W}^{(k),T} \mathbf{U}_j^T + \alpha_{i-j}$$

As they show, the final two terms do not have overlapping expressivity. This affords the model greater representational power. For example, a model may learn to use the absolute embeddings to determine an offset, then the relative embedding to fine tune the position.

One advantage that learnable positional encodings hold over fixed ones, is the flexibility to deal with non-sequential insertions into the data. For example, consider the BERT training objective. We set aside special tokens [CLS] and [SEP], to denote the sentence meaning and separation respectively. The latter have a well-defined positional meaning, but the former do not. A learned positional encoding can deal with this by 'untying' the [CLS] token. Ke et. al. (2020) treat this particular case by moving the [CLS] token to the zeroth sequential slot, and giving it its own shared parameter theta. This parameter is the same for all other tokens, indicating that the [CLS] token is equally "far" from all other tokens. Otherwise, there would be a bias to learn to give early sequence tokens more weight to [CLS].

Conclusion

In this article we went through the logical upgrade to absolute positional encodings: relative encodings. Beginning with some simple examples of day-to-day language, we motivated the need for using relative encodings in a model dealing with sequential data.

We showed that so long as the network has a reliable set of values for the distance between two points in a sequence, it can use that to compute relative positions. This allowed us to construct Toeplitz relative encoding tensors by recycling our already hard-earned knowledge about absolute positional encodings. Furthermore, due to the special Toeplitz properties of our encodings, we were able to find a way to compute relative position **logits** in O(n) time. We presented the code for doing so, and also discussed extensions to both right-moving sequences as well as bi-directional relative encodings.

Finally, we compared our fixed sinusoidal encodings to using learned parameters. The advantages of the fixed encodings are speed, sequence length flexibility, and a built in reliable model of quantifying position. The built-in encodings are also able to take advantage of models with different key-query sizes. They can also be tweaked into a hybrid form, where some learnable parameters are set aside for storing global context and positional information. In comparison, the learned parameter model has a simple representation, and also the freedom to let the model decide how to compare sequences. A unique advantage learnable parameters hold is in dealing with non-sequential information, like special tokens that do not/should not have a well-defined position. These can be treated by untying weights, and reserving special learnable parameters for the non-sequential tokens.

Thanks for sticking it out this far. As always feel free to comment or contact me with any questions or comments!

References

1. Vaswani, Ashish, et al. “Attention is all you need.” *arXiv preprint arXiv:1706.03762* (2017).
2. Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani. “Self-attention with relative position representations.” *arXiv preprint arXiv:1803.02155* (2018).
3. Dai, Zihang, et al. “Transformer-xl: Attentive language models beyond a fixed-length context.” *arXiv preprint arXiv:1901.02860* (2019).
4. Ke, Guolin, Di He, and Tie-Yan Liu. “Rethinking the Positional Encoding in Language Pre-training.” *arXiv preprint arXiv:2006.15595* (2020).
5. Raffel, Colin, et al. “Exploring the limits of transfer learning with a unified text-to-text transformer.” *arXiv preprint arXiv:1910.10683* (2019).

Attention NLP Machine Learning Transformers

Written by Jonathan Kernes

Following

423 Followers · Writer for Towards Data Science

I'm a physics PhD recently taken over by the allure of AI. When I'm not at the computer you might catch me listening to hip hop or watching basketball.

More from Jonathan Kernes and Towards Data Science

Jonathan Kernes in Towards Data Science

SentencePiece Tokenizer Demystified

An in depth dive into the inner workings of the SentencePiece tokenizer, why it's so powerfu...

18 min read · Feb 4, 2021

287 5

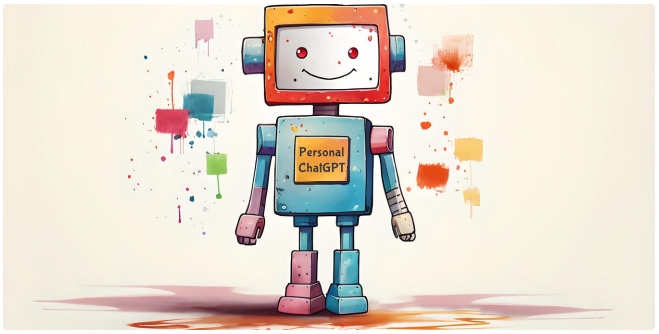
Antonis Makropoulos in Towards Data Science

How to Build a Multi-GPU System for Deep Learning in 2023

This story provides a guide on how to build a multi-GPU system for deep learning and...

10 min read · Sep 17

549 11



Robert A. Gonsalves in Towards Data Science

Your Own Personal ChatGPT

Jonathan Kernes in Towards Data Science

How you can fine-tune OpenAI's GPT-3.5 Turbo model to perform new tasks using you...

✨ · 15 min read · Sep 8

595 7

Locality Sensitive Hashing: How to Find Similar Items in a Large Set,...

We offer a guide to the art of locality sensitive hashing, with applications to document...

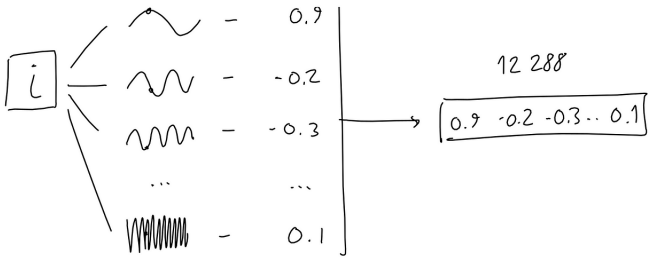
12 min read · Feb 4, 2021

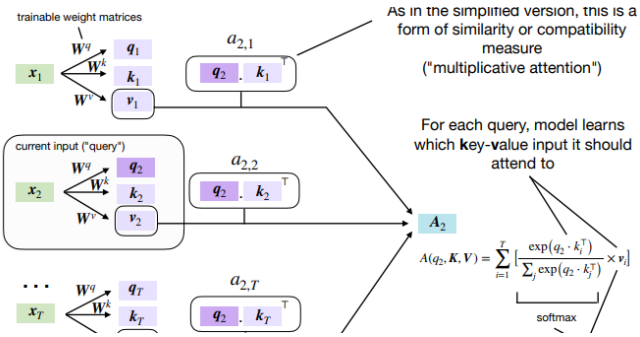
114 1

See all from Jonathan Kernes

See all from Towards Data Science

Recommended from Medium





Rand in AI Mind

Creating Sinusoidal Positional Embedding from Scratch in...

Recent days, I have set out on a journey to build a GPT model from scratch in PyTorch....

6 min read · Jun 28

103

Zain ul Abideen

Attention Is All You Need: The Core Idea of the Transformer

An overview of the Transformer model and its key components.

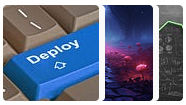
6 min read · Jun 26

144

Lists

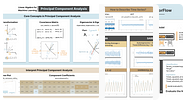
Natural Language Processing

669 stories · 283 saves



Predictive Modeling w/ Python

20 stories · 452 saves

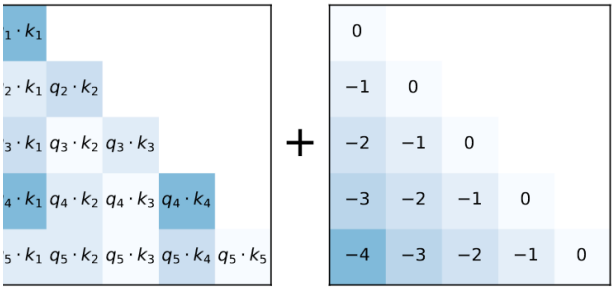


Practical Guides to Machine Learning

10 stories · 519 saves

The New Chatbots: ChatGPT, Bard, and Beyond

13 stories · 133 saves



Amy Pajak

ALiBi: Attention with Linear Biases

Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation

12 min read · Jul 3

8

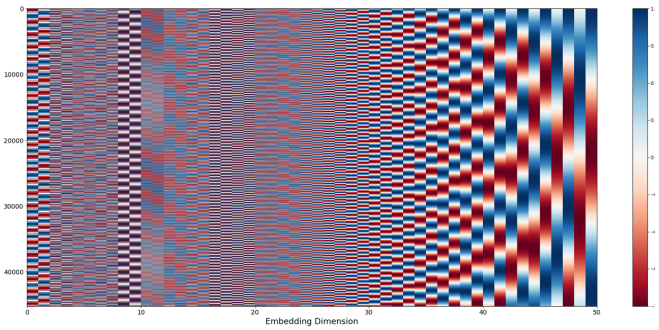
Dixn Jakindah

Top P, Temperature and Other Parameters

Large Language Models(LLMs) are essential tools in natural language processing (NLP)...

3 min read · May 18

7



Eugene Ku

Transformer Architecture (Part 1—Positional Encoding)

Nowadays, arguably the most popular and influential model behind the hypes of deep...

4 min read · Aug 22

15

Galina Alperovich in GoPenAI

The Secret Sauce behind 100K context window in LLMs: all tricks...

tldr; techniques to speed up training and inference of LLMs to use large context...

16 min read · May 16

1.4K 4

See more recommendations