HANDS-ON TUTORIALS

Master Positional Encoding: Part I

We present a “derivation” of the fixed positional encoding that powers Transformers, helping you get a full intuitive understanding.

Jonathan Kernes · Follow
Published in Towards Data Science
17 min read · Feb 15, 2021

Listen Share

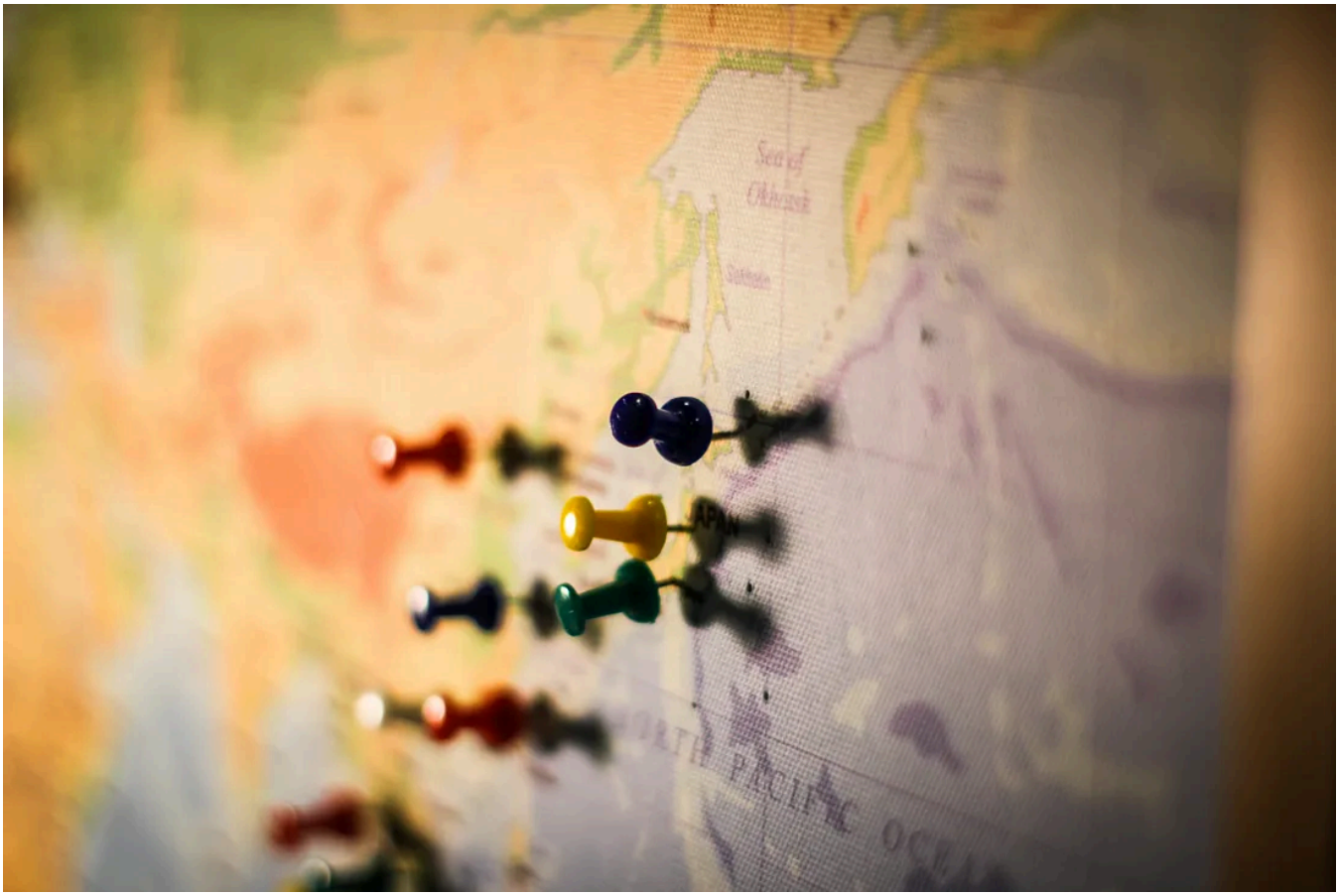


Photo by T.H. Chia on Unsplash

This is Part I of two posts on positional encoding (UPDATE: Part II is now available here!):

- **Part I:** the intuition and “derivation” of the fixed sinusoidal positional encoding.

- **Part II:** how do we, and how *should* we actually inject positional information into an attention model (or any other model that may need a positional embedding). This focuses on relative encodings.

As far as I know, there are some original insights in here regarding positional encoding viewed as a manifold. But, the world is a big place, so I doubt I'm the first to notice this; if anyone knows a good link, please let me know!

Part I: the fixed sinusoidal positional encoding

Introduction

Imagine you've finally managed to work your way through understanding transformers. You've drawn those fancy colored matrices mapping keys to queries, and convinced yourself of the meaning of the logits tensor. You've even gone a step further and figured out how multi-head attention works, admiring the cleverness and efficiency that a little tensor reshaping can provide. You've learned how to mask your input sequences by directly modifying logits and how to gracefully transfer sequences from an encoder to a decoder. You're so satisfied with yourself you plop down on your couch, throw your feet up and treat yourself to a big chocolate chip cookie… No. Two chocolate chip cookies. You're worth it.

Contented with your newfound knowledge, you finally turn to the embedding layer. "It's easy," you muse to yourself, "I've been turning words into vectors since the backstreet boys were back." As you turn your eyes towards the bottom of the model architecture diagram, something unfamiliar catches your gaze. The **positional encoding** layer. You immediately stop eating your cookie, crumbs falling to the floor, as you suddenly remember. *Positional information doesn't show up anywhere.*

Not yet panicking, you read ahead hoping you'll find see an easy solution. Instead all you get are a few sentences telling you about some fixed sinusoid function and the directions that when referring to the usual embedding layer "…the two can be summed." Furiously turning the page back and forth, you realize there's nothing else. Having taken a physics class once before, you know what just happened. You've been "it's trival"'ed.

Positional encoding is basically the "it's trivial" of attention models. One of many dreaded 2–3 word phrases like "it's fine" or "it doesn't matter", where the intent is most definitely the opposite of what you're saying. It's *not* fine, it *does* matter, and positional encoding is *not* trivial.

Now, that we all acknowledge the situation, let's actually get into what positional encoding is. We are going to "derive" the answer by taking a physicist's approach. Come up with a few simple theories/guesses for how to accomplish our task, then slowly improve upon them until we get an encoding that mostly satisfies what we want. We will put off where and how to pop in the positional encoding layer to Part II. For now, we are just trying to figure out how to give a sequence some positional meaning.

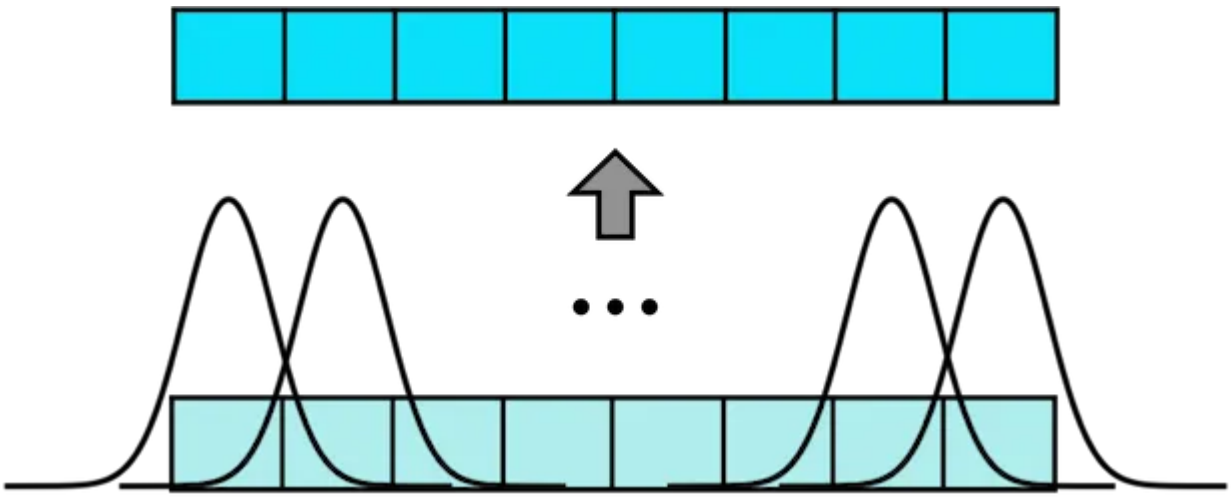
Discrete positional encoding

The most difficult part of this might just be trying to describe what it is exactly that we're trying to do. **If this subsection doesn't make much sense, just directly skip to the next one.** Abstractly, we have the following goal:

> *A positional encoding is a finite dimensional representation of the location or "position" of items in a sequence. Given some sequence A = [a_0, …, a_{n-1}], the positional encoding must be some type of tensor that we can feed to a model to tell it where some value a_i is in the sequence A.*

Admittedly, this is vague; this is because the "feed to a model" line is also vague. To fix this, let's assume we want to train a model to do the following task:

> *Given a sequence A of length T, output another sequence A' of length T where the output for each index is the weighted sum of it's neighbors.*



Source: the current author. The bottom layer is our input encoding vector. The overlaid Gaussians show the "model" we feed this into, producing an output vector of the same size. Each dark blue square is the weighted Gaussian average of its lighter counterpart.

This model is meaningless. I repeat. This model is meaningless. It's just to give us an idea of an example of one way you could feed a positional encoding to a neural network.

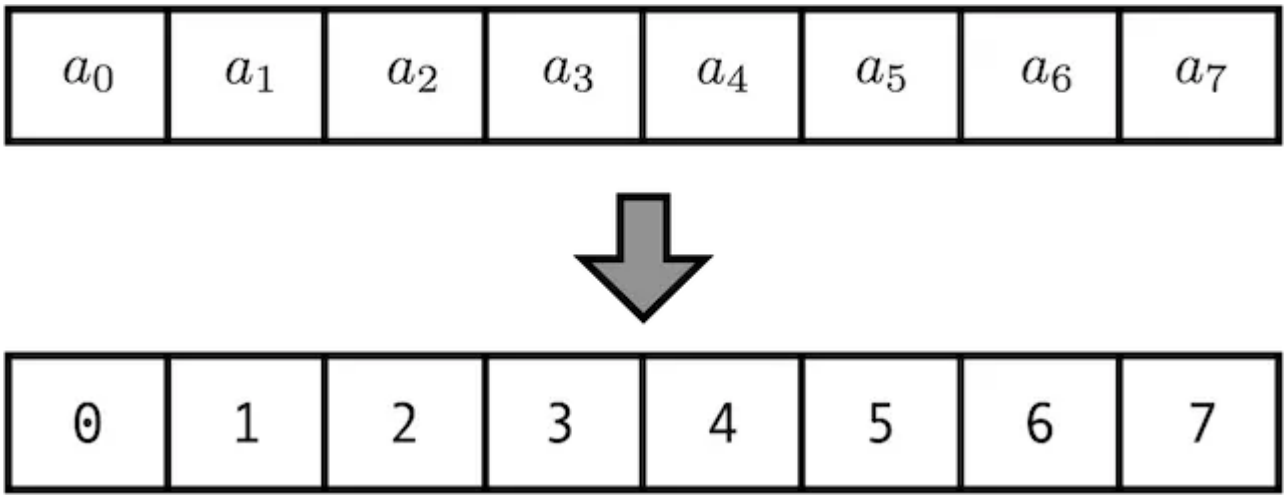
We are looking for a positional encoding *tensor* not a positional encoding *network*. This means that that our positional encoding tensor should be a representative of the position at each index, and our network will take care of whatever operations need to be done to that.

This encoding is going to have the same dimension as the sequence length, which could be very long. Also, the sequence length is not even a fixed quantity. Feeding a variable length vector into a model is a big problem. We assume that there is some maximum value called the **max_position** for which our model just can't handle any longer sequences. For all of our solutions we fix our encoding to be this long, and truncate down as needed for inputs.

Hopefully everything will be clear with an example.

Guess #1: just count.

Let's start with the simplest positional encoding we can possible create. We just count! We map every every element $a\_i \rightarrow i$, yielding the following encoding:



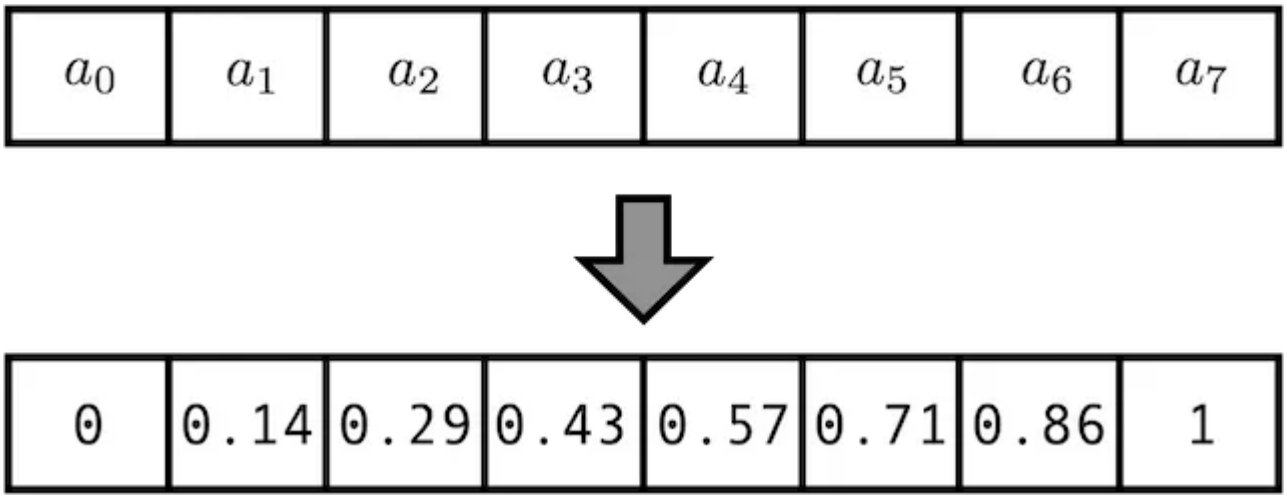
Source: current author. Sequence_length = 8. We simply create a positional encoding where each entry is its index number.

Pretty basic, we just created a new vector where every entry is its index number. This is an **absolute positional encoding.** Every position in the sequence is labeled by its position relative to a global origin of coordinates, assumed to be the start of the current sequence. We will continue to work with absolute encodings, as they can be used to derive **relative encodings**.

Now let's be critical. Here's what's wrong with our initial guess: the scale of these numbers is huge. If we have a sequence of 500 tokens, we'll end up with a 500 in our vector. In general, neural nets like their weights to hover around zero, and usually be equally balanced positive and negative. If not, you open yourself up to all sorts of problems, like exploding gradients and unstable training.

Guess #2: normalize the “just count” guess

This seems like the obvious solution → Just divide everything by the largest integer so all of the values are in [0,1]. This would give:



Source: current author. We normalize each entry by dividing by the sequence_length (which is 8 here).

But oh no! we’ve introduced another problem. We can no longer deal with arbitrary sequence length. That’s because each of these entries is divided by the sequence length. A positional encoding value of say 0.8 means a totally different thing to a sequence of length 5 than it does for one of length 20. (For length 5, 0.8=4/5 meaning it would be the 4th element. For sequence length 20, 0.8=16/20 means 0.8 represents the 16th element!). **Naively normalizing doesn’t work with variable sequence lengths**.

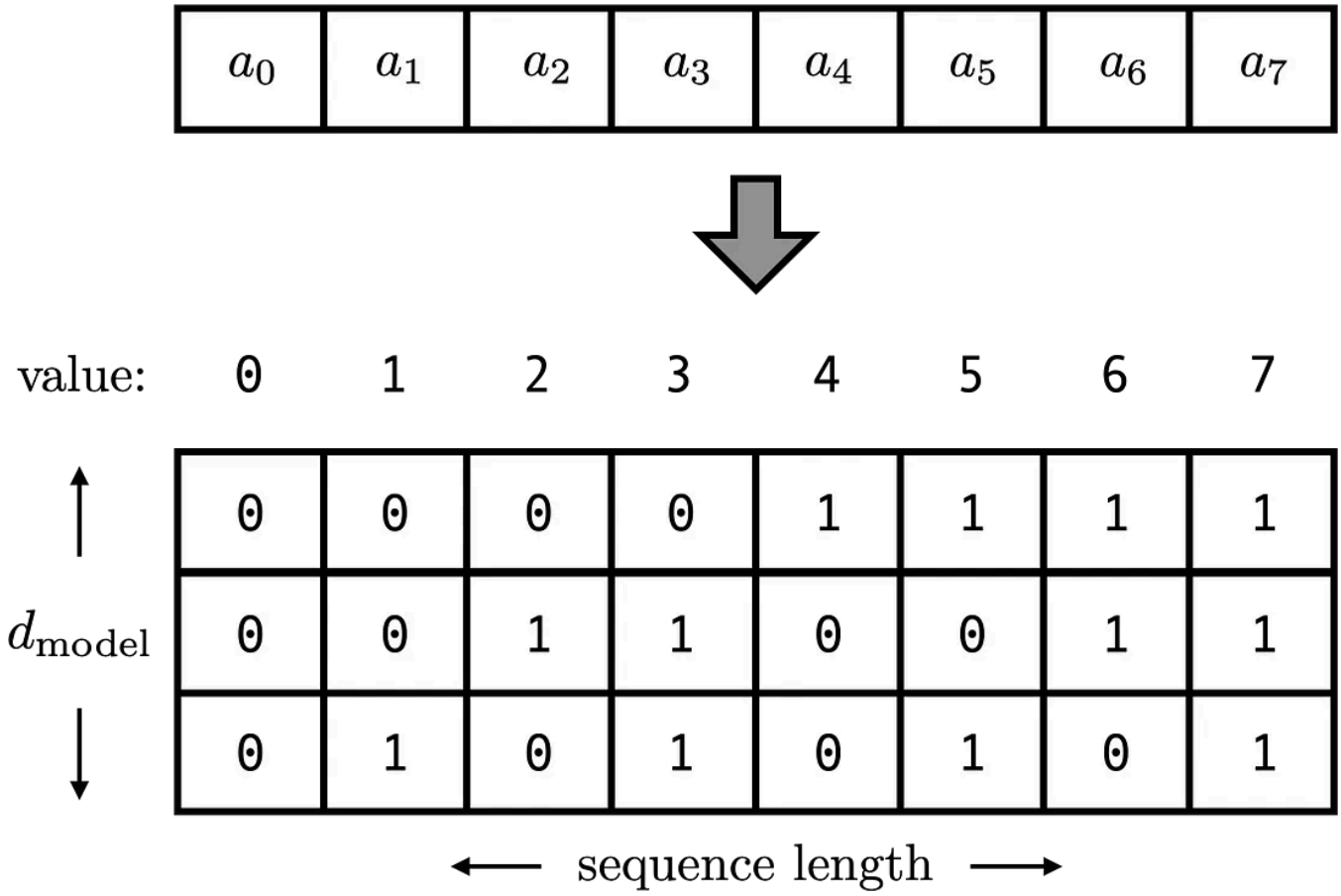
What we need to be able to do, is find a way to count without ever using numbers greater than 1. That’s a big clue, and you may be able to guess the answer.

Use binary numbers.

Guess #3: just count! but using binary instead of decimal

Instead of writing say 35 for the 35th element, we could instead represent it via its binary form 100011. Cool, everything is less than one. But, in our excitement, we forgot that the number 35 is the same whether we use binary or decimal; it’s always just 35. We didn’t gain anything… Instead, we actually need to do two things: 1) convert our integer to binary, and 2) convert our scalar into a vector.

Whoa, scalar to a vector? What does that mean? It means that our positional encoding *vector* now becomes a positional encoding *matrix*. Each number gets its own **binary vector** as opposed to just an integer. This is a pretty big conceptual shift, but by increasing the dimensionality, we are able to both keep arbitrary long sequences, while also restricting numbers to the range [0,1]! Our encoding now looks like this:



Source: current author. The matrix version of positional encoding. Each binary vector's entries represent a scalar in binary. e.g. index 5 holds 100 in binary, which is the value 4.

We have some freedom. We can choose the dimension of this new embedding space that holds the binary vector to be whatever we like. The only thing this dimension does is tell us how big our sequences can be, 2^dim to be precise. For simplicity, let's choose it to be equal to *d_model*, the embedding dimension that our model will use. To make sure we understand the vector representation, the 35th position would be represented as the binary vector

35 ← → [0, …, 0, 1, 0, 0, 0, 1, 1]

Analysis time. What are some problems with our current guess?

1. We still haven't fully normalized. Remember we want things to be positive and negative roughly equally. This is easily fixed: just rescale [0,1] ->[-1,1] via f(x) = 2x-1.
2. Our binary vectors come from a discrete function, and not a discretization of a continuous function.

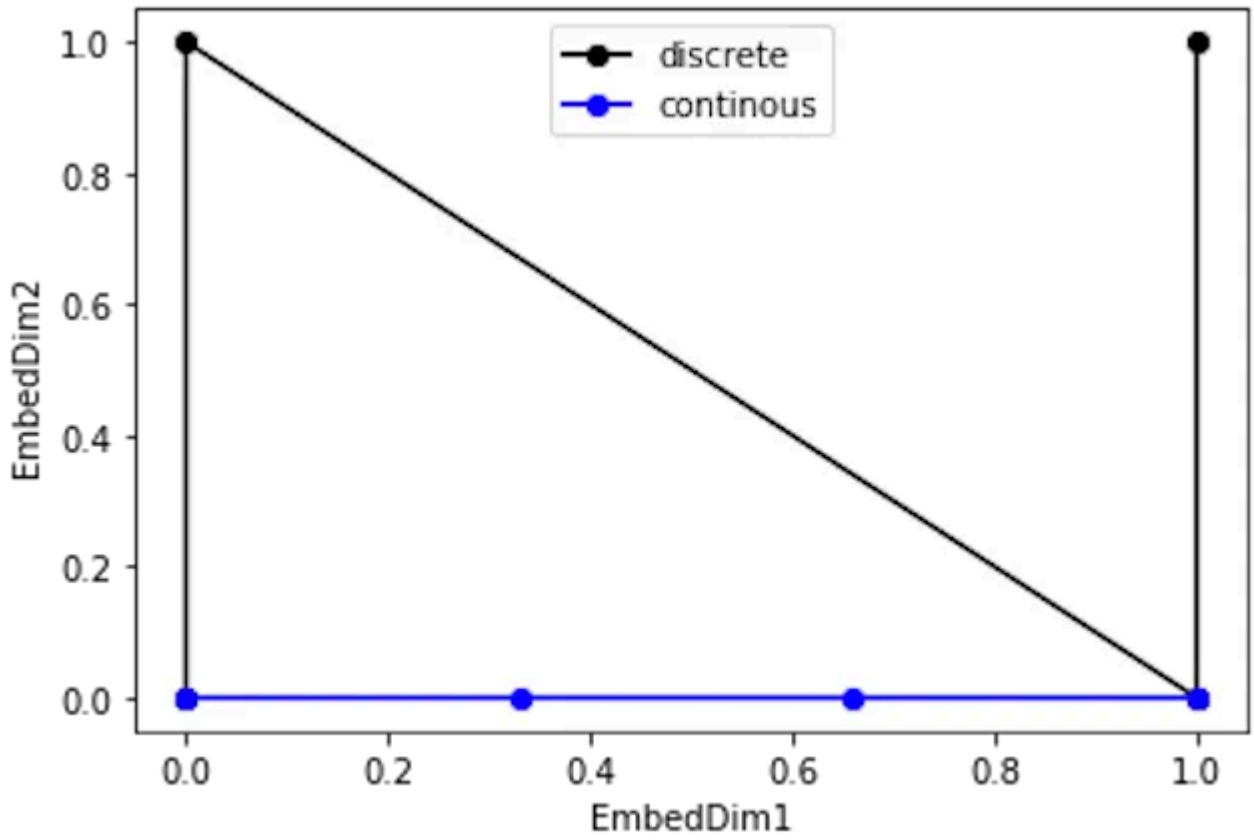
The latter point is subtle, and it has to do with using a *vector* to represent a *scalar* position. Let's compare two encodings, that try to measure a distance from 0 →3. The first encoding will be really simple: only use the x-axis dimension and put the position there, represented by a variable x in [0,3]

Continuous encoding: [x, 0]

Now let’s compare it to our binary vector discrete encoding. Since this is discrete, we can only represent 4 positions: 0, 1, 2, and 3. These give 4 points:

Binary vector encoding: [0,0], [0,1], [1,0], [1,1]

Any time we discretize something, we have inclination to believe that that something can be **interpolated,** meaning that we can construct a continuous function by connecting the dots. Take a look at the two interpolations for our two encodings:



Source: current author. The space curve showing flow of position. The discrete curve (black) runs from (0,0)->(1,0)->(0,1)->(1,1), and is much more jagged than the smooth continuous curve (blue).

You can see that the continuous encoding curve is smooth, and it’s easy to estimate intermediate positions. For the binary vector…that thing is a mess. This will lead us to our next hint: **find a way to make the binary vector a discretization of something continuous.**

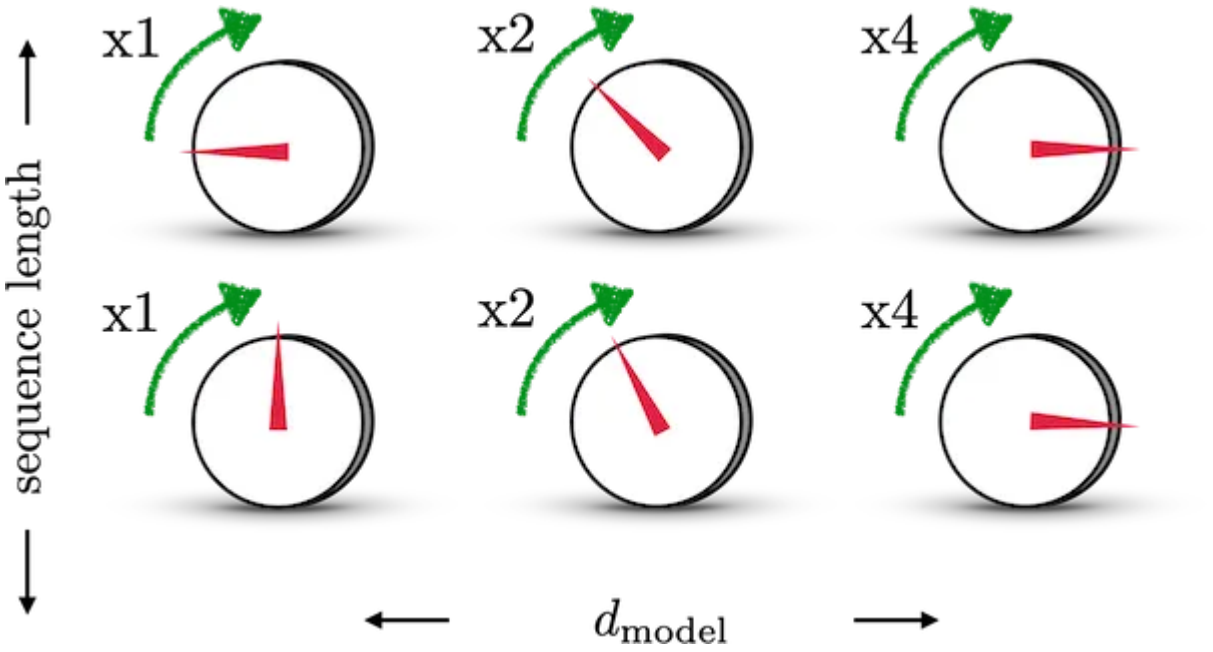
So with pictures, we want a function that connects the dots in a smooth way that looks natural. For anyone who has studied geometry, what we are really doing is finding an embedding manifold.

Yes, if you use positional encoding, you can sound super fancy and tell people that you're using manifolds. What this means in human terms, is that we are looking for a curve in the *d_model* embedding space, that as you walk along it slowly increases your "position" in a continuous way. Doing this properly brings us to our next guess.

Guess #4: use a continuous binary vector

To make our binary vector continuous, we need a function that interpolates a back and forth cycle $0 \to 1 \to 0 \to 1 \to 0\ldots$ etc. Cycles you say? Time to bring in trig functions. Let's use a sine function. Also, a sine function exists on [-1,1] so it's also properly normalized, which is an added bonus!

Here's the image/analogy you should now have. Every matrix entry in the positional encoding matrix represents a "dial" like on a stereo. Typically your dial might do something like adjust the music volume (volume is analogous to position). Like normal dials, the metaphorical dials we use in the encoding matrix can continuously turn from off (0) to on (1). To get a precise volume (position), we use a sequence (vector) of dials with different sensitivity. The first dial will adjust the volume ever so slightly, perhaps by 1 unit, a barely audible difference. The second will be more powerful, adjusting volume by 2 units. The third will adjust by 4 units, the fourth by 8 units, etc.



Source: current author. Each position in the sequence (column) is represented by a positional embedding vector (row), that be visualized a setting of a bunch of dials. Dials are depict matrix elements. As dimension increases, dials become more sensitive, and increase position more.

In this way, you can get a precise volume *over vast scales* since the dial strength increases exponentially. Instead of building one massive dial that contains say 512 sound levels, we can

build 8 tiny dials with just as much precision (2^8=256 if you're wondering why 8 dials). Super efficient huh!

NOTATION WARNING: we transposed the positional encoding matrix relative to the previous section. We now have dim M = (sequence_length, d_model), which is the usual way this is written.

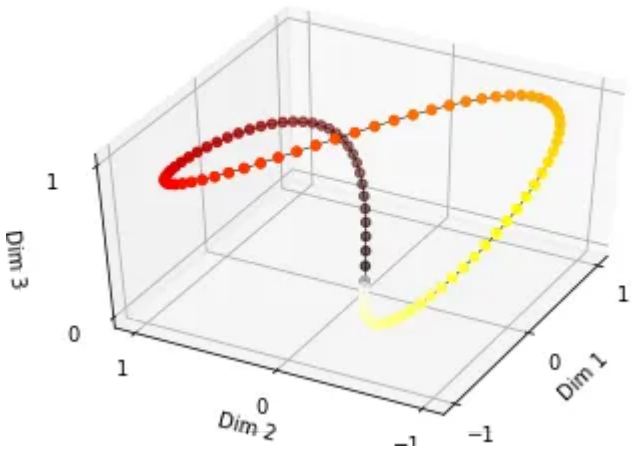
We now want our sine functions to undergo one rotation at precisely the right moments. For the first entry, it should shift $0 \to 1$ and back every time we move one step in the sequence. This means it needs a frequency of \pi/2 for the first dial, \pi/4 for the second, \pi/8 for the third, and on and on…

We now have our first prototype positional encoding tensor. It consists of a **matrix denoted by M**, where the y-axis of the matrix is the discrete position *x_i* in the sequence (0,1, …, n-1), and the x-axis is the vector embedding dimension. Elements of the matrix **M** are given by:

$$M_{ij} = \sin(2\pi i/2^j) = \sin(x_i \omega_j)$$

Here, omega is the dial frequency, which is monotonically decreasing with respect to embedding dimension. x_i is an integer giving the sequence position.

Let's now compare our continuous-discretized encoding vs. just that plain old discretized encoding.



Source: current author. The colors indicate the absolute position that that coordinate represents. Frequencies are omega=pi/2, pi/4, pi/8 for dimensions 1, 2 and 3 respectively.

Look, it's all continuous-y and manifold-y.

Analysis time. We are very close to the final answer. There are two things we still want to fix.

First, there is a discontinuity problem. The above plot is a smooth curve like we wanted. However, it is also a closed curve. Consider moving from position *n*, to *n+1*. This would be the part of the curve that transitions yellow $\rightarrow$ brown. By construction, the position *n+1* is equivalent to position 1. This was necessary to make our curve continuous, but it's also not what we want! *n+1* and 1 should be far apart, not close together. For anyone who has studied finite Fourier transforms, this problem should be familiar. Only half of the frequencies are unique.

To fix this, we observe that there's no reason that we *have to* have our embedding vector line up with a binary vector. In fact, the frequencies (omega_0, omega_1, …, omega_d_model) can be anything, so long as they are *monotonically increasing*, which just means they increase, and only increase. We can lower all of the frequencies, and in this way keep ourselves far away from the boundary.

$$M_{ij} = \sin(x_i \omega_0^{j/d_{\text{model}}})$$

The slowed down dial version. omega_0 is the min frequency. Check the edge cases: j=0 gives the largest frequency, j=d_model gives the smallest.

The original authors choose a minimum frequency of 1/10,000 for this purpose. they then increase frequencies exponentially via omega_min^{range(d)/d} to achieve the monotonicity criteria. Why choose 10,000? I truly have no idea, I'm guessing trial and error made this work the best.

The second problem isn't a problem insomuch as a shortcoming. There is no simple operation that allows us to add or subtract, say, 5 positional units from a position vector. Such an operation, known as translation, would allow us to write the positional encoding vector of index *i* as a function of the vector for index *j*.

Why is this a very desirable property to have? Imagine we have a network that is trying to translate the sentence "I am going to eat." The combination "is/am/are" +"going"+"to" +"verb" is a very common grammatical structure, with a fixed positional structure. "going" always ends up at index 1, "to" at index 2, etc. In this case, when translating "verb", we may want the network to learn to pay attention to the noun "I" that occurs before "am going to". "I" is located 4 position units to the left of "verb". Since our attention layer uses linear transformations to form the keys, queries, and values, it would be nice if we have positional encodings such that the linear transformation can translate the position vector located 4 units to the left, so that it lines up with the position vector of "verb". The query and key would then match up perfectly.

In summary, we want to see if we can modify our encoding so that it can be translated via linear transformation.

Final guess, #5: Use both sine and cosine to make translation easy.

The math is going to seem a little heavy, but it's not complicated. We just need to keep track of notation. First, let's summarize our current positional encoding matrix

$$\mathrm{PE} = \begin{pmatrix} \mathbf{v}^{(0)} \\ \vdots \\ \mathbf{v}^{(\text{seq len}-1)} \end{pmatrix}$$

The row-vector **v** is a vector of sines evaluated at a single position *x*, but with varying frequencies

$$\mathbf{v}^{(i)} = [\sin(\omega_0 x_i), \cdots, \sin(\omega_{n-1} x_i)]$$

Each row-vector represents the positional encoding vector of a single, discrete position. The positional encoding matrix **PE,** is a vector of these vectors, and so it is a (seq_len, n) dimensional matrix (seq_len is the sequence length). We now want to find a linear transformation **T(dx)** such that the following equation holds:

$$\mathrm{PE}(x + \Delta x) = \mathrm{PE}(x) \cdot \mathbf{T}(\Delta x)$$

How will we do this? First, notice that since all of the elements of **PE** are sines, the positions *x* are actually *angles*. From trigonometry, we know that any operation T that shifts the argument of a trig function must be some kind of rotation. Rotations can famously be applied by applying a linear transformation to a (cosine, sine) pair. Using the standard results for rotation matrices, we can make use of the following identity:

$$\begin{pmatrix} \cos(\theta + \phi) \\ \sin(\theta + \phi) \end{pmatrix} = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix} \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix}$$

to build a rotation.

> *Note, this formula has the linear operator acting from the left. In our network, linear transformations are conventionally applied from the right as we have so far shown. It's*

> *conceptually easier to do the left operation so we will write that answer here, but keep in mind that you will need to take the transpose of **T** to apply it to our current **PE** matrix.*

To build an encoding that supports translations via linear operators, we do the following. Create a duplicate encoding matrix using cosine instead of sine. Now, build a new positional encoding matrix as follows: alternating between the cosine and sine matrices, pop out the first column and append it to the final **PE** matrix. This is equivalent to replacing every sin(…) with a [cos(…) sin(…)] pair. In our new **PE** matrix, the row-vectors look like the following:

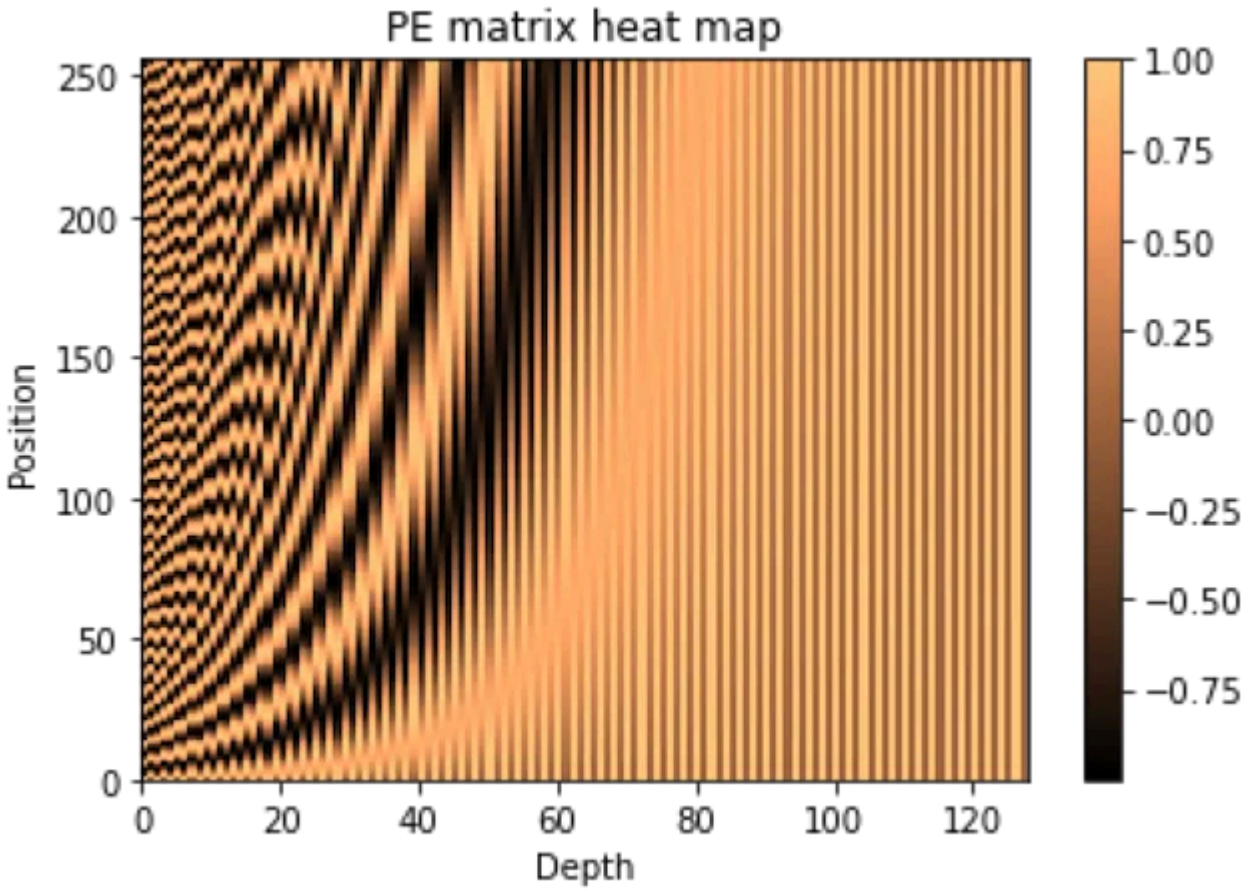
$$\mathbf{v}^{(i)} = [\cos(\omega_0 x_i), \sin(\omega_0 x_i), \ldots, \cos(\omega_{n-1} x_i), \sin(\omega_{n-1} x_i)]$$

We now build the full linear transformation by using a bunch of block-diagonal linear transformations. Each block will have a different matrix, since the frequencies that the block acts on are different. For example, in order to translate the *k*th dial with frequency omega_k by dx units, we would need a total angle shift of delta=omega_k*dx. The **T** matrix can now be written as:

$$\mathbf{T}(\Delta x) = \begin{pmatrix} \begin{bmatrix} \cos(\omega_0 \Delta x) & -\sin(\omega_0 \Delta x) \\ \sin(\omega_0 \Delta x) & \cos(\omega_0 \Delta x) \end{bmatrix} & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \begin{bmatrix} \cos(\omega_{n-1} \Delta x) & -\sin(\omega_{n-1} \Delta x) \\ \sin(\omega_{n-1} \Delta x) & \cos(\omega_{n-1} \Delta x) \end{bmatrix} \end{pmatrix}$$

If you take the transpose of this, you can directly insert it into our previous PE(x+dx)=PE(x)*T equation, thereby proving by construction the existence of a translation matrix!

That wraps up mostly everything. Let’s now take a look at what our PE matrix actually looks like.



Source: current author.

Here’s how to interpret the above PE matrix heat map.

1. **Most of the PE matrix is not needed for encoding.** Parts of the plot that are darker indicate dials that have been “turned on”, meaning that the sines and cosines in that region have values far away from their initial values of 0 and 1 respectively. This indicates how much of the embedding space is being used to store positional information. As you can see by following the black curve-ish thing, activating a dial one step deeper along depth becomes exponentially more difficult (the black curve looks like e^x).

2. **Vertical lines at greater depth vary less than loss at lower depth.** Pick a fixed depth, then move upwards noting how the color shifts light →dark →light… The frequency of this cycle *decreases* with depth, showing our intuition earlier that the dials at greater depth are more sensitive.

Final answer. No more guesses

At this point we’ve done everything we need to do, so let’s summarize:

1. Positional encoding is represented by a matrix. For sequence length T and model depth D, this would be a (T, D) tensor.

2. The “positions” are just the indices in the sequence.

3. Each row of the PE matrix is a vector that represents the interpolated position of the discrete value associated with the row index

4. The row-vector is an alternating series of sines and cosines, with frequencies that decrease according to a geometric series

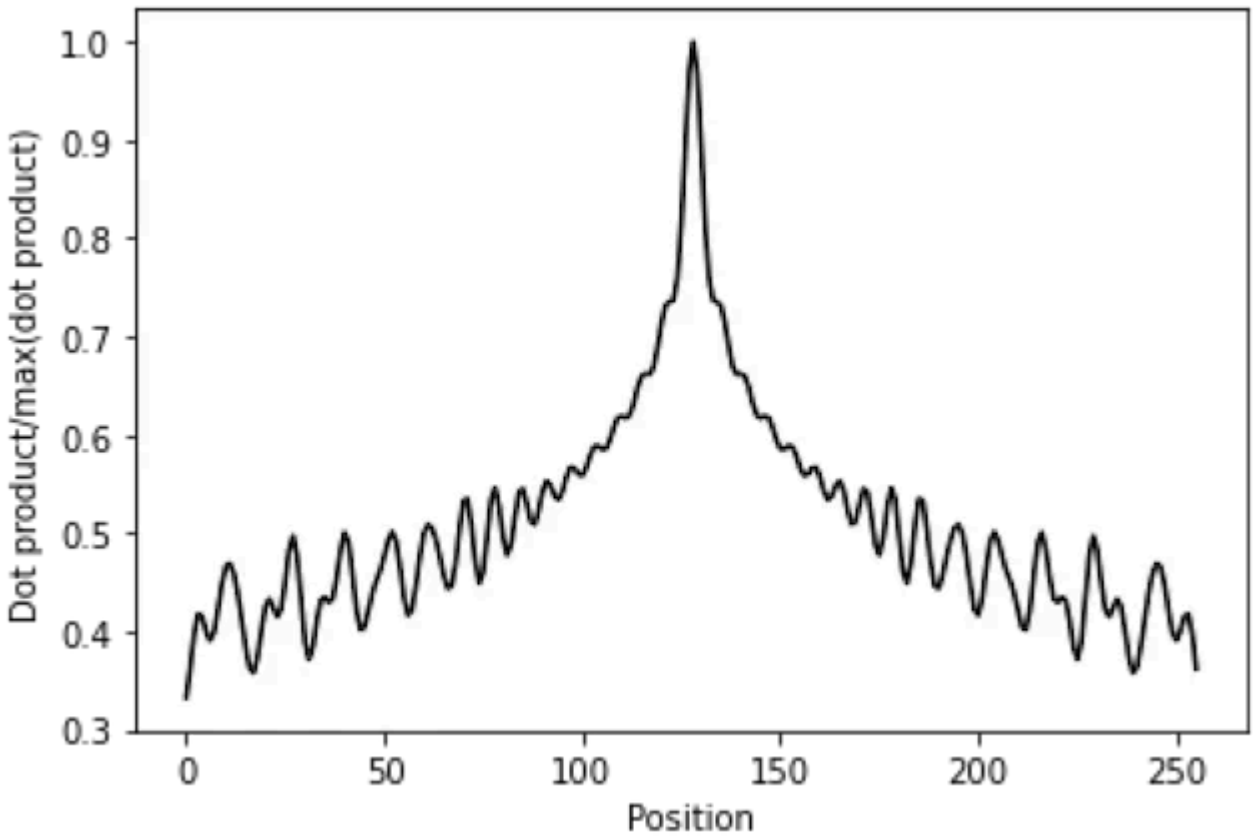
5. There exists a matrix multiplication that can shift the position of any row-vector we want.

Here are two ways to create a positional encoding matrix: one in numpy and one using only TensorFlow operators. Since the positional encoding matrix can be initialized at the same time as the layer that uses it, this formulation isn't of much use. But it's good practice working with tensors.

A couple of notes to understand the code. First we make use of numpy broadcasting to create the PE matrix angles via an outer product of positions and frequencies. Second, we directly apply sine and cosine to slices of the angle matrix, using subscripts, which is a nice trick.

Bonus property

There is an additional property that we would like our positional encoding to satisfy, and it is related to how positional encodings are used in an actual attention model. Attention is predicated on the fact that each embedding associated with each position in our sequence has some representative key or query. These keys and queries allow us to quickly determine is the current position should "attend" on another position, by performing the dot product query_i@key_j. If the dot product is large, then it implies the key at *j* matches the query at *i*. In order for this to work with positional encoding vectors, we would like that positions close to each other return large key-query dot products, while those far away return small ones. Take a look at the figure below:



Source: current author. We show the dot product of vector v^{(128)} with all other positional vectors for a PE matrix with parameters d_embed=128, max_position=256. Dot products are normalized by the max value.

This figure shows the dot product between a particular positional encoding vector representing the 128th position, with every other positional encoding vector. Notice, that without at all planning ahead for this, we get exactly what we wanted. The dot product is

maximal when comparing v^{(128)} to itself, and steadily decreases as we get further away. A nice bonus feature!

The reasons for this can be traced back to the orthogonality of sines and cosines. As we get further away, the frequencies between positional encoding vectors are no longer correlated, and the summation of random sines and cosines is known to tend to zero.

Conclusion

Thanks for sticking with it until the end. Like all things in academia, the things that are explained the least tend to be the most difficult. Ironic, but oh well.

We learned that positional encoding is a means of translating the location of objects in a sequence into information that a neural network (or other model) can understand and use. First trying a naive approach where we just give the model integers corresponding to position, we learned that this awful for a number of reasons: the numbers can be too large, there's not an obvious way to insert this information, it doesn't extend well to sequences of varying length.

We next made a series of guesses, acting like physicists trying to produce a working theory. By continually adjusting and altering our guesses to incorporate more desired characteristics, we eventually landed on sinusoidal positional encoding matrix. The matrix uses vectors to represent sequence positions. It can be properly scaled, allows for translations, and deal with varying sequence lengths easily.

The details and intuition behind implementation will be (hopefully) explored in a future article. For example, in that article, we ask questions like "why do we just add positional encodings to regular sequence embeddings?" Here, it seems like a decent unsubstantiated answer is that the PE matrix is sparse. Only a small proportion is used, so it seems like the model may treat the addition more like a concatenation, reserving some part of the embedding for position, and the rest for tokens.

References

1. Vaswani, Ashish, et al. "Attention is all you need." *arXiv preprint arXiv:1706.03762*