Sinusoidal Embeddings

This post goes into the math behind sinusoidal embeddings as mentioned in the original "Attention Is All You Need" paper. This is mostly for my understanding of the topic and the Workshop.

Why Positional Encoding?

When the paper was released, the state of the art models were Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). To keep it simple: RNNs have a memory property—they can look at previous states and *infer* the positions of each word. This is not true for transformers, which can't distinguish the position of token i from token j.

Attention is completely position-invariant. It doesn't show the order or positions of tokens. However, word position is crucial for modeling language. For example: "dog bites man" and "man bites dog" are two very different sentences.

We need a way to teach the model the position of words in a sentence so it can answer: "what is the 50th token in the sequence?" and "is token n three tokens *before* token m?"

The key idea: shift in token position → shift in phase.

The Proposed Solution

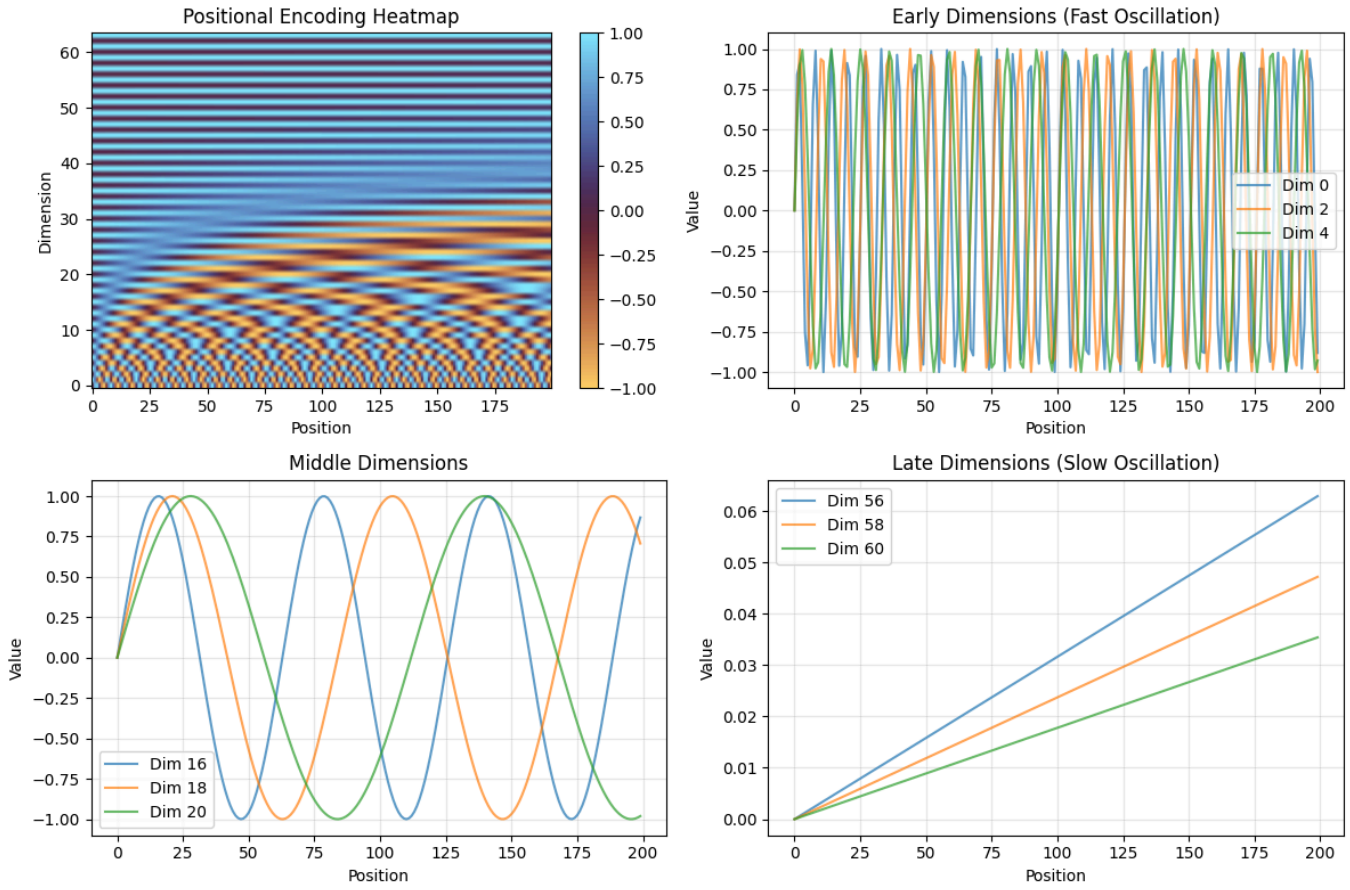
To fix this, the paper's authors used the following functions:

$$\mathrm{PE}_{(pos,2i)} = sin\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = cos\left(\frac{pos}{10000^{2i/d_{model}}}\right)$$

where:

- $pos =$ position of the token
- $i =$ dimension index

When we chart this, we get:



The implementation looks like this:

```
>>> import math
>>> import torch
>>>
>>> seq, embd = 5, 4  # 5 tokens, 4 dimensions per token
>>> assert embd % 2 == 0  # sinusoidal embeddings assume `embd` is even
>>> pe = torch.zeros((seq, embd)) ; pe.shape
torch.Size([5, 4])
>>> pe
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]])
>>> pos = torch.arange(0, seq) ; pos.shape # token indices
torch.Size([5])
>>> pos
tensor([0, 1, 2, 3, 4])
>>> # add dimension at 1st index [5] -> [5, 1]
>>> pos = pos.unsqueeze(1) ; pos.shape
torch.Size([5, 1])
```

```
>>> pos
tensor([[0],
        [1],
        [2],
        [3],
        [4]])
>>> denom = torch.exp(torch.arange(0, embd, 2) * -(math.log(10_000) / embd))
>>> #                  ^^^^^^^^^^^^^^^^^^^^^^^^^^^~ we need pairs of dimenison
to have same frequency, e.g. i = 1 => (0,1), i = 2 => (2, 3), ...
>>> denom
tensor([1.0000, 0.0100])
>>> denom.shape
torch.Size([2])
>>> pe[:, 0::2] = torch.sin(pos * denom)
>>> pe
tensor([[ 0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.8415,  0.0000,  0.0100,  0.0000],
        [ 0.9093,  0.0000,  0.0200,  0.0000],
        [ 0.1411,  0.0000,  0.0300,  0.0000],
        [-0.7568,  0.0000,  0.0400,  0.0000]])
>>> pe[:, 1::2] = torch.cos(pos * denom)
>>> pe
tensor([[ 0.0000,  1.0000,  0.0000,  1.0000],
        [ 0.8415,  0.5403,  0.0100,  0.9999],
        [ 0.9093, -0.4161,  0.0200,  0.9998],
        [ 0.1411, -0.9900,  0.0300,  0.9996],
        [-0.7568, -0.6536,  0.0400,  0.9992]]
>>> # remember to `pe.unsqueeze(0)` : shape [1, seq_len, embd]
>>> # for batching!
```

Why The Code Is In Log Space

Numerical stability.

$$
\begin{aligned}
\frac{1}{10000^{2i/d}} &= \frac{1}{e^{\mathrm{n}\, 10000^{2i/d}}} && \left(x = e^{\mathrm{n}\, x}\right) \\
&= e^{-\mathrm{n}\, 10000^{2i/d}} && \left(\frac{1}{} = {}^{-1}\right) \\
&= e^{-(2i/d)\,\mathrm{n}\, 10000} && \left(\mathrm{n}\ = \mathrm{n}\right) \\
&= e^{-ln\ 100000\ (2i/d)}
\end{aligned}
$$

We Can't Actually Prove This Is Exactly What's Happening

We have theoretical proof for why things *can* happen this way—the encodings can theoretically encode position linearly, in a way where models can learn the underlying patterns and *know* they represent positions.

But we don't *actually* know if the model interprets them as we expect. Large Language Models are black box function approximators. Empirically, when you ablate or change the encodings, performance changes. However, the model could be:

- Learning positions explicitly
- Learning *something* correlated with position
- Learning a completely different pattern that happens to work
- Using encodings in ways we don't know about

A typical large model has millions of parameters across attention heads, feedforward networks, residual connections, etc. Positional information flows through all of it. A linear layer is just one hypothesis of how it *might* work.

What we do know: when you remove positional encoding, the model breaks.

What's (Maybe) Actually Happening

You have tokens as rows and token dimensions as columns. When you move from one token to the next, the early dimensions show large differences according to the sinusoidal embeddings. As you move further down the dimensions, the difference at a specific position i (between two tokens j and k) becomes very small.

When you take the dot product of these two token vectors, that small difference appears. You can measure how similar or far apart the two tokens are via the dot product, since it implicitly encodes similarity.

We add this matrix of computed sines and cosines to the input and forward it to the Transformer block. Now, learned linear projections (Q, K matrices) can amplify and extract these embedded position signals from the tokens.

Sinusoidal embeddings give the model access to **relative position using linear operations**, whereas learned absolute embeddings don't generalize to unseen lengths in the same way.

The Choices That Led to This Math

To encode position into attention, the authors wanted a function satisfying these constraints:

1. Deterministic (learned positions were considered, but this is cheaper)

2. Smooth with respect to position
3. Multi-scale (short- and long-range positions are represented)
4. Linearly exploitable for relative position
5. Stable for long sequences

The sinusoidal formula is basically the simplest function satisfying all of them. Let's examine each design choice:

Why `pos` in the Numerator?

Because it's the independent variable. You want the encoding to change as you move forward in the sequence. Writing $sin(_i \cdot pos)$ means "position advances → phase advances."

If it were in the denominator, early tokens would change wildly and later tokens would barely move. We want the opposite: linear progress through the sequence should correspond to linear phase motion.

Why? Because if you did something like $sin(1/pos)$, you immediately lose the property $(p+k) \ \text{simpefunctionof} \ (p)$. There's no fixed matrix M_k, and the relationship becomes position-dependent. If phase advance is non-linear in position, the rotation depends on p itself and the whole argument for sinusoidal embeddings collapses.

Suppose we used $sin((p))$ where (p) is non-linear (e.g., p^2 or $\circ\, p$). There's no identity letting us write $(p+k) = (p) + \text{constantdependinonon } k$. When we try to repeat the derivation, the coefficients need both p and k. There's no fixed matrix M_k, and a linear layer cannot represent shifts in position. We'd get $\text{PE}_{p+k} = M_{p,k}\text{PE}_p$, which is useless since the model would need different weights for every absolute position—defeating the point.

So `pos` goes in the numerator because we want translation along the sequence to map to rotation in feature space.

Why Divide by Something at All?

If we just used $sin(pos)$, all dimensions would oscillate at the same frequency. Tokens 1 and 100 wouldn't have any meaningful difference due to modulo periodicity.

When we divide by a scale factor, we get different frequencies:

- Small denominators = fast oscillations (local coherence)
- Large denominators = slow oscillations (global coherence)

This lets us encode both "nearby" and "far-apart" in the same vector.

Why 10000?

It's not sacred—you could use 8192. It's small enough to avoid numerical stability issues and large enough to cover most sequence lengths. It's a baked-in hyperparameter. Changing it only changes the frequency range. It was chosen because it gives every order of magnitude in token distance enough representational capacity.

Why the Exponent $2i/d_{model}$?

Because it maps dimension index to frequency logarithmically. As i increases linearly, the denominator grows *exponentially*, giving geometrically spaced frequencies instead of linearly spaced ones.

Why is linear spacing bad? Because frequencies would be too close at one scale. Log spacing gives equal resolution to both short and long ranges.

What's "resolution" here? It's how finely the representation can distinguish changes in position at a given scale (e.g., 1-10 tokens, 100-200 tokens, 600-700 tokens). Language has structure at many scales, and you don't know in advance which scale matters. We want representational power to capture short, medium, and long-range dependencies simultaneously. No scale is starved of bandwidth.

Why Sine and Cosine?

Because together they form a *phase-preserving basis*. When the phase shifts (or we advance along the sequence), the representation changes in a way that keeps phase information intact and recoverable.

If you only used $x = sin()$, you'd immediately lose information: $sin(-)$ gives the same value. Two phases map to the same output.

But with the pair (sin, cos), when you advance the token position, the phase shift becomes a fixed linear transformation. How do we know this?

Encode each position p as the vector $(sin(p), cos(p))$—a point on the unit circle at angle p.

For positions p and q, the dot product becomes:

$$v(p) \cdot v(q) = sin(p)sin(q) + cos(p)cos(q)$$

Using the trig identity $sin\ \cdot sin\ + cos\ \cdot cos\ = cos(-)$:

$$v(p) \cdot v(q) = cos((p - q))$$

The dot product eliminates absolute position and gives relative position offset. The model doesn't need to subtract positions or compute p and q explicitly—the information about how far apart tokens are (up to periodicity) just emerges.

Why We Don't Actually Use Sinusoidal Embeddings Anymore

In practice, we use RoPE. It doesn't change the math fundamentally but addresses limitations of sinusoidal embeddings:

1. **Position is injected too early**: Positions get mixed with token semantics, passed through layers, and distorted. The model still uses it, but indirectly.
2. **Relative position is implicit**: We can prove representation is possible with linear layers, but the model must discover the underlying structure on its own.
3. **Periodicity, frequency, and scaling issues** as sequence length grows.

The idea is neat and works, but can we do better? Can we apply "positional information as phase changes" more effectively? Yes.

RoPE (Rotary Positional Encoding)

RoPE takes the rotation-matrix insight and moves it **inside attention itself**.

Instead of:

- Encoding position into embeddings
- Hoping linear layers recover relative offsets

RoPE rotates the query and key vectors by a position-dependent angle.

When attention computes QK^T, it actually computes:

$$(Q \cdot {}_p)(K \cdot {}_q)^T$$

Because rotations compose nicely:

$${}^T_p \cdot {}_q = {}_{q-p}$$

The dot product itself becomes a function of relative position. It's the same idea—phase shift = relative position—but cleaner, since the model doesn't need to extract the information anymore. It's already there, enforced by structure instead of being recoverable later during training.

State of the Art (SOTA) Peek

Current research has models to train of basically infinite context length. This is highly experimental and updates are coming as in basically right now, but the main idea is to use RoPE as training wheels. Fundamentally, you train the model with RoPE turned on. So the model learns the positional encoding of the tokens, but then, you freeze training, checkpoint and remove the positional encoding, and start training again. This results in a major spike in

loss which goes down very fast back to the previous levels. This allows us to train models with a much larger context length than what the embeddings would allow for.