

# Cross-entropy Loss Intuition

explain properly.

The cross-entropy loss before training is 4.17, and after 1000 epochs is 3.93. How can we make sense of it intuitively?

Cross-entropy in this context is referring to how likely we are to pick the wrong word. So here,

$H$

$($

$T$

$,$

$q$

$)$

$=$

$-$

$\sum$

$i$

$=$

$1$

$N$

$1$

$N$

$\log$

$q$

$($

$x$

$i$

$)$

$)$

where

$q$

$($

$x$

$i$

$)$

is the probability of picking the right word, as estimated by the model. If

$q$

$($

$x_i$   
 is close to 1, then  
 $\log q_i$   
 is close to 0; similarly, if  
 $q_i$   
 is small, then  
 $\log q_i$   
 is a large negative number, so  
 $-\log q_i$   
 will be a large positive number. Now to build the intuition:  
 to start,  
 $-\log q_i$   
 $=$   
 $4.17$   
 , so  
 $q_i$   
 $=$   
 $0.015$   
 , or around  
 $1/64.715$   
 . Recall that the vocabulary size  
 $|V|$   
 $=$   
 $65$   
 , so what we're basically saying here is that the model is as  
 good at choosing the next letter as randomly picking from  
 our vocabulary. After training,  
 $-\log q_i$   
 $=$

3.93

, so we're now basically choosing between 50 letters. This is a very small improvement, so something is probably wrong.

To get an intuition for how the loss relates to the model's performance, think about the model choosing among

$V$

$\sim$

tokens; when

$V$

$\sim$

is small, the model is more likely to guess right. In addition, we know

$\max$

$V$

$\sim$

$=$

$V$

, which can help us understand if our model is learning at all.

$V$

$\sim$

$=$

$\exp$

$($

$L$

$)$

Let me break this down intuitively and clearly.

## ? What's happening?

You trained a model to predict the next character (or word), and you're using **cross-entropy loss** to measure how good it is. This loss tells us how *surprised* the model is when it sees the actual correct next token.

- Before training, the loss was **4.17**
- After 1000 epochs, the loss is **3.93**

## Reminder: What is Cross-Entropy?

Cross-entropy loss for classification is:

$$\text{CrossEntropy} = -\frac{1}{N} \sum_{i=1}^N \log q(x_i)$$

Here:

- $q(x_i)$  is the probability your model assigns to the **correct** token at position  $i$
  - The smaller  $\log q(x_i)$ , the better (since the model is more confident and correct)
- 

## Intuition: How to Interpret the Numbers

Cross-entropy loss can be related to **how many choices (tokens)** the model is effectively guessing between. This is given by:

$$\tilde{V} = \exp(\text{loss})$$

So:

- **Before training:**

$$\tilde{V} = \exp(4.17) \approx 64.7$$

This means the model is effectively choosing **randomly** among ~65 tokens. If your vocabulary size  $|V| = 65$ , the model is basically doing *no better than random guessing*.

- **After training:**

$$\tilde{V} = \exp(3.93) \approx 50.8$$

Now it's like the model is guessing among ~51 tokens. That's only a **minor improvement** over random.

---

## What This Means

- **Before:** Your model assigns ~1.5% chance to the right token on average.

- **After:** Now it assigns ~2% chance. Not much better.

This tells you:

- The model is learning *a little*, but **not much**.
  - Ideally, as training progresses, the loss should drop much more — closer to 1 or 2.
  - A good model should narrow down the "effective vocabulary" it's guessing from to maybe 5–10 tokens or fewer.
- 

## Why Might This Happen?

If the improvement is this small, something is likely wrong:

- Model is underfitting (not complex enough)
  - Training data is not informative or is too small
  - Model architecture isn't learning sequence dependencies (maybe you're not using enough layers or recurrent components for sequential data)
  - Learning rate, optimization, or other hyperparameters are poorly tuned
- 

## Summary

- Cross-entropy loss of 4.17 → ~65-way random guessing
- Loss of 3.93 → ~51-way guessing
- Barely any improvement → model might be stuck
- Ideally, you want the loss to drop much more — indicating the model is predicting more confidently and accurately

Let me know if you'd like help diagnosing why the loss isn't improving much — I can guide you through that too.

what if cross entropy loss is 0.22. my vocab size is 4000 tokens

If your **cross-entropy loss is 0.22** and your **vocabulary size is 4000**, that's a **very good result**. Let's walk through the intuition behind this.

---

## Step-by-Step Intuition

### 1. Effective Vocabulary Size ( $\tilde{V}$ ):

You can estimate the **effective number of choices** your model is guessing from using:

$$\tilde{V} = \exp(\text{loss}) = \exp(0.22) \approx 1.246$$

➡ This means:

Your model is assigning, **on average**, around **80% probability to the correct token** (since  $1/1.246 \approx 0.80$ ).

---

### 2. Compare with Random Guessing:

If your model were guessing randomly:

$$\text{Loss}_{\text{random}} = \log(\text{vocab size}) = \log(4000) \approx 8.29$$

But your model's loss is:

$$0.22 \ll 8.29$$

➡ That's a **huge difference**, and shows your model is **highly confident and accurate**.

---

## What This Implies:

- The model has learned the structure of your data **very well**.
  - It's **predicting the correct token with high probability**.
  - The performance is **far better than random**, which would have a cross-entropy loss of  $\sim 8.29$  for a vocab of 4000.
-

## For Perspective:

| Loss        | ~Effective Vocab Size | Avg. Prob. Assigned to Correct Token |
|-------------|-----------------------|--------------------------------------|
| 8.29        | 4000                  | 0.00025 (random)                     |
| 4.00        | ~55                   | 0.018                                |
| 2.00        | ~7.4                  | 0.135                                |
| 0.69        | ~2                    | 0.50                                 |
| <b>0.22</b> | <b>~1.25</b>          | <b>~0.80</b>                         |

So you're **nearing deterministic predictions** — that's exceptional.

## Caveat

If the loss is **too low too quickly**, also consider:

- Are you **overfitting**? (very low loss on training but poor on validation/test)
- Is your task **too simple or your labels too predictable**?

Otherwise, 0.22 is **excellent** and shows the model is making accurate, confident predictions.

Let me know if you'd like to check whether it's overfitting or visualize prediction probabilities.

llama vs gpt architecture state differences

Here's a clear comparison of the **LLAMA** (Meta) and **GPT** (OpenAI) architectures, specifically focusing on their **decoder-only Transformer** structures and the **state differences** that impact performance, speed, and memory:

## Core Architecture Similarity

Both LLAMA and GPT are **decoder-only Transformers**, meaning they only use the Transformer decoder block (no encoder). They're autoregressive, generating tokens one at a time.

## Key Differences Between LLAMA and GPT

| Component / Feature        | GPT (e.g., GPT-2/3)                            | LLAMA (e.g., LLAMA 2)  |
|----------------------------|--|--|
| Position Embeddings        | Absolute positional embeddings                 | <b>Rotary Positional Embeddings (RoPE)</b> – better generalization to longer context |
| LayerNorm Placement        | <b>Post-norm</b> (after attention/feedforward) | <b>Pre-norm</b> – better stability during training                                   |
| Feedforward Dim Expansion  | 4x hidden size (e.g., 4096 → 16384)            | 2x hidden size (e.g., 4096 → 8192) – faster, lighter                                 |
| Attention Type             | Standard self-attention                        | RoPE + often uses grouped-query attention in newer LLAMA variants                    |
| Parameter Efficiency       | Less efficient per FLOP                        | Highly optimized per FLOP (trained with more efficiency in mind)                     |
| Tokenizer                  | Byte-level BPE (GPT-2/3)                       | SentencePiece (Unigram tokenizer)  |
| Training Stability         | Less stable at scale (GPT-3 had issues)        | LLAMA uses <b>PreNorm</b> + <b>SWIGLU</b> + <b>RoPE</b> for improved training        |
| Output Layer Sharing       | Separate embedding and output layers           | Often shares input and output embeddings (tied weights)                              |
| Inference Memory Footprint | Larger KV cache for attention                  | LLAMA optimizes KV cache layout (smaller memory use)                                 |
| Open Weights?              | No (GPT-3/4 proprietary)                       | Yes (LLAMA 2 and 3 models are openly released)                                       |

## Key Architectural Choices Impacting State

Let's break down "state differences" — i.e., things that affect how **attention states (KV cache)** and **model memory** behave:

### 1. Rotary Positional Embeddings (LLAMA) vs Absolute Positional (GPT):



- GPT uses **learned absolute positions**, which limits generalization to unseen sequence lengths.
- LLAMA uses **RoPE**, which **encodes positions through rotation in complex space** → scales better to longer sequences and generalizes better.

## 2. Key/Value (KV) Cache Memory Layout:

- GPT uses standard tensor layouts, which can be memory-heavy.
- LLAMA optimizes KV cache **to minimize memory and improve speed**, especially in inference.

## 3. Attention Mechanism:

- LLAMA may use **grouped-query or multi-query attention** in some versions, reducing KV duplication across heads → smaller KV state.

---

## Training & Inference Speed Impacts

- **LLAMA models are often faster** during inference because:
  - Smaller FFN dimension (2x vs 4x).
  - Pre-norm helps gradient flow.
  - RoPE allows for longer contexts.
  - Memory-efficient KV cache layout.

---

## Example in Practice

| Metric                    | GPT-3 (175B) | LLAMA 2 (70B)                                  |
|---------------------------|--------------|--|
| Parameters                | 175B         | 70B  |
| Training Compute (FLOPs)  | ~3.1e23      | ~2.6e23  |
| Tokens Trained On         | ~300B        | 2T   |
| Performance on benchmarks | Comparable   | Comparable or better (esp. on open benchmarks) |

| Metric                     | GPT-3 (175B) | LLAMA 2 (70B) |
|----------------------------|--------------|---------------|
| Inference latency & memory | Higher       | Lower         |

### ✔ Summary

- LLAMA architecture is **more efficient and optimized** (especially for open-source deployment).
- GPT architecture is **powerful but heavier**, and newer models (like GPT-4) use undisclosed upgrades.
- LLAMA’s use of **RoPE, PreNorm, and efficient feedforward size** makes it fast, stable, and memory-efficient — **great for deployment**.

Let me know if you want a visual diagram or to dive into inference implications (like quantization or KV cache reuse).

tell me about rms norm and code in tensorflow

### ✔ What is RMSNorm?

RMSNorm (Root Mean Square Layer Normalization) is a **normalization technique** that is similar to LayerNorm but **does not subtract the mean**. It’s faster and often more stable in large language models (like LLAMA).

### Difference Between RMSNorm and LayerNorm:

| Aspect         | LayerNorm                                | RMSNorm   |
|----------------|--|---|
| Formula        | $\frac{x-\mu}{\sqrt{\sigma^2+\epsilon}}$ | $\frac{x}{\sqrt{\frac{1}{d}\sum x^2+\epsilon}}$ |
| Uses mean?     | Yes                                      | No  |
| Uses variance? | Yes                                      | Uses only <b>RMS</b> (root mean square)         |
| Speed          | Slower (due to mean and variance)        | <b>Faster</b> (no mean subtraction)             |

| Aspect    | LayerNorm | RMSNorm                               |
|-----------|-----------|---------------------------------------|
| Stability | Good      | Improved in some Transformer variants |

## RMSNorm Formula

Let's denote input as  $x \in \mathbb{R}^d$ :

$$\text{RMSNorm}(x) = \frac{x}{\text{RMS}(x)} \cdot \gamma = \frac{x}{\sqrt{\frac{1}{d} \sum_{i=1}^d x_i^2 + \epsilon}} \cdot \gamma$$

Where:

- $\gamma$  is a learnable scale parameter
- $\epsilon$  is a small constant to prevent division by zero

## TensorFlow Code for RMSNorm

Here's a minimal implementation of **RMSNorm as a custom layer**:

python

```
import tensorflow as tf

class RMSNorm(tf.keras.layers.Layer):
    def __init__(self, hidden_size, epsilon=1e-8, **kwargs):
        super(RMSNorm, self).__init__(**kwargs)
        self.hidden_size = hidden_size
        self.epsilon = epsilon

    def build(self, input_shape):
        # Learnable scale parameter  $\gamma$  (same shape as last dim of input)
        self.scale = self.add_weight(
            name="scale",
            shape=(self.hidden_size,),
            initializer="ones",
            trainable=True
        )
```

```
def call(self, x):
    rms = tf.sqrt(tf.reduce_mean(tf.square(x), axis=-1, keepdims=True) +
self.epsilon)
    norm_x = x / rms
    return norm_x * self.scale
```

## Example Usage

python

```
# Input tensor (batch_size, seq_len, hidden_dim)
inputs = tf.random.normal(shape=(32, 128, 512))

# RMSNorm layer
rmsnorm = RMSNorm(hidden_size=512)
output = rmsnorm(inputs)
```

## When to Use RMSNorm

- In **LLMs or Transformers** where training speed and memory matter
- For better **training stability** in deep models
- As a drop-in replacement for LayerNorm (but test on your task)

Let me know if you want a PyTorch version or to benchmark it vs LayerNorm!

self.add\_weight ??

Great question. Let's break down `self.add_weight` in TensorFlow clearly.

## What is `self.add_weight`?

In TensorFlow's Keras API, `self.add_weight` is a method used inside a custom layer to create trainable or non-trainable weights (like parameters in your layer).

### Syntax:

```
python

self.add_weight(
    name=None,
    shape=None,
    dtype=None,
    initializer=None,
    trainable=True,
    ...
)
```

## What It Does in the RMSNorm Example:

In the RMSNorm code:

```
python

self.scale = self.add_weight(
    name="scale",
    shape=(self.hidden_size,),
    initializer="ones",
    trainable=True
)
```

It does the following:

- Creates a weight tensor named `"scale"`
- With shape `[hidden_size]` (one learnable scale per feature)
- Initialized to all ones

- **Trainable**, so it will get updated during backpropagation

This is equivalent to the  $\gamma$  (**gamma**) in LayerNorm — a per-dimension scaling factor.

## Why Use It?

When you're building a custom layer in Keras, you **should use** `self.add_weight` instead of `tf.Variable` directly because:

1. It automatically registers the weight with the layer.
2. It ensures the weight shows up in `.trainable_variables`.
3. It supports model saving/loading and training integration.

## Quick Example Outside RMSNorm:

python

```
class MyBiasLayer(tf.keras.layers.Layer):
    def build(self, input_shape):
        self.bias = self.add_weight(
            name="bias",
            shape=(input_shape[-1],),
            initializer="zeros",
            trainable=True
        )

    def call(self, inputs):
        return inputs + self.bias
```

Let me know if you want to use `self.add_weight` to create non-trainable weights or in a more advanced setting like attention layers.

why axis=-1, keepdims=True ?

Excellent question. Let's break down why we use `axis=-1` and `keepdims=True` in this line from the RMSNorm implementation:

```
python
```

```
rms = tf.sqrt(tf.reduce_mean(tf.square(x), axis=-1, keepdims=True) + self.epsilon)
```

## Purpose of the Line

This computes the **root mean square (RMS)** value **per vector (last dimension)** of the input tensor `x`.

## What does `axis=-1` mean?

- It means: "**reduce across the last dimension**"
- For an input of shape `(batch_size, seq_len, hidden_size)`, `axis=-1` means you're computing the RMS for **each token embedding** of size `hidden_size`

So you're computing RMS for each `[hidden_size]` slice independently.

## What does `keepdims=True` do?

- It **preserves the number of dimensions** in the output.
- Without `keepdims=True`, you'd get shape `(batch_size, seq_len)` after reducing over the last dimension.
- But you want to **divide** `x` **by** `rms`, and that only works cleanly if `rms` has shape `(batch_size, seq_len, 1)` — so it can be broadcast across the last dimension.

## Why both are important?

If you had:

```
python
```

```
rms = tf.sqrt(tf.reduce_mean(tf.square(x), axis=-1)) # No keepdims
```

Then `rms` would have shape `(batch, seq_len)`, and the division:

```
python
```

```
x / rms
```

would fail or broadcast incorrectly.

## ✓ Summary

| Argument                   | Why it's needed   |
|----------------------------|---|
| <code>axis=-1</code>       | Normalize each feature vector independently (last dimension)              |
| <code>keepdims=True</code> | Retain shape so it can broadcast during division ( <code>x / rms</code> ) |

Let me know if you want a visual tensor example or explanation of broadcasting here!

(`batch_size, seq_len, hidden_size`) i have as `x`. you perform rms norm on `axis=-1`. what about batchnorm ? how it works here ? showby code writing batchnorm class.

Great — you're asking how **BatchNorm** compares with **RMSNorm** in terms of dimension reduction and implementation. Let's walk through:

## ✓ Given:

Input tensor `x` has shape:

`(batch_size, seq_len, hidden_size)`



This is typical for Transformer outputs.

---

## 🤔 How BatchNorm Works:

**Batch Normalization** normalizes across the **batch dimension**, computing **mean and variance per feature** (i.e., per hidden size dimension).

So:

- For input shape `(B, T, H)` :
  - Mean and variance are computed over **both B and T** (i.e., across all tokens in the batch).
  - One mean/variance per feature `h` (total `hidden_size` scalars).

So we reduce over `axes=[0, 1]`.

---

## 🔧 BatchNorm from Scratch in TensorFlow (Custom Layer):

python

```
import tensorflow as tf

class CustomBatchNorm(tf.keras.layers.Layer):
    def __init__(self, hidden_size, momentum=0.99, epsilon=1e-5, **kwargs):
        super(CustomBatchNorm, self).__init__(**kwargs)
        self.hidden_size = hidden_size
        self.momentum = momentum
        self.epsilon = epsilon

    def build(self, input_shape):
        # Learnable parameters: scale (gamma) and shift (beta)
        self.gamma = self.add_weight(
            name="gamma",
            shape=(self.hidden_size,),
            initializer="ones",
            trainable=True
        )
        self.beta = self.add_weight(
```

```

        name="beta",
        shape=(self.hidden_size,),
        initializer="zeros",
        trainable=True
    )
    # Running stats (non-trainable)
    self.moving_mean = self.add_weight(
        name="moving_mean",
        shape=(self.hidden_size,),
        initializer="zeros",
        trainable=False
    )
    self.moving_variance = self.add_weight(
        name="moving_variance",
        shape=(self.hidden_size,),
        initializer="ones",
        trainable=False
    )

def call(self, x, training=False):
    if training:
        # Compute mean & variance over batch and time
        mean = tf.reduce_mean(x, axis=[0, 1])
        variance = tf.reduce_mean(tf.square(x - mean), axis=[0, 1])

        # Update running averages
        self.moving_mean.assign(self.momentum * self.moving_mean + (1 -
self.momentum) * mean)
        self.moving_variance.assign(self.momentum * self.moving_variance + (1 -
self.momentum) * variance)
    else:
        # Use moving averages during inference
        mean = self.moving_mean
        variance = self.moving_variance

    norm_x = (x - mean) / tf.sqrt(variance + self.epsilon)
    return norm_x * self.gamma + self.beta

```

## Summary of Axis Differences:

| Normalization | Mean/Variance Axis                                 | Learnable Params Shape      | Typical Use  |
|---------------|--|-----------------------------|--|
| RMSNorm       | <code>axis=-1</code> (per token)                   | <code>(hidden_size,)</code> | Transformers, LLMs                                   |
| BatchNorm     | <code>axis=[0, 1]</code> (across batch + sequence) | <code>(hidden_size,)</code> | CNNs, MLPs (can be used in NLP, but less common now) |

## ⚠ In NLP / Transformers:

BatchNorm is **rarely used** because:

- It's sensitive to variable-length sequences.
- Training can be unstable with varying batch sizes.
- LayerNorm or RMSNorm is preferred for **sequence-level** normalization.

Let me know if you'd like LayerNorm from scratch too, or how these affect convergence!

1/2

explain rope intuitively and example along with formulas.

Absolutely — let's dive into **RoPE** (Rotary Positional Embeddings) and build up your intuition with examples, formulas, and clear reasoning.

## 🧠 What Is RoPE?

**RoPE** = **Rotary Positional Embeddings**, introduced in [RoFormer \(Su et al., 2021\)](#).

It's a way to encode **positional information** directly into the **attention mechanism**, by **rotating query and key vectors** — rather than adding or concatenating positional embeddings.

## ✓ Why RoPE? (vs absolute positions)

- Traditional Transformers use **absolute position embeddings** (e.g., learned vectors for each position).
- These are **added to the input**, but don't generalize to **longer sequences** than seen in training.
- **RoPE lets the model generalize better**, and handles **relative positions implicitly** — helping performance on long contexts.

## High-Level Intuition

Imagine:

- You want each token to carry "**where it is**" in the sequence.
- Instead of saying "this is position 5" via a learned embedding, RoPE encodes position **as a rotation angle** in vector space.

Tokens "rotate" in embedding space depending on their position — closer tokens have smaller angle differences, and that becomes meaningful when **dot products** (attention scores) are computed.

## RoPE Core Idea (Formulas)

Let's break this down step-by-step for a 1D vector.

### 1. Split Query/Key into Pairs of 2D Elements

If you have a vector  $x$  of dimension  $d$ , break it into  $d/2$  2D blocks:

$$x = [x_1, x_2, x_3, x_4, \dots, x_{d-1}, x_d] \Rightarrow [(x_1, x_2), (x_3, x_4), \dots, (x_{d-1}, x_d)]$$

Each pair gets **rotated** based on the token's position.

## 2. Apply Rotation per Position

For position  $p$ , and 2D block  $(x_1, x_2)$ , apply rotation:

$$\begin{bmatrix} x'_1 \\ x'_2 \end{bmatrix} = \begin{bmatrix} \cos(\theta_p) & -\sin(\theta_p) \\ \sin(\theta_p) & \cos(\theta_p) \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

This is a standard **2D rotation matrix**.

---

## 3. What is $\theta_p$ ?

$$\theta_p = p / 10000^{2i/d}$$

Where:

- $p$  = position
- $i$  = dimension index (within the pair)
- $d$  = total dimension

Just like sinusoidal encoding: lower dimensions rotate faster, higher dimensions slower.

---

## 4. Apply This to Queries and Keys

In **attention**, instead of using regular  $q$  and  $k$ , RoPE applies **position-based rotations**:

$$q' = \text{RoPE}(q, p_q), \quad k' = \text{RoPE}(k, p_k)$$

Now compute attention as:

$$\text{score}(i, j) = q'_i \cdot k'_j$$

This **implicitly encodes relative position**  $(i - j)$  in the dot product!

---



## Intuitive Analogy

- Imagine each token has a **direction vector**.

- With RoPE, tokens at different positions **spin** (rotate) by different angles.
- When you compute similarity (via dot product), the **angle difference = relative position**.
- The model can now **focus based on distance** in a **learned, flexible** way.

## Simple Example (Python/Numpy)

python

```
import numpy as np

def rope_rotate(x, pos, dim):
    x = x.reshape(-1, 2)
    theta = 1.0 / (10000 ** (2 * np.arange(x.shape[0]) / dim))
    angle = pos * theta
    cos = np.cos(angle)
    sin = np.sin(angle)
    x1 = x[:, 0]
    x2 = x[:, 1]
    x_rot = np.stack([x1 * cos - x2 * sin, x1 * sin + x2 * cos], axis=-1)
    return x_rot.reshape(-1)

# Example
x = np.array([1.0, 0.0, 0.5, 0.5]) # 4D vector = 2 pairs
rotated = rope_rotate(x, pos=5, dim=4)
print(rotated)
```

## Advantages of RoPE

| Feature                                 | Benefit                             |
|---|-------------------------------------|
| Encodes <b>relative positions</b>       | Great for long-range attention      |
| No need for learned position embeddings | Lighter                             |
| Generalizes to longer sequences         | Unlike absolute position embeddings |

| Feature                | Benefit                            |
|------------------------|------------------------------------|
| Improves extrapolation | Especially in LLMs (LLAMA uses it) |

## ✓ Summary

- RoPE applies **position-based rotations** to queries and keys.
- This encodes **relative positional info** inside the attention dot product.
- It's fast, parameter-free, and **generalizes better** to longer contexts.
- Widely used in LLAMA, GLM, and other modern Transformers.

Let me know if you want this visualized with plots or built into a Transformer block!

so rope gets applied in attention mechanism or before it with embedding vector

Great question — this gets to the heart of how **RoPE** is different from other positional encoding methods.

## ✓ Where is RoPE applied?

RoPE is **not added to the embedding vector** (like absolute or sinusoidal positional encodings).

Instead, **RoPE is applied directly inside the attention mechanism**, specifically to the **query ( Q )** and **key ( K )** matrices, before computing attention scores.

## 🧩 How it fits into the Transformer:

Let's break it down step-by-step:

### 🔄 Traditional (Absolute) Positional Encoding:

```
python
```

```
x = token_embedding + position_embedding
```

- Position is **added** before attention or any layers.

### With RoPE:

```
python

Q = linear_projection(x)
K = linear_projection(x)
V = linear_projection(x)

Q_rotated = apply_rope(Q, position)
K_rotated = apply_rope(K, position)

attention_scores = Q_rotated @ K_rotated.T
```

- RoPE is applied **after projection to Q and K**, but **before dot product** in self-attention.
- Vectors **Q** and **K** are rotated **according to their position**.
- No positional embedding is added to **x**.

### In Code (Pseudocode Style):

```
python

def self_attention(x):
    Q = x @ Wq  # shape: (batch, seq_len, dim)
    K = x @ Wk
    V = x @ Wv

    Q = apply_rope(Q, positions)
    K = apply_rope(K, positions)

    scores = Q @ K.T
    weights = softmax(scores / sqrt(d))
    return weights @ V
```



## 🤔 Why Not Add RoPE to Embeddings?

Because RoPE is **designed to encode position as a geometric transformation** (rotation) that modifies how attention scores behave. Adding it to embeddings would **lose the relative position benefit** RoPE is meant to give.

## ✅ Summary

| Positional Encoding Method    | Applied to       | Timing                  |
|-------------------------------|------------------|-------------------------|
| Absolute (learned/sinusoidal) | Token embeddings | Before all layers       |
| RoPE (Rotary)                 | Q and K matrices | Before attention scores |

⚠️ RoPE replaces positional embeddings, not complements them.

Let me know if you want to see RoPE integrated into a real attention layer in TensorFlow or PyTorch.

explain SwiGLU activation intuitively and example along with formulas.

**SwiGLU** is a **gated activation** that combines the benefits of a **gating mechanism** (like GLU) with the smooth, non-linear behavior of the **Swish** (SiLU) function. It's become popular in modern Transformer feed-forward layers because it often gives better performance and faster convergence than plain ReLU, GELU, or even the original GLU.

## 1. 🔍 Intuition

### 1. Gating:

- A gate lets the model **modulate** (i.e. turn up or down) part of its activation, rather than passing everything through equally.
- In a plain GLU, you split a linear projection into two halves,  $a$  and  $b$ , and compute

$$\text{GLU}(a, b) = a \times \sigma(b),$$

where  $\sigma(b)$  (sigmoid) acts as the gate: values near 0 shut the signal off, values near 1 let it pass.

## 2. Swish (SiLU):

- Swish (also known as SiLU) is  $x \times \sigma(x)$ . It's smoother than ReLU/GELU and often improves gradient flow.
- Instead of using  $\sigma(b)$  as the gate, SwiGLU uses  $\text{SiLU}(b) = b \cdot \sigma(b)$ . This gives a gate that can go negative (unlike  $\sigma$ ), and scales itself by its own value.

## 3. Putting it together:

- You still split your linear projection into two vectors  $(a, b)$ .
- Compute

$$\text{SwiGLU}(a, b) = a \odot \text{SiLU}(b) = a \odot (b \times \sigma(b)).$$

- This means **the gate both squeezes** (via  $\sigma$ ) **and scales** (via the  $b$  multiplier) the signal  $a$ .

Because the gate is smoother and self-scaling, SwiGLU often learns richer interactions than plain GLU or standard activations.

## 2. Formulas

1. **Split** your projected vector  $z \in \mathbb{R}^{2d}$  into two halves of size  $d$ :

$$z = [a, b], \quad a, b \in \mathbb{R}^d.$$

2. **Gate** with SiLU:

$$\text{SiLU}(b) = b \times \sigma(b) \quad \text{where } \sigma(b) = \frac{1}{1 + e^{-b}}.$$

3. **Multiply:**

$$\text{SwiGLU}(z) = a \odot \text{SiLU}(b) = a \odot (b \cdot \sigma(b)).$$

4. **Feed-forward layer** with SwiGLU:

$$\text{FFN}(x) = W_2 \left( \text{SwiGLU}(W_1 x) \right) + b_2,$$

where  $W_1 \in \mathbb{R}^{2d \times h}$ ,  $W_2 \in \mathbb{R}^{h \times 2d}$ , and  $h$  is the “intermediate” size.

### 3. TensorFlow Example

python

```
import tensorflow as tf

class SwiGLU(tf.keras.layers.Layer):
    def __init__(self, dim, **kwargs):
        super().__init__(**kwargs)
        self.dim = dim

    def build(self, input_shape):
        # W1 projects to 2*dim
        self.w1 = self.add_weight(
            "w1", shape=(input_shape[-1], 2*self.dim),
            initializer="glorot_uniform", trainable=True
        )
        self.b1 = self.add_weight(
            "b1", shape=(2*self.dim,),
            initializer="zeros", trainable=True
        )
        # W2 projects back to output dim
        self.w2 = self.add_weight(
            "w2", shape=(self.dim, input_shape[-1]),
            initializer="glorot_uniform", trainable=True
        )
        self.b2 = self.add_weight(
            "b2", shape=(input_shape[-1],),
            initializer="zeros", trainable=True
        )

    def call(self, x):
        # Linear -> split
        z = tf.matmul(x, self.w1) + self.b1 # shape (... , 2*dim)
        a, b = tf.split(z, num_or_size_splits=2, axis=-1)

        # SwiGLU activation
        gate = b * tf.sigmoid(b) # SiLU(b)
```

```

gated = a * gate                                #  $a \odot \text{SiLU}(b)$ 

# Project back
out = tf.matmul(gated, self.w2) + self.b2
return out

# Usage
x = tf.random.normal((32, 128, 512))            # e.g. batch, seq_len, hidden
ffn = SwiGLU(dim=2048)                          # intermediate size
y = ffn(x)                                       # shape (32,128,512)

```

## 4. 🚀 Why SwiGLU Helps

- **Richer gating** than GLU: gate can be negative or  $>1$ , and self-scaling.
- **Smoother gradients** than plain sigmoid gating.
- **State-of-the-art** in many LLMs (e.g., GPT-4, LLAMA) to speed up convergence and improve downstream performance.

Let me know if you'd like to see benchmarks comparing SwiGLU to other activations or dig into GeGLU (using GELU instead of SiLU)!

2/2

this si also correct ?

```

import tensorflow as tf
from tensorflow.keras import layers

class SwiGLU(tf.keras.layers.Layer):
    def __init__(self, hidden_dim):
        super().__init__()
        self.lin1 = layers.Dense(2 * hidden_dim) # W1
        self.lin2 = layers.Dense(hidden_dim)     # W2

    def call(self, x):
        x_ = self.lin1(x)                        # shape: (... , 2d)
        a, b = tf.split(x_, num_or_size_splits=2, axis=-1) # split
        gated = a * (b * tf.sigmoid(b))          # SwiGLU:  $a \odot$ 

```

```
SiLU(b)
return self.lin2(gated)
```

Yes — your shorter implementation is functionally correct.

Here's why it matches the formal SwiGLU definition and a couple of optional tweaks you might consider.

## ✓ Why it works

python

```
x_ = self.lin1(x)                # → (... , 2 · hidden_dim)
a, b = tf.split(x_, 2, axis=-1)   # a, b ∈ ℝ^{hidden_dim}
gated = a * (b * tf.sigmoid(b))   # a ⊙ SiLU(b)
return self.lin2(gated)           # → (... , hidden_dim)
```

### 1. Projection to 2 d

`lin1` plays the role of  $W_1$ , producing the concatenated vectors  $a$  and  $b$ .

### 2. SwiGLU activation

`b * tf.sigmoid(b)` is exactly the SiLU/Swish of  $b$ .

Multiplying by `a` implements  $a \odot \text{SiLU}(b)$ .

### 3. Projection back

`lin2` is  $W_2$ , bringing the dimension back to `hidden_dim` (your model size).

Dense layers already include bias terms and a good default initializer, so everything needed for training is there.

## 🔧 Optional refinements

| Suggestion               | Code tweak   | Why / when                        |
|--------------------------|--|-----------------------------------|
| Use the built-in SiLU op | <code>gate = tf.nn.silu(b)</code> then <code>gated = a * gate</code> | Slightly clearer, identical math. |

| Suggestion  | Code tweak  | Why / when  |
|---|---|---|
| Separate <b>model dim</b> and <b>intermediate dim</b> | <pre>python\nclass SwiGLU(tf.keras.layers.Layer):\n    def __init__(self, model_dim, ffn_dim):\n        ...\n        self.lin1 = layers.Dense(2 * ffn_dim)\n        self.lin2 = layers.Dense(model_dim)</pre> | In many Transformers you expand to $2 \times \text{ffn\_dim}$ ( $\approx 4 \times \text{model\_dim}$ ) and then project back. |
| Add dropout between the two Dense layers              | <pre>gated = self.dropout(gated, training=training)</pre>   | Mirrors common FFN structure in Transformers.   |

## 🧩 Putting those tweaks together (optional)

python

```
class SwiGLU(tf.keras.layers.Layer):
    def __init__(self, model_dim, ffn_dim, dropout_rate=0.0):
        super().__init__()
        self.lin1 = layers.Dense(2 * ffn_dim)
        self.lin2 = layers.Dense(model_dim)
        self.dropout = layers.Dropout(dropout_rate)

    def call(self, x, training=False):
        a, b = tf.split(self.lin1(x), 2, axis=-1)
        gated = a * tf.nn.silu(b)
        gated = self.dropout(gated, training=training)
        return self.lin2(gated)
```

But if you just need a compact SwiGLU block, your original snippet is perfectly fine.

suppose i have multi-head attention going by

#

# Forward pass

```
#
```

---

---

---

```
def call(
    self,
    query,
    value=None,
    key=None,
    mask=None,
    use_causal_mask=False,
    training=None
):
    if value is None:
        value = query
    if key is None:
        key = value

    B = tf.shape(query)[0]

    # 1. Linear projections
    q = self.wq(query) # (B, T_q, d_model)
    k = self.wk(key)   # (B, T_k, d_model)
    v = self.wv(value) # (B, T_v, d_model)

    # 2. Reshape for multi-head
    q = self._split_heads(q, B) # (B, h, T_q, depth)
    k = self._split_heads(k, B) # (B, h, T_k, depth)
    v = self._split_heads(v, B) # (B, h, T_v, depth)
```

how to apply rotatory embedding.

Below is a “drop-in” recipe that shows **where** and **how** to inject Rotary Positional Embeddings (RoPE) into the multi-head attention block you pasted.

---

## 1. Where does RoPE go?

SCSS

```
query → Wq ┘  
key   → Wk ┘ → RoPE(q, k) ← (add here!)  
value → Wv ┘  
  
↓  
scaled-dot-prod-attention
```

Only Q and K are rotated, V is untouched.

## 2. Helper: apply\_rope()

python

```
def apply_rope(x, sin, cos):  
    """  
    x      : (B, h, T, d) even-sized last dim (d must be multiple of 2)  
    sin    : (T, d//2)      broadcastable  
    cos    : (T, d//2)  
    """  
    x_even = x[..., 0::2]      # (... , d/2)  
    x_odd  = x[..., 1::2]      # (... , d/2)  
    # rotate  
    x_rot_even = x_even * cos - x_odd * sin  
    x_rot_odd  = x_even * sin + x_odd * cos  
    # interleave even/odd back together  
    x_rot = tf.stack([x_rot_even, x_rot_odd], axis=-1) # (... , d/2, 2)  
    return tf.reshape(x_rot, tf.shape(x))             # (... , d)
```

## 3. Making the sin/cos tables (on the fly)

python

```
def make_sincos(seq_len, dim, base=10000):  
    """  
    Returns sin, cos with shape (seq_len, dim//2)  
    """
```



```

pos = tf.cast(tf.range(seq_len), tf.float32) # (T,)
i   = tf.cast(tf.range(0, dim, 2), tf.float32) / dim # (d/2,)
theta = pos[:, None] / (base ** i[None, :]) # (T, d/2)
return tf.sin(theta), tf.cos(theta)

```

If you know the maximum sequence length up front you can pre-compute these once in `build()` and reuse them.

## 4. Patch your `call` method

python

```

def call(
    self,
    query,
    value=None,
    key=None,
    mask=None,
    use_causal_mask=False,
    training=None
):
    if value is None:
        value = query
    if key is None:
        key = value

    B = tf.shape(query)[0]
    Tq = tf.shape(query)[1] # sequence length of Q
    Tk = tf.shape(key)[1]

    # 1) Linear projections
    q = self.wq(query) # (B, Tq, d_model)
    k = self.wk(key)   # (B, Tk, d_model)
    v = self.wv(value) # (B, Tv, d_model)

    # 2) Split into heads
    q = self._split_heads(q, B) # (B, h, Tq, depth)
    k = self._split_heads(k, B) # (B, h, Tk, depth)
    v = self._split_heads(v, B) # (B, h, Tv, depth)

```

```

# 3) ----- ROTARY -----
# Build sin/cos for the longest sequence we need this step
max_len = tf.maximum(Tq, Tk)
sin, cos = make_sincos(max_len, self.depth)          # depth = d_model / num_heads

# Slice sin/cos to actual lengths (broadcast works automatically)
q = apply_rope(q, sin[:Tq], cos[:Tq])                # rotate Q
k = apply_rope(k, sin[:Tk], cos[:Tk])                # rotate K
# -----

# 4) Scaled-dot-product attention (unchanged)
scores = tf.matmul(q, k, transpose_b=True) / tf.math.sqrt(
    tf.cast(self.depth, q.dtype)
)                                                       # (B, h, Tq, Tk)
if use_causal_mask:
    # add causal mask here...
    ...
if mask is not None:
    # add padding mask here...
    ...

weights = tf.nn.softmax(scores, axis=-1)
context = tf.matmul(weights, v)                       # (B, h, Tq, depth)

# 5) Merge heads → (B, Tq, d_model)
context = self._merge_heads(context, B)

return self.wo(context)                               # final linear proj

```

## Notes & Tips

- Even dimension** – RoPE needs `depth` (per-head dim) to be an even number because it pairs `[even, odd]` slots.
- Performance** – For production you'd usually pre-compute `sin, cos` once in `build()` for `max_seq_len` and keep them as non-trainable buffers.
- No positional embeddings added elsewhere** – when using RoPE, drop any absolute/sinusoidal additions to the token embeddings.

**4. Only Q & K – leave V untouched.**

That's all you need to add rotary positional embeddings to your existing multi-head attention implementation. 🚀