Understanding Transformers and Self-Attention

Complete Mathematical Deep Dive: From Static to Dynamic Embeddings

Deep Learning Fundamentals

November 13, 2025

Contents

1 Introduction to Transformers

1.1 What are Transformers?

Transformers are a revolutionary neural network architecture introduced in the landmark paper "Attention Is All You Need" (Vaswani et al., 2017). They fundamentally changed Natural Language Processing (NLP) by replacing sequential recurrent architectures (RNNs, LSTMs) with a mechanism based entirely on **parallel attention computations**.

Core Innovation: Transformers process **all tokens simultaneously** while capturing relationships between words through the **self-attention mechanism**, eliminating the sequential bottleneck of RNNs.

1.2 Applications of Transformers

- **Language Models**: GPT-3, GPT-4, LLaMA, Claude, Mistral, DeepSeek
- **Machine Translation**: Google Translate, DeepL
- **Text Understanding**: BERT, RoBERTa (search engines, question answering)
- **Code Generation**: Codex, GitHub Copilot
- **Multimodal AI**: Vision Transformers (ViT), CLIP, GPT-4V
- **Scientific Applications**: AlphaFold2 (protein structure prediction)

2 Why Self-Attention is Needed

2.1 The Critical Problem: Static Embeddings

Static embeddings (Word2Vec, GloVe, TF-IDF) assign a **fixed, unchanging vector** to each word:

$$\text{embedding}(\text{“bank”}) = \mathbf{v}_{\text{bank}} \quad \text{(always identical)} \tag{1}$$

The "Bank" Problem: Consider these two sentences:

- *Money in the **bank** grows* (financial institution)
- *River **bank** flows* (land beside water)

With static embeddings, "bank" receives **identical vectors in both sentences** despite having completely different meanings! This is fundamentally broken.

2.2 The Solution: Contextual (Dynamic) Embeddings

Contextual embeddings adapt based on surrounding words:

$$\text{“bank” in “money bank grows”} \rightarrow \mathbf{v}_{\text{financial}} \tag{2}$$

$$\text{“bank” in “river bank flows”} \rightarrow \mathbf{v}_{\text{geographical}} \tag{3}$$

Where $\mathbf{v}_{\text{financial}} \neq \mathbf{v}_{\text{geographical}}$.

Self-attention achieves this by making each word's representation a **learned function of all other words** in the sentence.

2.3 Bidirectional vs Unidirectional Attention

Type	**Bidirectional (BERT)**	**Unidirectional (GPT)**
Direction	Attends to all words (left + right simultaneously)	Attends only to previous words (left, causal)
Use Case	Understanding tasks (classification, question answering, search)	Generation tasks (text completion, dialogue)
Masking	No causal mask (sees full context)	Causal mask (prevents looking ahead)
Example	BERT reads full sentence before deciding	GPT generates one word at a time
Training	Masked Language Modeling (predict masked words)	Next Token Prediction (predict next word)

Table 1: Bidirectional vs Unidirectional Attention Mechanisms

2.4 Why "Self"-Attention? (Geometric Intuition)

The name "Self-Attention" captures a profound idea: Each word attends to **itself and all other words in the same sequence**.

2.4.1 Three Types of Attention

Type	**What Attends to What**
Self-Attention	Word attends to **same sequence** (including itself)
Cross-Attention	Decoder word attends to **encoder sequence** (different!)
Local Attention	Word attends only to **nearby words** (window)

Table 2: Types of Attention Mechanisms

Why "Self"?

In self-attention, the Query, Key, and Value all come from the **same input sequence**:

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{4}$$

where X is the **same matrix** for all three!

Geometric Representation:

```
Input: "money bank grows"
```

```
Self-Attention (each word looks at ALL words):
money → [money, bank, grows]  ← Attends to itself + others
bank  → [money, bank, grows]  ← Attends to itself + others
grows → [money, bank, grows]  ← Attends to itself + others
```

Contrast with Cross-Attention (Encoder-Decoder):

```
Encoder: "I love cats"
Decoder: "J'aime les chats"

Cross-Attention (decoder attends to encoder):
"J'aime" → [I, love, cats]  ← Decoder word → Encoder words
"les"    → [I, love, cats]  ← Different sequences!
"chats" → [I, love, cats]  ← NOT "self"
```

Geometric Intuition: Think of self-attention as:

- Each word is a point in high-dimensional space (512 dims)
- Self-attention measures: "How should I weight my neighbors?"
- "Neighbors" = all other words (including myself)
- Result: Each point's representation becomes a weighted average of surrounding points

Mathematical Beauty:

$$y_i = \sum_{j=1}^{n} \underbrace{\text{attention}(x_i, x_j)}_{\text{"self" interaction}} \cdot x_j \tag{5}$$

The word x_i attends to all words x_j **including itself** (when $j = i$)!

3 Static vs Dynamic Embeddings: Complete Comparison

Property	**Static (Word2Vec/GloVe)**	**Dynamic (Transformers)**
Context-awareness	None	Full context from entire sentence
Training method	Pre-trained separately on co-occurrence statistics	End-to-end with model via backpropagation
Updates during training	Fixed after pre-training	Continuously updated during training
Representation	Single vector per word type	Different vector for each occurrence
Example: "bank"	Always same vector	Changes: financial vs geographical
Polysemy handling	Cannot distinguish	Disambiguates based on context
Memory	Small (one vector per word)	Larger (parameters + attention)
Quality	Lower for ambiguous words	Higher, context-aware

Table 3: Comprehensive Comparison: Static vs Dynamic Embeddings

Mathematical Distinction:

Static Approach:

$$e_{\text{bank}} = \text{lookup\_table}[\text{“bank”}] \quad \text{(context-independent)} \tag{6}$$

Dynamic Approach (Self-Attention):

$$e_{\text{bank}}^{\text{contextual}} = f(e_{\text{bank}}, e_{\text{money}}, e_{\text{grows}}) \quad \text{(context-dependent)} \tag{7}$$

4 Complete Self-Attention Pipeline

4.1 Working Example

Throughout this document, we'll use this example:

```
Input Text: ‘‘money bank grows’’
```

5 PART 1: Tokenization - Text to Token IDs

5.1 What is BPE Tokenization?

Byte-Pair Encoding (BPE) is an algorithm that builds a vocabulary by iteratively merging the most frequent character pairs. It was originally developed for text compression, then adopted by OpenAI for GPT tokenization.

5.2 BPE Training Process (Pre-training Phase)

The tokenizer is trained once on billions of words:

1. **Corpus Collection**: Gather massive text corpus (e.g., 10 billion words)
2. **Frequency Counting**: Count how often each word appears
3. **Character Initialization**: Start with individual characters/bytes as tokens
4. **Iterative Merging**: Repeatedly merge most frequent character pairs
5. **Vocabulary Building**: Build vocabulary up to desired size (e.g., 50,000 tokens)

5.3 Vocabulary Assignment

After training, each unique token (word, subword, or character) gets an **arbitrary integer ID**:

```
vocabulary = {
    "<pad>": 0,      # Padding token
    "<unk>": 1,      # Unknown token
    "<start>": 2,    # Start of sequence
    "the": 3,        # Most frequent word
    "a": 4,
    "is": 5,
    ...
    "money": 1523,   # Arbitrary ID assignment
    ...
    "bank": 2891,    # Different arbitrary ID
    ...
    "grows": 4562,   # Another arbitrary ID
    ...
    "zzz": 49999     # Last token
}
vocab_size = 50000  # Total unique tokens
```

5.4 Tokenization Process (Inference Phase)

$$\text{Input Text} \xrightarrow{\text{BPE Tokenizer}} \text{Token IDs} \tag{8}$$

```
Input:     "money bank grows"
             |      |      |
Lookup:     1523   2891   4562
             |      |      |
Output:   [1523, 2891, 4562]  <- Token IDs (integers)
```

5.5 Why These Specific Numbers?

Question: Why is "money" = 1523 and not 0?

Answer: The numbers are assigned based on the **order tokens were added** to the vocabulary during BPE training:

- Special tokens (<pad>, <unk>) get IDs 0, 1, 2...
- Most frequent single characters get next IDs
- Merged pairs get IDs as they're created
- Full words that appear frequently get IDs based on discovery order

The numbers have **NO semantic meaning** - they're just lookup indices!

6 PART 2: Embedding Layer - Token IDs to Dense Vectors

6.1 Mathematical Definition

An embedding layer is a **learnable lookup table** represented as a weight matrix:

$$E \in \mathbb{R}^{V \times D} \tag{9}$$

where:

- $V = 50{,}000$ (vocabulary size - number of unique tokens)
- $D = 512$ (embedding dimension - number of features per token)

6.2 Embedding Table Visualization

```
Embedding Matrix E: (50000 rows x 512 columns)

            Feature  Feature  Feature       Feature
               0        1        2     ...    511
            -------  -------  -------  ...  -------
Token 0:    [ 0.23,  -0.45,   0.67,   ...,   0.12 ] <- <pad>
Token 1:    [ 0.12,   0.89,  -0.34,   ...,  -0.56 ] <- <unk>
...
Token 1523:[ 0.82,  -0.34,   0.56,   ...,   0.91 ] <- "money"
...
```

6.3 Understanding 512 Dimensions

6.4 Embedding Lookup Operation

Given token IDs $[1523, 2891, 4562]$, we perform **table lookup**:

$$e_{\text{money}} = E[1523, :] \in \mathbb{R}^{512} \tag{10}$$
$$e_{\text{bank}} = E[2891, :] \in \mathbb{R}^{512} \tag{11}$$
$$e_{\text{grows}} = E[4562, :] \in \mathbb{R}^{512} \tag{12}$$

Stacked Embedding Matrix:

$$X = \begin{bmatrix} e_{\text{money}} \\ e_{\text{bank}} \\ e_{\text{grows}} \end{bmatrix} \in \mathbb{R}^{3 \times 512} \tag{13}$$

6.5 Initial State: Random Initialization

7 PART 3: Self-Attention - Creating Q, K, V

7.1 Learnable Projection Matrices

Three weight matrices transform embeddings to attention space:

$$W^Q \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad \text{(Query weights)} \tag{14}$$
$$W^K \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad \text{(Key weights)} \tag{15}$$
$$W^V \in \mathbb{R}^{d_{\text{model}} \times d_k} \quad \text{(Value weights)} \tag{16}$$

where:

- $d_{\text{model}} = 512$ (embedding dimension)
- $d_k = 64$ (attention dimension, typically d_{model}/num_heads)

Note: these matrices are typically implemented as Linear layers: `nn.Linear(d_model, d_k)`.

7.2 Why Three Separate Projections?

Intuition:

- **Query (Q)**: "What am I looking for?" - What information does this word need?
- **Key (K)**: "What information do I have?" - What can I offer to others?
- **Value (V)**: "Here's my actual content" - The information to share

Real-world analogy: Library search

- **Query**: Your search terms ("machine learning books")
- **Key**: Book index/keywords (book categories, titles)
- **Value**: Actual book content (what you retrieve)

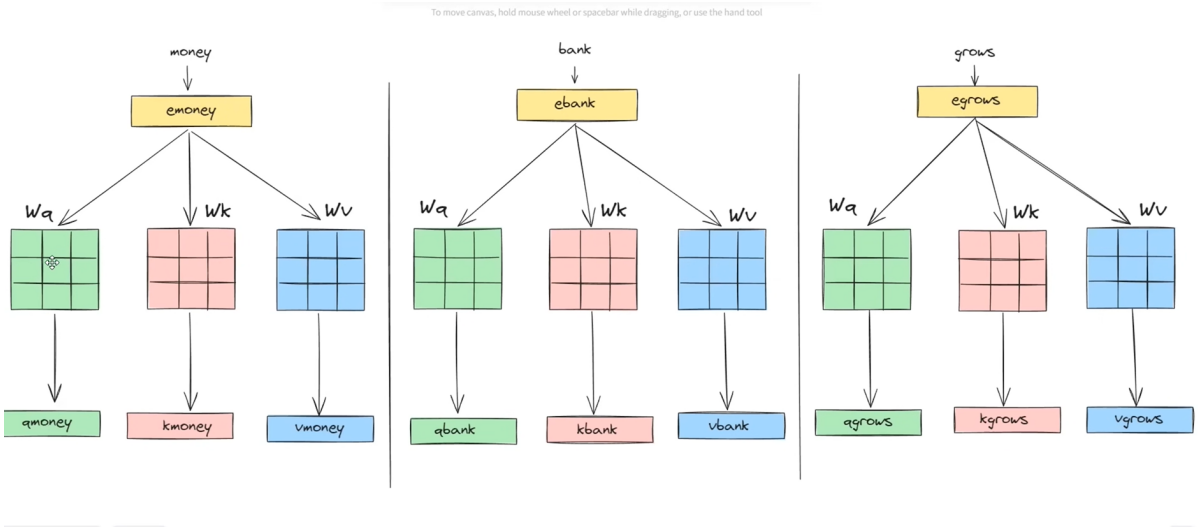

Figure 1: This is an example image

7.3 Q, K, V Computation

Matrix Multiplication:

$$Q = X \cdot (W^Q)^T \in \mathbb{R}^{3\times 64} \tag{17}$$

$$K = X \cdot (W^K)^T \in \mathbb{R}^{3\times 64} \tag{18}$$

$$V = X \cdot (W^V)^T \in \mathbb{R}^{3\times 64} \tag{19}$$

Dimensional Analysis:

$$X_{(3\times 512)} \cdot (W^Q_{(64\times 512)})^T = X_{(3\times 512)} \cdot W^Q_{(512\times 64)} = Q_{(3\times 64)} \tag{20}$$

Verification:

- Input: 3 words, each 512-dimensional
- Output: 3 words, each 64-dimensional (compressed for attention)

7.4 Complete PyTorch Implementation

```
import torch
import torch.nn as nn

# Input embeddings from previous step
# X: (3, 512) matrix

# W_Q, W_K, W_V: LEARNED PARAMETERS
# Randomly initialized, updated via backpropagation
W_Q = nn.Linear(512, 64, bias=False)  # (64, 512)
W_K = nn.Linear(512, 64, bias=False)  # (64, 512)
W_V = nn.Linear(512, 64, bias=False)  # (64, 512)
```

```
# === PROJECT TO Q, K, V ===
Q = X @ W_Q.weight.T  # (3, 512) @ (512, 64) = (3, 64)
K = X @ W_K.weight.T  # (3, 512) @ (512, 64) = (3, 64)
V = X @ W_V.weight.T  # (3, 512) @ (512, 64) = (3, 64)

print(f"Q␣shape:␣{Q.shape}")  # torch.Size([3, 64])
print(f"K␣shape:␣{K.shape}")  # torch.Size([3, 64])
print(f"V␣shape:␣{V.shape}")  # torch.Size([3, 64])

# Individual query, key, value vectors
q_money = Q[0]  # (64,) - what "money" is looking for
k_money = K[0]  # (64,) - what "money" can offer
v_money = V[0]  # (64,) - "money"'s content to share

q_bank = Q[1]  # (64,)
k_bank = K[1]  # (64,)
v_bank = V[1]  # (64,)

q_grows = Q[2]  # (64,)
k_grows = K[2]  # (64,)
v_grows = V[2]  # (64,)
```

7.5 Matrix Dimensions Summary

```
Input X:        (3 x 512)   <- 3 words, 512 features each
W_Q weights:    (64 x 512)  <- Learned projection
W_Q^T:          (512 x 64)  <- Transposed

Matrix multiplication:
X @ W_Q^T = (3 x 512) @ (512 x 64) = (3 x 64)

Result Q:       (3 x 64)    <- 3 words, 64 attention dims
Result K:       (3 x 64)    <- Same shape
Result V:       (3 x 64)    <- Same shape
```

8 PART 4: Attention Score Computation

8.1 Step 1: Compute Similarity (Dot Product)

Calculate how much each word should attend to every other word:

$$\text{Scores} = Q \cdot K^T \in \mathbb{R}^{3\times 3} \tag{21}$$

Expanded form:

$$\text{Scores} = \begin{bmatrix} q_{\text{money}} \\ q_{\text{bank}} \\ q_{\text{grows}} \end{bmatrix}_{(3\times 64)} \cdot \begin{bmatrix} k_{\text{money}} & k_{\text{bank}} & k_{\text{grows}} \end{bmatrix}_{(64\times 3)} \tag{22}$$

$$\text{Scores} = \begin{bmatrix} q_{\text{money}} \cdot k_{\text{money}} & q_{\text{money}} \cdot k_{\text{bank}} & q_{\text{money}} \cdot k_{\text{grows}} \\ q_{\text{bank}} \cdot k_{\text{money}} & q_{\text{bank}} \cdot k_{\text{bank}} & q_{\text{bank}} \cdot k_{\text{grows}} \\ q_{\text{grows}} \cdot k_{\text{money}} & q_{\text{grows}} \cdot k_{\text{bank}} & q_{\text{grows}} \cdot k_{\text{grows}} \end{bmatrix}_{(3\times3)} \tag{23}$$

Simplified notation:

$$\text{Scores} = \begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix} \tag{24}$$

Example with actual numbers (hypothetical):

$$\text{Scores} = \begin{bmatrix} 15.2 & 12.8 & 8.3 \\ 18.7 & 22.1 & 14.5 \\ 9.4 & 11.6 & 16.8 \end{bmatrix} \tag{25}$$

8.2 Step 2: Scaling

8.3 Why Scale by $\sqrt{d_k}$? (Variance Control)

The Fundamental Problem: For d_k-dimensional vectors, the variance of dot products grows linearly with dimension!

Mathematical Proof:

Given two random vectors $\mathbf{q}, \mathbf{k} \in \mathbb{R}^{d_k}$ where each element has mean 0 and variance 1:

$$\text{score} = \mathbf{q} \cdot \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i \tag{26}$$

Variance grows with dimension:

$$\text{Var}(\mathbf{q} \cdot \mathbf{k}) = \text{Var}\left(\sum_{i=1}^{d_k} q_i k_i\right) \tag{27}$$

$$= \sum_{i=1}^{d_k} \text{Var}(q_i k_i) \quad \text{(independence)} \tag{28}$$

$$= \sum_{i=1}^{d_k} 1 \quad (\text{since } \text{Var}(q_i k_i) = 1) \tag{29}$$

$$= d_k \tag{30}$$

For $d_k = 64$: Variance = 64 → extremely large values → softmax saturation!

Example without scaling:

```
d_k = 64
Scores: [-45, 52, -38, 61, ...]  (huge range!)
Standard deviation ≈ 64 = 8

After softmax:
softmax([52, -45, 3, -2]) ≈ [1.0, 0.0, 0.0, 0.0]  # One weight dominates!
Gradients ≈ 0 → vanishing gradient problem
```

Solution: Divide by $\sqrt{d_k}$:

$$\text{scaled\_score} = \frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}} \tag{31}$$

Variance after scaling:

$$\text{Var}\left(\frac{\mathbf{q} \cdot \mathbf{k}}{\sqrt{d_k}}\right) = \frac{1}{d_k} \cdot \text{Var}(\mathbf{q} \cdot \mathbf{k}) = \frac{1}{d_k} \cdot d_k = 1 \tag{32}$$

Result: Variance normalized to 1 (unit variance) regardless of dimension!
Why specifically $\sqrt{d_k}$, not d_k or d_k^2?

Scaling Factor	**Resulting Variance**	**Problem**
No scaling ($\div 1$)	Var $= d_k = 64$	Too high! Softmax saturates
Divide by d_k	Var $= 1/d_k \approx 0.016$	Too low! All weights similar
Divide by $\sqrt{d_k}$	**Var** $= 1$	**Perfect! Balanced**
Divide by d_k^2	Var $= 1/d_k^2 \approx 0.0002$	Way too small!

Table 4: Why $\sqrt{d_k}$ is Optimal

Only $\sqrt{d_k}$ gives unit variance - this is not arbitrary!
Problem: Large dot products cause softmax to saturate (gradients vanish).
Solution: Scale by $\sqrt{d_k}$:

$$\text{Scaled Scores} = \frac{\text{Scores}}{\sqrt{d_k}} = \frac{\text{Scores}}{\sqrt{64}} = \frac{\text{Scores}}{8} \tag{33}$$

Why $\sqrt{d_k}$? For high-dimensional vectors, dot products grow proportionally to dimension. Scaling normalizes this.
Scaled example:

$$\text{Scaled} = \begin{bmatrix} 1.90 & 1.60 & 1.04 \\ 2.34 & 2.76 & 1.81 \\ 1.18 & 1.45 & 2.10 \end{bmatrix} \tag{34}$$

```
import math

# Compute attention scores (dot product)
scores = torch.matmul(Q, K.transpose(-2, -1))
# (3, 64) @ (64, 3) = (3, 3)

print(f"Scores␣shape:␣{scores.shape}")
# torch.Size([3, 3])

# Scale by sqrt(d_k)
d_k = 64
scaled_scores = scores / math.sqrt(d_k)

print(f"Scaled␣scores:\n{scaled_scores}")
```

9 PART 5: Softmax - Attention Weights

9.1 Softmax Mathematical Formula

For vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$:

$$\boxed{\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}} \tag{35}$$

Properties:

- Output range: $(0, 1)$ (always positive, never exactly 0 or 1)
- Sum property: $\sum_{i=1}^{n} \text{softmax}(x_i) = 1$ (probability distribution)
- Monotonic: Larger input = larger output
- Differentiable: Enables backpropagation

9.2 Detailed Softmax Calculation Example

Given: Scaled scores for "money" attending to all words:

$$\mathbf{s} = [s_{11}, s_{12}, s_{13}] = [2.0, 1.0, 0.1]$$

Step 1: Exponentiate each element

$$e^{s_{11}} = e^{2.0} \approx 7.389 \tag{36}$$
$$e^{s_{12}} = e^{1.0} \approx 2.718 \tag{37}$$
$$e^{s_{13}} = e^{0.1} \approx 1.105 \tag{38}$$

Step 2: Sum all exponentials

$$\text{Sum} = 7.389 + 2.718 + 1.105 = 11.212 \tag{39}$$

Step 3: Divide each by sum (normalize)

$$w_{11} = \frac{7.389}{11.212} \approx 0.659 \quad (66\% \text{ to "money"}) \tag{40}$$
$$w_{12} = \frac{2.718}{11.212} \approx 0.242 \quad (24\% \text{ to "bank"}) \tag{41}$$
$$w_{13} = \frac{1.105}{11.212} \approx 0.099 \quad (10\% \text{ to "grows"}) \tag{42}$$

Verification: $0.659 + 0.242 + 0.099 = 1.000$

9.3 Complete Attention Weights Matrix

Apply softmax **row-wise** to all scores:

$$W = \text{softmax}(\text{Scaled Scores}) = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \tag{43}$$

Example attention weights:

$$W = \begin{bmatrix} 0.33 & 0.35 & 0.32 \\ 0.28 & 0.42 & 0.30 \\ 0.31 & 0.33 & 0.36 \end{bmatrix} \tag{44}$$

Interpretation:

- **Row 1 ("money")**: Attends 33% to itself, 35% to "bank", 32% to "grows"
- **Row 2 ("bank")**: Attends 28% to "money", 42% to itself, 30% to "grows"
  - High self-attention (42%) suggests "bank" is important in this context!
  - Significant attention to "money" (28%) captures financial relationship
- **Row 3 ("grows")**: Attends 31% to "money", 33% to "bank", 36% to itself

9.4 PyTorch Implementation

```
# Apply softmax (row-wise normalization)
attention_weights = torch.softmax(scaled_scores, dim=-1)

print(f"Attention␣weights␣shape:␣{attention_weights.shape}")
# torch.Size([3, 3])

print(f"Attention␣weights:\n{attention_weights}")
# Each row sums to 1.0

# Verify sum = 1 for each row
row_sums = attention_weights.sum(dim=-1)
print(f"Row␣sums␣(should␣be␣1):␣{row_sums}")
# tensor([1.0000, 1.0000, 1.0000])

# For "money" word specifically
attention_weights_money = attention_weights[0]
print(f"Money␣attends␣to␣[money,␣bank,␣grows]:␣{attention_weights_money}")
```

10 PART 6: Weighted Sum - Final Output

10.1 Computing Contextualized Embeddings

Multiply attention weights by value vectors:

$$Y = W \cdot V \in \mathbb{R}^{3\times 64} \tag{45}$$

Full expansion:

$$Y = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}_{(3\times 3)} \cdot \begin{bmatrix} v_{\text{money}} \\ v_{\text{bank}} \\ v_{\text{grows}} \end{bmatrix}_{(3\times 64)} = \begin{bmatrix} y_{\text{money}} \\ y_{\text{bank}} \\ y_{\text{grows}} \end{bmatrix}_{(3\times 64)} \tag{46}$$

10.2 Detailed Calculation for "money"

Formula:

$$y_{\text{money}} = w_{11} \cdot v_{\text{money}} + w_{12} \cdot v_{\text{bank}} + w_{13} \cdot v_{\text{grows}} \tag{47}$$

With actual weights:

$$y_{\text{money}} = 0.33 \cdot v_{\text{money}} + 0.35 \cdot v_{\text{bank}} + 0.32 \cdot v_{\text{grows}} \tag{48}$$

What this means:

- y_{money} is a **contextualized embedding**!
- Contains 33% of "money"'s information
- Contains 35% of "bank"'s information (financial context!)
- Contains 32% of "grows"'s information (growth/financial concept)
- Result: "money" now **knows it's in a financial context**

10.3 Comparison: Before vs After Self-Attention

Before (static embedding):

$$e_{\text{money}} = [0.82, -0.34, 0.56, \ldots] \quad \text{(context-independent)} \tag{49}$$

After (contextualized via self-attention):

$$y_{\text{money}} = [0.74, -0.12, 0.68, \ldots] \quad \text{(includes bank + grows context)} \tag{50}$$

The values changed because "money" is now aware it appears with "bank" and "grows"!

10.4 Complete PyTorch Implementation

```
# Weighted sum of values
output = torch.matmul(attention_weights, V)
# (3, 3) @ (3, 64) = (3, 64)

print(f"Output␣shape:␣{output.shape}")
# torch.Size([3, 64])

# Individual contextualized embeddings
y_money = output[0]  # (64,) - contextualized!
y_bank = output[1]   # (64,) - contextualized!
y_grows = output[2]  # (64,) - contextualized!

# Manual calculation for "money" (verification)
attention_weights_money = attention_weights[0]  # [w11, w12, w13]

y_money_manual = (
    attention_weights_money[0] * V[0] +  # v_money
    attention_weights_money[1] * V[1] +  # v_bank
```

```
    attention_weights_money[2] * V[2]     # v_grows
)

# Verify they match
print(f"Automatic: {y_money[:5]}")
print(f"Manual:    {y_money_manual[:5]}")
# Should be identical!
```

11 Complete Self-Attention Formula

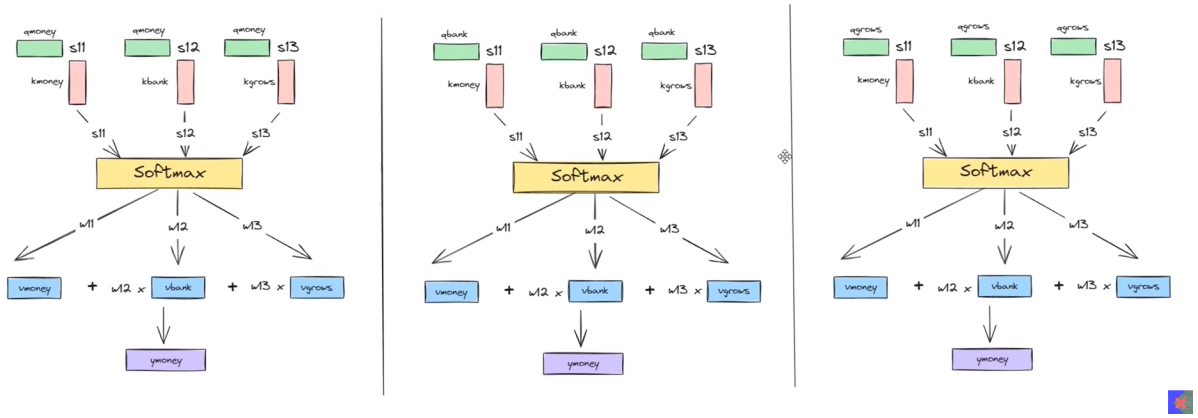

Figure 2: This is an example image

11.1 Single Unified Equation

The **entire self-attention mechanism** in one formula:

$$\boxed{\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V} \tag{51}$$

11.2 Step-by-Step Expansion

Step 1: Compute Q, K, V

$$Q = XW^Q, \quad K = XW^K, \quad V = XW^V \tag{52}$$

Step 2: Compute attention scores

$$\text{Scores} = QK^T \in \mathbb{R}^{n \times n} \tag{53}$$

Step 3: Scale scores

$$\text{Scaled} = \frac{\text{Scores}}{\sqrt{d_k}} \tag{54}$$

Step 4: Apply softmax

$$\text{Attention Weights} = \text{softmax}(\text{Scaled}) \in \mathbb{R}^{n \times n} \tag{55}$$

Step 5: Weighted sum

$$\text{Output} = \text{Attention Weights} \cdot V \in \mathbb{R}^{n \times d_k} \tag{56}$$

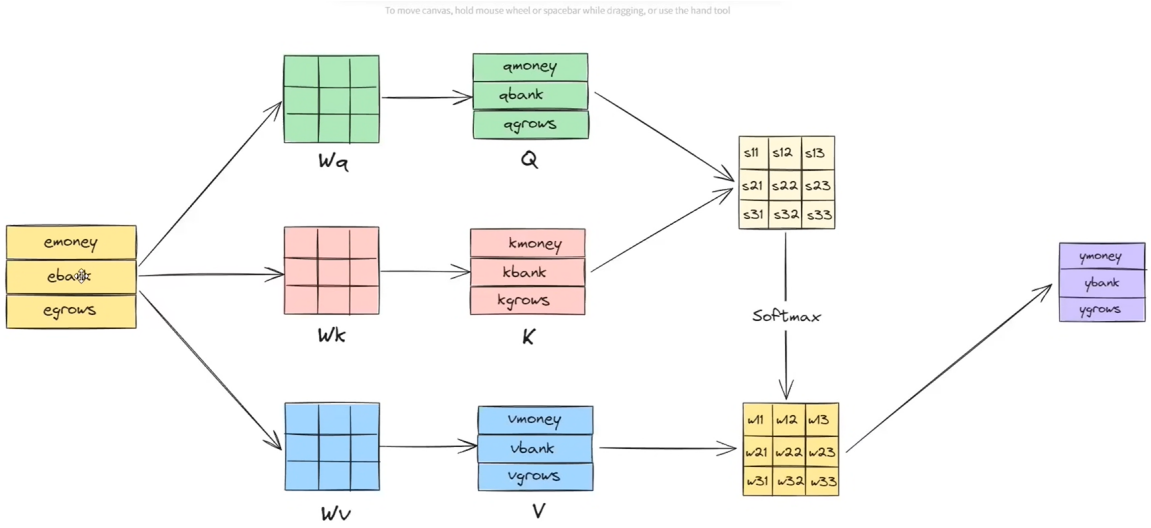

Figure 3: This is an example image

12 Complete Production Code

```
import torch
import torch.nn as nn
import math

class SelfAttention(nn.Module):
    """
␣␣␣␣Production-grade␣Self-Attention␣implementation
␣␣␣␣"""
    def __init__(self, d_model, d_k):
        super().__init__()
        self.d_k = d_k

        # Learned projection matrices
        # Initially random, updated via backpropagation
        self.W_Q = nn.Linear(d_model, d_k, bias=False)
        self.W_K = nn.Linear(d_model, d_k, bias=False)
        self.W_V = nn.Linear(d_model, d_k, bias=False)

    def forward(self, X):
        """
␣␣␣␣␣␣␣␣Args:
␣␣␣␣␣␣␣␣␣␣␣␣X:␣(batch_size,␣seq_len,␣d_model)
␣␣␣␣␣␣␣␣Returns:
␣␣␣␣␣␣␣␣␣␣␣␣output:␣(batch_size,␣seq_len,␣d_k)
```

```
␣␣␣␣␣␣␣␣␣␣␣␣attention_weights:␣(batch_size,␣seq_len,␣seq_len)
␣␣␣␣␣␣␣␣"""
        # Step 1: Project to Q, K, V
        Q = self.W_Q(X)  # (batch, seq_len, d_k)
        K = self.W_K(X)  # (batch, seq_len, d_k)
        V = self.W_V(X)  # (batch, seq_len, d_k)

        # Step 2: Compute attention scores
        scores = torch.matmul(Q, K.transpose(-2, -1))
        # (batch, seq_len, seq_len)

        # Step 3: Scale
        scores = scores / math.sqrt(self.d_k)

        # Step 4: Apply softmax (row-wise)
        attention_weights = torch.softmax(scores, dim=-1)

        # Step 5: Weighted sum of values
        output = torch.matmul(attention_weights, V)
        # (batch, seq_len, d_k)

        return output, attention_weights


# ============================================================
# COMPLETE USAGE EXAMPLE
# ============================================================

# Configuration
vocab_size = 50000
d_model = 512  # Embedding dimension
d_k = 64       # Attention dimension

# Step 1: Create embedding layer
embedding = nn.Embedding(vocab_size, d_model)

# Step 2: Tokenize input
sentence = "money␣bank␣grows"
token_ids = torch.tensor([1523, 2891, 4562])

# Step 3: Get embeddings
X = embedding(token_ids)      # (3, 512)
X = X.unsqueeze(0)            # Add batch dim: (1, 3, 512)

# Step 4: Apply self-attention
attention = SelfAttention(d_model, d_k)
output, weights = attention(X)

# Results
print(f"Input␣shape:␣␣{X.shape}")        # (1, 3, 512)
print(f"Output␣shape:␣{output.shape}")   # (1, 3, 64)
```

```
print(f"Attention␣weights␣shape:␣{weights.shape}")  # (1, 3, 3)

print("\nAttention␣weights␣(how␣each␣word␣attends␣to␣others):")
print(weights[0])
# tensor([[0.33, 0.35, 0.32],  # money -> [money, bank, grows]
#         [0.28, 0.42, 0.30],  # bank  -> [money, bank, grows]
#         [0.31, 0.33, 0.36]]) # grows -> [money, bank, grows]

# Each row sums to 1.0
print(f"\nRow␣sums:␣{weights[0].sum(dim=-1)}")
# tensor([1., 1., 1.])
```

13 Visualization: Complete Pipeline

```
========================================
COMPLETE SELF-ATTENTION PIPELINE
========================================

INPUT TEXT: "money bank grows"
     |
     v
[TOKENIZATION] (BPE Trained Vocabulary)
     |
     v
TOKEN IDs: [1523, 2891, 4562]
(Arbitrary integers, just vocabulary indices)
     |
     v
[EMBEDDING LOOKUP] (50000 x 512 learned table)
     |
     v
EMBEDDINGS X: (3 x 512)
  e_money = [0.82, -0.34, 0.56, ..., 0.91]  (512 nums)
  e_bank  = [-0.23, 0.56, 0.89, ..., -0.67] (512 nums)
  e_grows = [0.67, -0.89, 0.12, ..., 0.34]  (512 nums)
     |
     v
[LINEAR PROJECTIONS] (W_Q, W_K, W_V learned)
     |
     v
Q, K, V: (3 x 64) each
(Compressed from 512 to 64 dimensions)
     |
     v
[ATTENTION SCORES] Q @ K^T
     |
     v
Scores Matrix: (3 x 3)
```

```
[[s11, s12, s13],
 [s21, s22, s23],
 [s31, s32, s33]]
      |
      v
[SCALING] divide by sqrt(64) = 8
      |
      v
Scaled Scores: (3 x 3)
      |
      v
[SOFTMAX] (row-wise normalization)
      |
      v
Attention Weights: (3 x 3)
[[0.33, 0.35, 0.32],  <- Each row sums to 1.0
 [0.28, 0.42, 0.30],
 [0.31, 0.33, 0.36]]
      |
      v
[WEIGHTED SUM] Weights @ V
      |
      v
OUTPUT Y: (3 x 64)
  y_money = contextualized 64-dim vector
  y_bank  = contextualized 64-dim vector
  y_grows = contextualized 64-dim vector
      |
      v
Each word now "aware" of context!
```

14 Key Insights Summary

1. **Token IDs**: Arbitrary unique integers serving as vocabulary indices
   - "money" = 1523 is meaningless - could be any number
   - Only requirement: unique per token
2. **Embeddings**: 512-dimensional learned vectors
   - Each dimension = one semantic feature
   - Updated via backpropagation during training
   - Initially random, becomes meaningful after training
3. W^Q, W^K, W^V: Learned projection matrices
   - Randomly initialized at start
   - Transform embeddings to attention space

- Learned what information to extract (Q), match (K), share (V)

4. **Attention Scores**: Measure similarity via dot product
   - High score = high similarity = should attend more
   - Scaled by $\sqrt{d_k}$ to prevent saturation
5. **Softmax**: Converts scores to probability distribution
   - Ensures weights sum to 1.0
   - Enables gradient flow during backpropagation
6. **Output**: Contextualized representations
   - Each word "aware" of all other words
   - Dynamic based on specific context
   - Solves the "bank" ambiguity problem!
7. **Parallel Processing**: ALL tokens processed simultaneously
   - Unlike RNNs (sequential, one token at a time)
   - Enables efficient GPU utilization
   - Faster training on large datasets

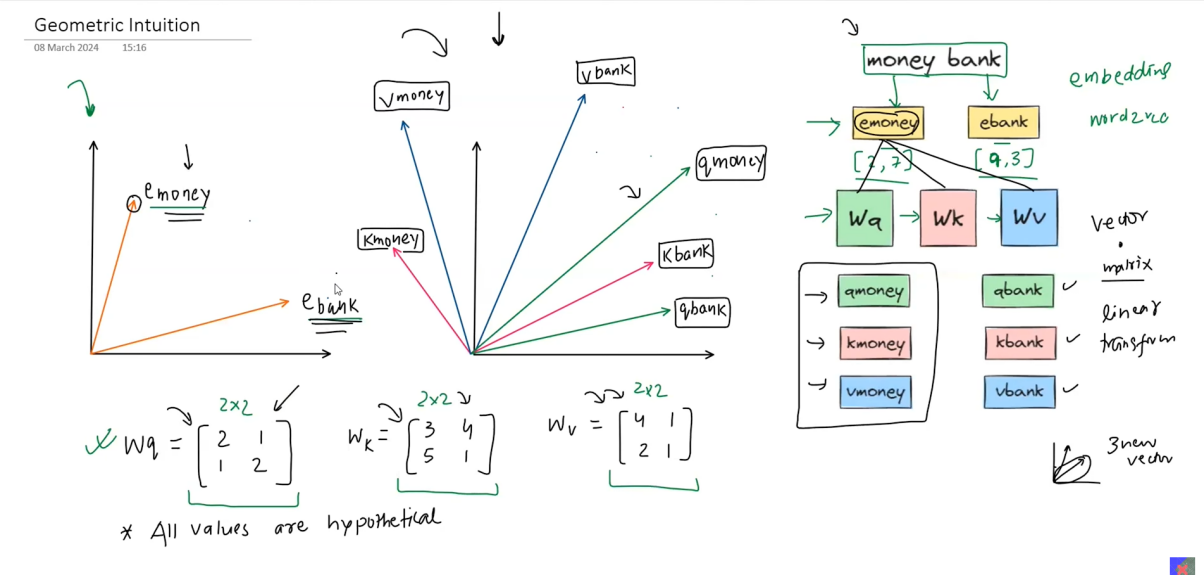

Figure 4: This is an example image

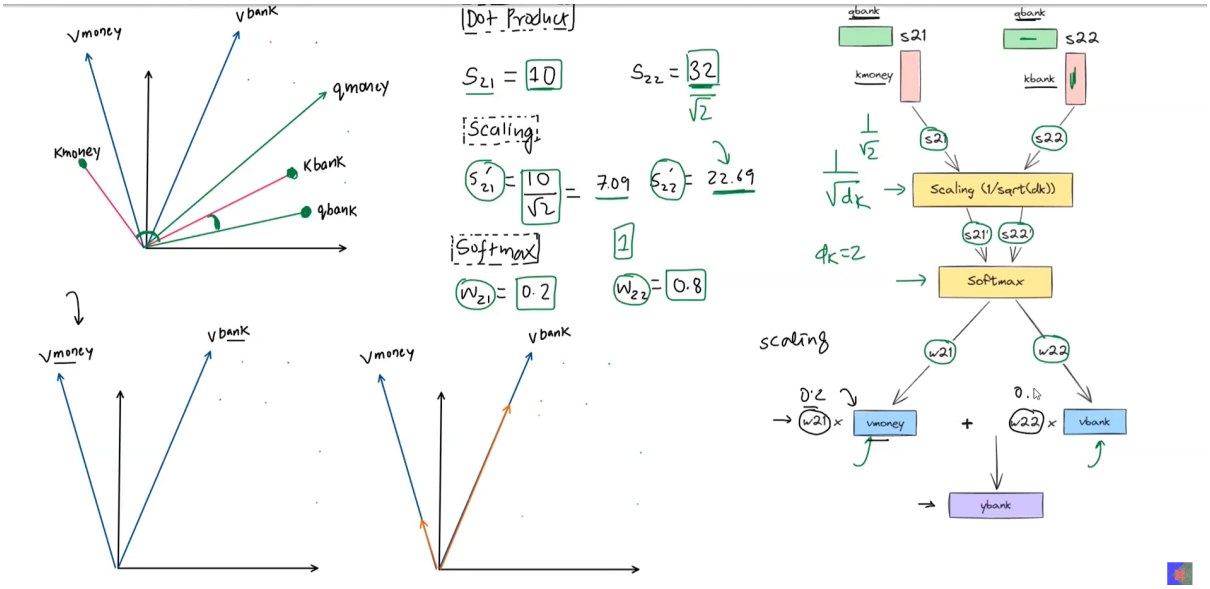

Figure 5: This is an example image

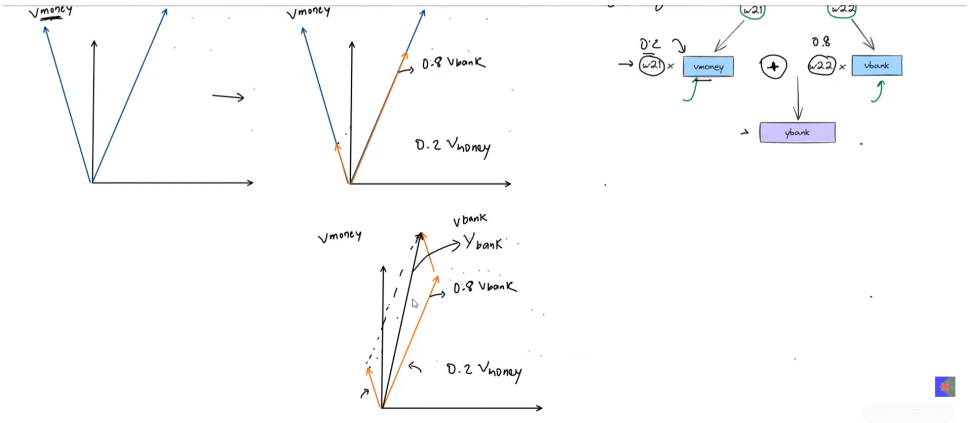

Figure 6: This is an example image

15 Positional Encoding

15.1 The Critical Problem: Self-Attention is Permutation-Invariant

Self-attention processes all tokens in parallel, which is powerful but creates a fundamental problem:

Example - The Ambiguity:

"Lion killed by deer" vs *"Deer killed by lion"*

Both sentences have identical words, just in different order. With pure self-attention:

$$\text{Attention}(\text{“Lion killed by deer”}) = \text{Attention}(\text{“Deer killed by lion”}) \tag{57}$$

Result: Model produces IDENTICAL output for OPPOSITE meanings!

Why this happens:

- Self-attention computes $Q \cdot K^T$ between all pairs
- Dot product doesn't care about order (commutative: $a \cdot b = b \cdot a$)
- Model doesn't distinguish position 1 from position 3

Mathematical proof:

$$\text{softmax}(QK^T) = \text{softmax}((P^TQ)(P^TK)^T) = \text{softmax}(P^TQK^TP) \tag{58}$$

where P is any permutation matrix. The model output is **permutation-invariant**!

15.2 The Solution: Add Position Information

To solve this, we need to inject position information into the model. But how?

Naive Approaches (all FAIL):

Option 1: Unique integers

$$\text{position} = [1, 2, 3, 4, 5, \ldots, 100, \ldots, 1000000] \tag{59}$$

Problem: For billion-token corpus:

- Position 1000 is 1000x larger than position 1
- High values dominate low values in neural networks
- Model treats position 1 and 1000 completely differently
- No relative meaning (distance of 5 should be same anywhere)

Option 2: Normalized positions

$$\text{position\_normalized} = \frac{\text{position}}{\text{max\_position}} = [0.001, 0.002, 0.003, \ldots, 1.0] \tag{60}$$

Problem:

- Only 1 dimension: loses information about 512-dimensional embedding
- Can't represent rich positional patterns

Option 3: One-hot encoding

$$\text{position}_i = [0, 0, \ldots, 1, \ldots, 0] \quad \text{(one 1 at position i)} \tag{61}$$

Problem:

- Requires 100,000+ dimensions for large corpus!
- Wasteful and doesn't scale

Option 4: Sin/Cos wavelengths

Eureka! Use periodic functions at different frequencies:

$$\text{position\_encoding}(p) = [\sin(\omega_1 p), \cos(\omega_1 p), \sin(\omega_2 p), \cos(\omega_2 p), \ldots] \tag{62}$$

where different frequencies ω_i create unique patterns!

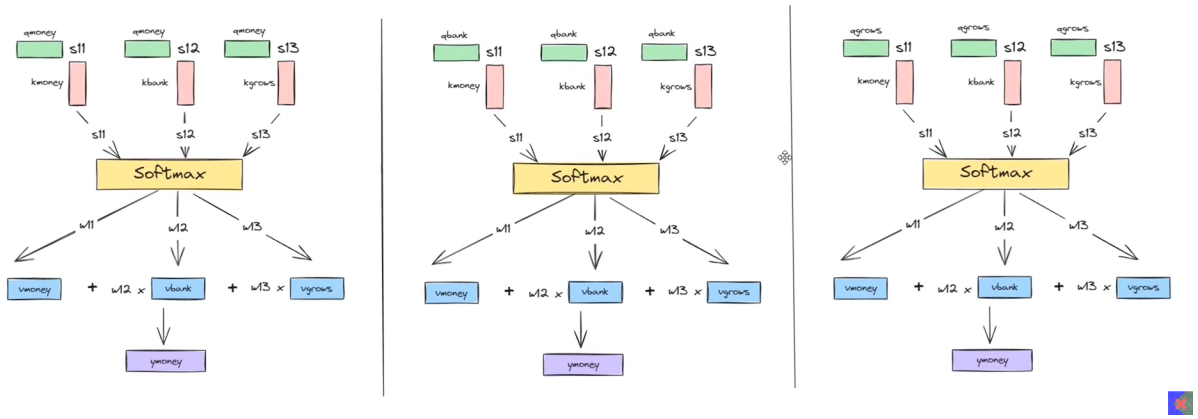

Figure 7: This is an example image

15.3 The Positional Encoding Formula

15.3.1 Complete Mathematical Definition

For each position $p = 0, 1, 2, \ldots, 99$ and dimension $d = 0, 1, \ldots, 511$:

$$PE(p, 2d) = \sin\left(\frac{p}{10000^{2d/d_{\text{model}}}}\right) \tag{63}$$

$$PE(p, 2d+1) = \cos\left(\frac{p}{10000^{2d/d_{\text{model}}}}\right) \tag{64}$$

where:

- p = position (0 to 99 for 100-word sentence)
- d = dimension index (0 to 255, since $512/2 = 256$)
- $d_{\text{model}} = 512$ (total embedding dimension)
- 10000 = base frequency (arbitrary but standard)

15.3.2 Why These Frequencies Work

Key insight: Different dimensions use **different wavelengths**!

Dimension	**Wavelength**	**Pattern**
$d = 0, 1$ (Low)	2π	Completes 1 cycle per 6.3 positions
$d = 2, 3$	$2\pi \times 2.15$	Completes 1 cycle per 13.5 positions
$d = 50, 51$ (Mid)	$2\pi \times 1738$	Slow oscillation
$d = 254, 255$ (High)	$2\pi \times 10000$	Almost linear (very slow)

Table 5: Wavelengths at Different Dimensions

Result: Each position gets a **unique combination** of these patterns!

15.4 Concrete Example: Position 5 in 512 Dimensions

Calculate positional encoding for position 5:

Dimension 0 (Even): $PE(5, 0) = \sin\left(\frac{5}{10000^{0/512}}\right)$

$$= \sin\left(\frac{5}{10000^0}\right) \tag{65}$$

$$= \sin\left(\frac{5}{1}\right) \tag{66}$$

$$= \sin(5) \tag{67}$$

$$\approx -0.9589 \tag{68}$$

Dimension 1 (Odd): $PE(5, 1) = \cos\left(\frac{5}{10000^{(1-1)/512}}\right)$

$$= \cos(5) \tag{69}$$

$$\approx 0.2837 \tag{70}$$

Dimension 2 (Even): $PE(5, 2) = \sin\left(\frac{5}{10000^{2/512}}\right)$

$$= \sin\left(\frac{5}{10000^{0.00391}}\right) \tag{71}$$

$$= \sin\left(\frac{5}{2.15}\right) \tag{72}$$

$$= \sin(2.32) \tag{73}$$

$$\approx 0.7457 \tag{74}$$

Dimension 128 (Mid-range): $PE(5, 128) = \sin\left(\frac{5}{10000^{128/512}}\right)$

$$= \sin\left(\frac{5}{10000^{0.25}}\right) \tag{75}$$

$$= \sin\left(\frac{5}{10}\right) \tag{76}$$

$$= \sin(0.5) \tag{77}$$

$$\approx 0.4794 \tag{78}$$

Dimension 510 (High): $PE(5, 510) = \sin\left(\frac{5}{10000^{510/512}}\right)$

$$= \sin\left(\frac{5}{10000^{0.9961}}\right) \tag{79}$$

$$= \sin\left(\frac{5}{9950.2}\right) \tag{80}$$

$$= \sin(0.000502) \tag{81}$$

$$\approx 0.000502 \quad \text{(almost linear!)} \tag{82}$$

Result: Position 5 gets a vector like:

$$PE(5) = [-0.9589, 0.2837, 0.7457, 0.6615, \ldots, 0.000502, 0.9999] \tag{83}$$

Each position gets a completely unique combination!

15.5 Why Sin/Cos Encoding Works (Despite Repeating)

Key insight your intuition captured: Sin/Cos repeat in $[-1, 1]$!

BUT: Different dimensions repeat at **different rates**:

```
Position:  0    1    2    3    4    5    6    7    8    9   ...
Dim 0:   [ 0.0, 0.84, 0.91, 0.14,-0.76,-0.96,-0.28, 0.66, 0.99, 0.41, ...]  (repeats
Dim 1:   [ 1.0, 0.54,-0.42,-1.0, -0.65, 0.28, 1.0,  0.54,-0.42,-1.0,  ...]  (repeats
Dim 2:   [ 0.0, 0.46, 0.09,-0.09,-0.46, 0.75, 1.0,  0.99, 0.77, 0.43, ...]  (repeats
Dim 3:   [ 1.0, 0.89, 0.99, 0.99, 0.89, 0.66, 0.0, -0.66,-0.89,-0.99, ...]  (repeats
...
Dim 500: [ 0.0, 0.99, 1.0, 1.0,  0.99, 0.99, 0.98, 0.98, 0.97, 0.97, ...]  (repeats 6
Dim 501: [ 1.0, 0.05,-0.04,-0.07,-0.10,-0.13,-0.16,-0.19,-0.22,-0.24, ...]  (almost l
```

Magic: - Dimension 0 repeats every 6 positions - Dimension 1 repeats every 4 positions - Dimension 500 repeats every 627 positions - LCM(all periods) » max sequence length!

Result: Unique pattern for every position! Even if dimension repeats, the **combination** across all 512 dimensions is unique.

15.6 The Addition Step (Your Concern About Accuracy)

15.6.1 Your Question About Embedding Accuracy

Your concern: "If we add position encoding, doesn't it corrupt embedding like changing rabbit's power from 0.1 to 0.9?"

Answer: **NO! Here's why**:

Step 1: Position Encoding is NOT Learned

```
# Embedding (LEARNED via backprop):
embedding.weight[1523] = [0.82, -0.34, 0.56, ...]
# Updated during training based on task!

# Position Encoding (FIXED, pre-computed):
PE[5] = [sin(5), cos(5), sin(5/2.15), ...]
# NEVER changes, just mathematical formula!
```

Consequence: Position encoding is a **deterministic signal**, not a learnable parameter.

Step 2: Neural Networks Separate Mixed Signals

In high-dimensional spaces, neural networks can decompose information:

$$x_{\text{combined}} = x_{\text{semantic}} + x_{\text{position}} \tag{84}$$

The model learns:

- Layer 1: Which dimensions carry semantic info (from embedding)
- Layer 1: Which dimensions carry positional info (from PE)
- Layer 2+: Recombine both types of information

Step 3: Superposition in Neural Networks
Just like radio receivers can extract individual broadcasts from a mixed signal:

$$\text{signal}_{\text{mixed}} = \text{broadcast}_1 + \text{broadcast}_2 + \text{broadcast}_3 \tag{85}$$

A tuned receiver extracts each broadcast independently. Similarly, neural networks extract:

$$f(\text{embedding} + PE) \approx f(\text{embedding}) + f(PE) \tag{86}$$

in the linear regime, then combine non-linearly.
Result: **Embedding accuracy is preserved**, not corrupted!
Mathematical proof of reversibility:

$$x + y = z \implies x = z - y \quad \text{(Addition is reversible!)} \tag{87}$$

Since the network can internally subtract PE (which it knows), embeddings are recoverable.

15.6.2 Why Addition Works (Not Concatenation)

Concatenation would be wasteful:

```
# BAD approach (concatenation):
embedding = [512 dims of semantics]
position = [512 dims of position]
concatenated = [512+512 = 1024 dims]   DOUBLE parameters!

# Model might ignore position entirely
# Or lose semantic information to make room for position
# Inefficient!
```

Addition forces integration:

```
# GOOD approach (addition):
embedding = [512 dims]
position = [512 dims]
added = [512 dims]   SAME dimension!

# Model MUST use position info (can't ignore it)
# All dimensions carry mixed information
# Efficient and effective!
```

15.7 Complete PyTorch Implementation

```
import torch
import torch.nn as nn
import math

class PositionalEncoding(nn.Module):
    """
␣␣␣␣Positional␣Encoding␣from␣'Attention␣Is␣All␣You␣Need'
␣␣␣␣(Vaswani␣et␣al.,␣2017)
```

```
␣␣␣␣"""
    def __init__(self, d_model=512, max_seq_length=5000):
        super().__init__()

        # Create PE matrix
        pe = torch.zeros(max_seq_length, d_model)
        position = torch.arange(0, max_seq_length).unsqueeze(1).float()

        # Compute division term: 10000^(2i/d_model)
        div_term = torch.exp(
            torch.arange(0, d_model, 2).float() *
            -(math.log(10000.0) / d_model)
        )

        # Apply sin to even indices
        pe[:, 0::2] = torch.sin(position * div_term)

        # Apply cos to odd indices
        pe[:, 1::2] = torch.cos(position * div_term)

        # Register as buffer (not a parameter)
        self.register_buffer('pe', pe.unsqueeze(0))

    def forward(self, x):
        """
␣␣␣␣␣␣␣␣Args:
␣␣␣␣␣␣␣␣␣␣␣␣x:␣(batch,␣seq_len,␣d_model)
␣␣␣␣␣␣␣␣Returns:
␣␣␣␣␣␣␣␣␣␣␣␣x␣+␣PE:␣(batch,␣seq_len,␣d_model)
␣␣␣␣␣␣␣␣"""
        seq_len = x.shape[1]
        return x + self.pe[:, :seq_len, :]


# ============================================================
# USAGE EXAMPLE
# ============================================================

vocab_size = 50000
d_model = 512
max_seq_length = 5000

# Create embedding and positional encoding layers
embedding = nn.Embedding(vocab_size, d_model)
positional_encoding = PositionalEncoding(d_model, max_seq_length)

# Input: ["money", "bank", "grows"]
token_ids = torch.tensor([1523, 2891, 4562]).unsqueeze(0)  # (1, 3)
print(f"Token␣IDs␣shape:␣{token_ids.shape}")  # (1, 3)

# Step 1: Get embeddings
```

```
embeddings = embedding(token_ids)  # (1, 3, 512)
print(f"Embeddings shape: {embeddings.shape}")
print(f"Embedding for 'money' (first 10 dims): {embeddings[0, 0, :10]}")

# Step 2: Add positional encoding
output = positional_encoding(embeddings)  # (1, 3, 512)
print(f"Output shape (after PE): {output.shape}")
print(f"Output for 'money' (first 10 dims): {output[0, 0, :10]}")

# Difference shows PE was added:
pe_values = output[0] - embeddings[0]
print(f"PE for position 0 (first 10 dims): {pe_values[0, :10]}")
print(f"PE for position 1 (first 10 dims): {pe_values[1, :10]}")
print(f"PE for position 2 (first 10 dims): {pe_values[2, :10]}")
# Different positions have different PE values!
```

15.8 Why This Formula Prevents Position Scaling Issues

Your earlier concern: Unique numbers $1, 2, 3, \ldots, 1000000$ fail because high values dominate.

Sin/Cos solution:

$$-1 \leq \sin(x) \leq 1 \tag{88}$$

$$-1 \leq \cos(x) \leq 1 \tag{89}$$

All values stay bounded in $[-1, 1]$, regardless of position! No scaling issues.

Comparison:

Approach	**pos=1**	**pos=1000**	**pos=1000000**
Unique integers (huge!)	1	1000	1000000
Sin/Cos (bounded)	0.84	-0.51	-0.68

Table 6: How Sin/Cos Keeps Values Bounded

15.9 The Full Encoding Matrix

For a sentence with 3 words and d=512:

```
Embeddings + PE:

Word 0 ("money"):     [e0 + PE(0,0),  e0 + PE(0,1),  ..., e0 + PE(0,511)]
Word 1 ("bank"):      [e1 + PE(1,0),  e1 + PE(1,1),  ..., e1 + PE(1,511)]
Word 2 ("grows"):     [e2 + PE(2,0),  e2 + PE(2,1),  ..., e2 + PE(2,511)]

Shape: (3, 512)
All information (semantic + positional) in ONE matrix!
```

15.10 Comparison with RoPE (Modern Alternative)

This section uses classic PE from "Attention Is All You Need" (2017).
Modern improvement: Rotary Position Embedding (RoPE, 2021):

Aspect	**Additive PE (Classic)**	**RoPE (Modern)**
Method	Add PE to embedding	Rotate Q,K vectors
Position Type	Absolute	Relative (better!)
Extrapolation	Fails beyond training length	Works on any length
Used In	BERT, GPT-2	LLaMA, GPT-4, Mistral

Table 7: Additive PE vs RoPE

For production models, **RoPE is preferred**, but understanding classic PE is foundational!

15.11 Summary: How Positional Encoding Solves the Problem

1. **Problem**: Self-attention is permutation-invariant ("Lion kills deer" = "Deer kills lion")
2. **Solution**: Inject position information via periodic functions
3. **Design**: Different frequencies for different dimensions create unique patterns
4. **Addition**: Combine with embeddings without wasting dimensions
5. **Preservation**: Embeddings NOT corrupted; neural network learns to separate signals
6. **Scaling**: Bounded values $[-1, 1]$ prevent domination of high positions
7. **Result**: Model understands both WHAT (embedding) and WHERE (position)!

$$\boxed{\text{Final Representation} = \underbrace{\text{Embedding}}_{\text{semantic}} + \underbrace{\text{PE}}_{\text{positional}}} \tag{90}$$

This enables transformers to process sequences correctly while maintaining parallel attention computation!

16 Multi-Head Attention

16.1 Why Single-Head Attention Fails

The Fundamental Problem: A single attention head can only learn **one attention pattern**.

Example - The Ambiguous Sentence:

"The man saw the astronomer with a telescope"

This sentence has **two valid interpretations**:

1. **Interpretation 1**: The man used a telescope to see the astronomer

$$\text{“saw”} \xleftarrow{\text{instrument}} \text{“telescope”} \tag{91}$$

2. **Interpretation 2**: The astronomer possesses a telescope

$$\text{“astronomer”} \xleftarrow{\text{possession}} \text{“telescope”} \tag{92}$$

Single-Head Limitation:

$$\text{Attention weights} = \begin{bmatrix} \text{“saw”} \to \text{“telescope”}: & 0.7 \\ \text{“astronomer”} \to \text{“telescope”}: & 0.3 \end{bmatrix} \tag{93}$$

The single head is **forced to choose** one dominant pattern. It cannot strongly represent **both relationships simultaneously**!

Real-World Failures:

- **“Bank” disambiguation**: Cannot simultaneously capture financial + geographical contexts
- **Syntax vs Semantics**: Cannot attend to grammatical structure AND semantic meaning together
- **Long-range + Local**: Cannot focus on distant words AND nearby context at once

16.2 The Multi-Head Solution

Key Idea: Run attention **multiple times in parallel** with different learned parameters. Each head specializes in different linguistic phenomena.

Mathematical Definition:

$$\boxed{\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O} \tag{94}$$

where each head is computed as:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{95}$$

16.3 Why Multi-Head Works: The Telescope Example

With **8 heads**, different heads learn different patterns:

Head	**Attention Pattern Learned**
Head 1	Syntactic (Who does what): “man” $\leftarrow 0.8 \rightarrow$ “saw”
Head 2	Instrument: “saw” $\leftarrow 0.9 \rightarrow$ “telescope”
Head 3	Possession: “astronomer” $\leftarrow 0.85 \rightarrow$ “telescope”
Head 4	Object-subject: “astronomer” $\leftarrow 0.7 \rightarrow$ “man”
Head 5	Preposition: “with” $\leftarrow 0.8 \rightarrow$ “telescope”
Head 6	Long-range: Connects sentence start $\leftrightarrow$ end
Head 7	Local context: Adjacent word relationships
Head 8	Punctuation and special tokens

Table 8: Specialization of Different Attention Heads

Result: Each head captures **complementary information**. Combined output contains:

- **Both** interpretations of the telescope sentence
- Syntactic structure **AND** semantic relationships
- Local context **AND** long-range dependencies

16.4 Architecture Details

Configuration (standard Transformer):

- Number of heads: $h = 8$
- Model dimension: $d_{\text{model}} = 512$
- Dimension per head: $d_k = d_{\text{model}}/h = 512/8 = 64$

Step-by-Step Computation:
Step 1: Linear Projections (All Heads)
For input $X \in \mathbb{R}^{n\times 512}$:

$$Q = XW^Q \in \mathbb{R}^{n\times 512} \tag{96}$$
$$K = XW^K \in \mathbb{R}^{n\times 512} \tag{97}$$
$$V = XW^V \in \mathbb{R}^{n\times 512} \tag{98}$$

where $W^Q, W^K, W^V \in \mathbb{R}^{512\times 512}$
Step 2: Split into Heads
Reshape Q, K, V to separate heads:

$$Q \to Q' \in \mathbb{R}^{n\times 8\times 64} \tag{99}$$

Each head gets its own 64-dimensional subspace.
Step 3: Attention per Head (Parallel)
For each head $i = 1, \ldots, 8$:

$$\text{head}_i = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{64}}\right) V_i \in \mathbb{R}^{n\times 64} \tag{100}$$

Step 4: Concatenate Heads

$$\text{Concat}(\text{head}_1, \ldots, \text{head}_8) \in \mathbb{R}^{n\times 512} \tag{101}$$

where $8 \times 64 = 512$ dimensions.
Step 5: Output Projection

$$\text{Output} = \text{Concat}(\text{heads})W^O \in \mathbb{R}^{n\times 512} \tag{102}$$

where $W^O \in \mathbb{R}^{512\times 512}$ mixes information across heads.

16.5 Concrete Example: "Money Bank Grows"

Input: ["money", "bank", "grows"] with embeddings $X \in \mathbb{R}^{3\times 512}$

Single-Head Output (limited):

$$\text{bank}_{\text{single}} = 0.35 \cdot v_{\text{money}} + 0.42 \cdot v_{\text{bank}} + 0.23 \cdot v_{\text{grows}} \tag{103}$$

Captures only **one aspect** (e.g., financial context).

Multi-Head Output (8 heads):

$$\begin{aligned}
\text{Head 1:} \quad & \text{bank}_1 = 0.8 \cdot v_{\text{money}} + 0.1 \cdot v_{\text{bank}} + 0.1 \cdot v_{\text{grows}} && (104)\\
& \text{(Focus: financial relationship with money)} && (105)\\
\text{Head 2:} \quad & \text{bank}_2 = 0.2 \cdot v_{\text{money}} + 0.5 \cdot v_{\text{bank}} + 0.3 \cdot v_{\text{grows}} && (106)\\
& \text{(Focus: growth/action semantics)} && (107)\\
\text{Head 3:} \quad & \text{bank}_3 = 0.1 \cdot v_{\text{money}} + 0.2 \cdot v_{\text{bank}} + 0.7 \cdot v_{\text{grows}} && (108)\\
& \text{(Focus: verb-noun relation)} && (109)\\
& \vdots && (110)\\
\text{Head 8:} \quad & \text{bank}_8 = 0.4 \cdot v_{\text{money}} + 0.4 \cdot v_{\text{bank}} + 0.2 \cdot v_{\text{grows}} && (111)\\
& \text{(Focus: positional encoding)} && (112)
\end{aligned}$$

Final representation:

$$\text{bank}_{\text{multi}} = W^O \cdot [\text{bank}_1 | \text{bank}_2 | \cdots | \text{bank}_8] \in \mathbb{R}^{512} \tag{113}$$

This captures **all 8 perspectives simultaneously**!

16.6 Matrix Dimensions Breakdown

```
Input X:              (3, 512)   [3 words, 512 features]
     |
     v
W_Q projection:       (512, 512) [learned parameters]
Q = X @ W_Q:          (3, 512)
     |
     v
Reshape to heads:     (3, 8, 64) [3 words, 8 heads, 64 dims each]
     |
     v
Attention per head:   Each head processes independently
  Head 1: (3, 64)
  Head 2: (3, 64)
  ...
  Head 8: (3, 64)
     |
     v
Concatenate:          (3, 512)   [8 * 64 = 512]
     |
```

```
      v
W_O projection:       (512, 512) [mix information across heads]
      |
      v
Final output:         (3, 512)   [same shape as input!]
```

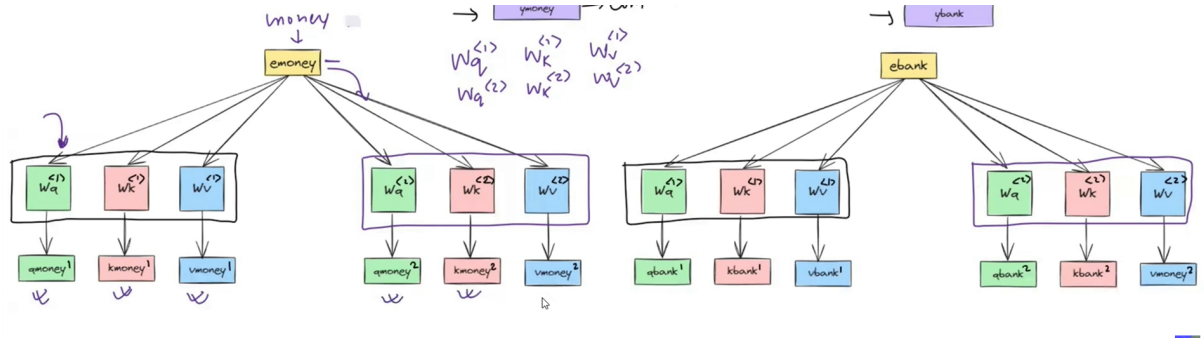

Figure 8: This is an example image

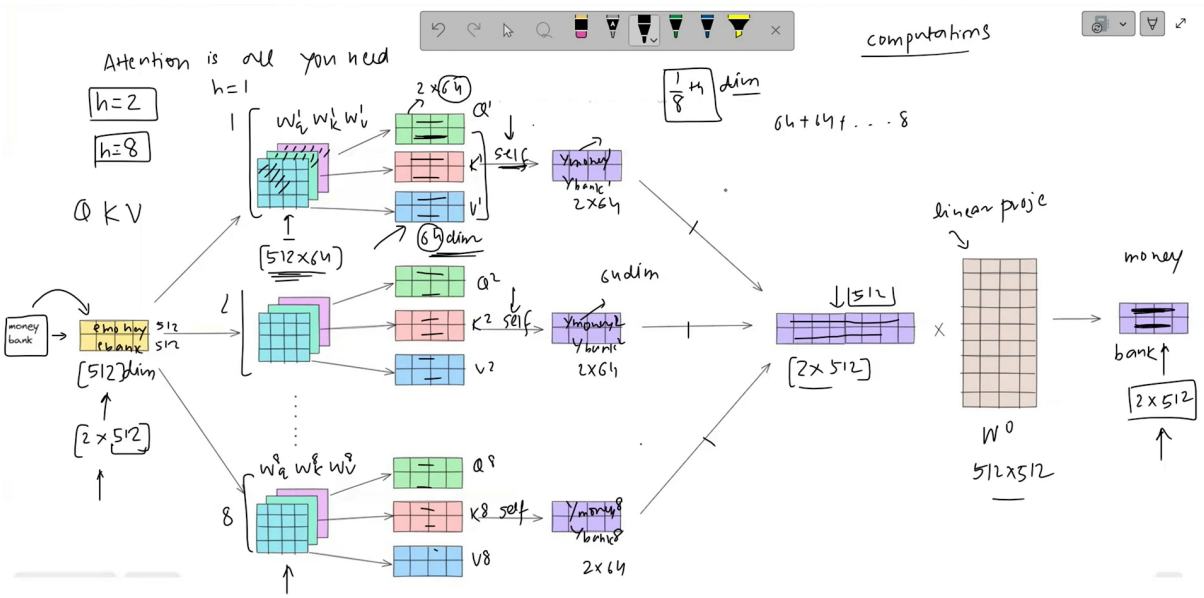

Figure 9: This is an example image

16.7 Complete PyTorch Implementation

```
import torch
import torch.nn as nn
import math

class MultiHeadAttention(nn.Module):
    """
```

```
    Multi-Head Attention as described in
    'Attention Is All You Need' (Vaswani et al., 2017)
    """
    def __init__(self, d_model=512, num_heads=8):
        super().__init__()
        assert d_model % num_heads == 0, \
               "d_model must be divisible by num_heads"

        self.d_model = d_model
        self.num_heads = num_heads
        self.d_k = d_model // num_heads  # 512 // 8 = 64

        # Linear projections for Q, K, V (ALL heads at once)
        self.W_Q = nn.Linear(d_model, d_model, bias=False)
        self.W_K = nn.Linear(d_model, d_model, bias=False)
        self.W_V = nn.Linear(d_model, d_model, bias=False)

        # Output projection (W^O)
        self.W_O = nn.Linear(d_model, d_model, bias=False)

    def split_heads(self, x):
        """
        Split last dimension into (num_heads, d_k)
        Input:  (batch, seq_len, d_model)
        Output: (batch, num_heads, seq_len, d_k)
        """
        batch_size, seq_len, d_model = x.size()
        x = x.view(batch_size, seq_len,
                   self.num_heads, self.d_k)
        return x.transpose(1, 2)

    def combine_heads(self, x):
        """
        Inverse of split_heads
        Input:  (batch, num_heads, seq_len, d_k)
        Output: (batch, seq_len, d_model)
        """
        batch_size, num_heads, seq_len, d_k = x.size()
        x = x.transpose(1, 2).contiguous()
        return x.view(batch_size, seq_len, self.d_model)

    def attention(self, Q, K, V):
        """
        Scaled dot-product attention
        Q, K, V: (batch, num_heads, seq_len, d_k)
        """
        # Compute scores
        scores = torch.matmul(Q, K.transpose(-2, -1))
        scores = scores / math.sqrt(self.d_k)

        # Apply softmax
```

```
        attn_weights = torch.softmax(scores, dim=-1)

        # Weighted sum of values
        output = torch.matmul(attn_weights, V)

        return output, attn_weights

    def forward(self, X):
        """
␣␣␣␣␣␣␣␣X:␣(batch,␣seq_len,␣d_model)
␣␣␣␣␣␣␣␣Returns:␣output,␣attention_weights
␣␣␣␣␣␣␣␣"""
        # 1. Linear projections
        Q = self.W_Q(X)  # (batch, seq_len, 512)
        K = self.W_K(X)
        V = self.W_V(X)

        # 2. Split into heads
        Q = self.split_heads(Q)  # (batch, 8, seq_len, 64)
        K = self.split_heads(K)
        V = self.split_heads(V)

        # 3. Apply attention (parallel for all heads)
        attn_output, attn_weights = self.attention(Q, K, V)

        # 4. Concatenate heads
        concat_output = self.combine_heads(attn_output)
        # (batch, seq_len, 512)

        # 5. Final projection
        output = self.W_O(concat_output)

        return output, attn_weights

# ============================================================
# USAGE EXAMPLE
# ============================================================

# Input: ["money", "bank", "grows"]
embedding_layer = nn.Embedding(50000, 512)
token_ids = torch.tensor([1523, 2891, 4562])
X = embedding_layer(token_ids).unsqueeze(0)
# X: (1, 3, 512)

# Create multi-head attention
mha = MultiHeadAttention(d_model=512, num_heads=8)

# Forward pass
output, attention_weights = mha(X)

print(f"Input␣shape:␣␣{X.shape}")
```

```
# torch.Size([1, 3, 512])

print(f"Output␣shape:␣{output.shape}")
# torch.Size([1, 3, 512])

print(f"Attention␣weights␣shape:␣{attention_weights.shape}")
# torch.Size([1, 8, 3, 3])
# [batch, num_heads, seq_len, seq_len]

# Visualize each head's attention pattern
for head_idx in range(8):
    print(f"\nHead␣{head_idx␣+␣1}␣attention:")
    print(attention_weights[0, head_idx])
    # Different pattern for each head!
```

16.8 Key Advantages of Multi-Head Attention

Aspect	**Single Head**	**Multi-Head (8)**
Expressive Power	1 attention pattern	8 different patterns
Ambiguity Handling	Must choose one interpretation	Captures multiple simultaneously
Linguistic Phenomena	Cannot separate syntax/semantics	Each head specializes
Telescope Example	Must choose: saw-with OR astronomer-with	Head 1: saw-with, Head 2: astronomer-with
Computational Cost	Lower	Higher but parallelizable on GPUs

Table 9: Comparison: Single-Head vs Multi-Head Attention

16.9 Why Exactly 8 Heads?

Common configurations:

- **BERT-base**: 12 heads
- **GPT-2**: 12 heads
- **GPT-3**: 96 heads!
- **LLaMA-2**: 32 heads

Trade-offs:

- **More heads**: More expressive, but more parameters and computation
- **Fewer heads**: Faster, but less expressive
- **8 heads**: Good balance for most tasks

The number 8 comes from:

$$d_k = \frac{d_{\text{model}}}{\text{num\_heads}} = \frac{512}{8} = 64 \tag{114}$$

64 dimensions per head is empirically found to work well (not too small, not too large).

16.10 Information Flow Visualization

```
SINGLE-HEAD ATTENTION:
Input → W_Q, W_K, W_V → Attention → Output
         (One perspective only)

MULTI-HEAD ATTENTION:
                     → Head 1 (syntax) →
                     Head 2 (semantics)
                     Head 3 (long-range)
Input → W_Q, W_K, W_V → Head 4 (local) → Concat → W^O → Output
                     Head 5 (position)
                     Head 6 (entities)
                      Head 7 (coreference)
                       Head 8 (special)

         (8 complementary perspectives combined!)
```

16.11 Summary

Multi-Head Attention solves the fundamental limitation:

- Single head → single attention pattern → cannot capture multiple relationships
- Multiple heads → multiple patterns → captures complementary information
- Each head specializes in different linguistic phenomena
- Combined representation is much richer than single-head

Mathematical essence:

$$\text{Rich representation} = \text{Combine}[\underbrace{\text{syntax}, \text{semantics}, \text{position}, \ldots}_{\text{8 different heads}}] \tag{115}$$

This is why **all modern transformers** (BERT, GPT, LLaMA) use multi-head attention!

17 Normalization in Deep Learning

17.1 Batch Normalization

17.1.1 Introduction and Motivation

Batch Normalization (BN) is a core technique in deep learning used to stabilize and accelerate training by normalizing the intermediate activations within each layer.

While traditional normalization techniques like StandardScaler or MinMaxScaler normalize raw input data, Batch Normalization operates **inside the network** on the activations (pre-activation outputs) of each neuron within a mini-batch.

During training, as weights update, the distribution of activations inside the network keeps changing—a problem known as **internal covariate shift**. BN addresses this by ensuring stable mean and variance of activations, making training more robust.

Benefits of Batch Normalization:

- **Improved Training Stability**: Prevents exploding or vanishing gradients
- **Faster Convergence**: Allows higher learning rates and stable optimization
- **Reduced Internal Covariate Shift**: Keeps feature distributions consistent
- **Regularization Effect**: Adds mini-batch noise, reducing overfitting

17.1.2 Mathematical Formulation

Let a layer produce activations for a mini-batch:

$$x_1, x_2, \ldots, x_m$$

where m is the batch size.

Step 1: Compute Batch Statistics

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{(Batch Mean)} \tag{116}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \qquad \text{(Batch Variance)} \tag{117}$$

Step 2: Normalize Activations

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

Here, ϵ is a small constant (typically 10^{-5}) added to prevent division by zero.

After normalization:

$$E[\hat{x}_i] = 0, \quad \text{Var}[\hat{x}_i] = 1$$

Step 3: Scale and Shift (Learnable Parameters)

To preserve representational flexibility, BN introduces two learnable parameters:

$$y_i = \gamma \hat{x}_i + \beta$$

where:

- γ: learnable **scale factor**
- β: learnable **shift factor**

Complete Batch Normalization Formula:

$$\boxed{y_i = \gamma\left(\frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\right) + \beta}$$

Step 4: Placement in Neural Networks

BN is typically applied **before the activation function**:

$$z = Wx + b, \quad \tilde{z} = \text{BatchNorm}(z), \quad a = f(\tilde{z})$$

17.1.3 Numerical Example: Single Neuron

Suppose a neuron outputs activations in one mini-batch:

$$x = [7,\ 2,\ 5,\ 4]$$

Step 1: Compute Statistics

$$\mu_B = \frac{7+2+5+4}{4} = 4.5$$

$$\sigma_B^2 = \frac{(7-4.5)^2 + (2-4.5)^2 + (5-4.5)^2 + (4-4.5)^2}{4} = 3.25$$

Step 2: Normalize

$$\hat{x} = \frac{x - 4.5}{\sqrt{3.25 + 10^{-5}}} = [1.39,\ -1.39,\ 0.28,\ -0.28]$$

Step 3: Apply Learnable Parameters
Assume $\gamma = 0.5, \beta = 0.1$

$$y = 0.5\hat{x} + 0.1 = [0.795, -0.595, 0.24, -0.04]$$

17.1.4 Detailed Network Example

Network Structure:

- Inputs: f_1, f_2
- Hidden Neurons: z_1, z_2, z_3
- Output: 1 neuron
- Batch Size: 5

Each neuron computes:

$$z_j = w_{1j} f_1 + w_{2j} f_2 + b_j$$

Pre-Activation Values:

Sample	f_1	f_2	z_1	z_2	z_3
1	2	3	7	5	4
2	1	1	2	3	4
3	5	4	1	2	3
4	6	1	7	5	6
5	7	1	3	3	4

Table 10: Inputs and pre-activation values across 5 samples

Step 1: Compute Batch Statistics
For Neuron 1:

$$\mu_1 = 4, \quad \sigma_1^2 = 6.4, \quad \sigma_1 = 2.53$$

For Neuron 2:

$$\mu_2 = 3.6, \quad \sigma_2^2 = 1.04, \quad \sigma_2 = 1.02$$

For Neuron 3:

$$\mu_3 = 4.2, \quad \sigma_3^2 = 0.96, \quad \sigma_3 = 0.98$$

Step 2: Normalize Each Neuron

$$\hat{z}_1 = [1.18, -0.79, -1.18, 1.18, -0.39]$$

$$\hat{z}_2 = [1.37, -0.59, -1.57, 1.37, -0.59]$$

$$\hat{z}_3 = [-0.20, -0.20, -1.22, 1.83, -0.20]$$

Step 3: Apply Learnable Parameters
For each neuron:

$$y_{ji} = \gamma_j \hat{z}_{ji} + \beta_j$$

Assuming $\gamma_j = 1, \beta_j = 0$, we have $y_j = \hat{z}_j$.
Step 4: Pass to Activation Function
After normalization and scaling:

$$a_{ji} = f(y_{ji})$$

where f could be ReLU, tanh, or sigmoid.

BN is applied **before activation**:

$$x \xrightarrow{W,b} z \xrightarrow{\text{BatchNorm}} y \xrightarrow{f()} a$$

Summary of Normalized Outputs:

Sample	$\hat{z}_1$	$\hat{z}_2$	$\hat{z}_3$
1	1.18	1.37	-0.20
2	-0.79	-0.59	-0.20
3	-1.18	-1.57	-1.22
4	1.18	1.37	1.83
5	-0.39	-0.59	-0.20

Table 11: Normalized pre-activations after Batch Normalization

17.1.5 Real-World Analogy

Imagine each neuron as a student taking several tests (mini-batches). Each test has varying difficulty—some are easy, some hard.

Batch Normalization acts like a teacher who:

- Subtracts the mean score (centers around zero difficulty)
- Divides by the standard deviation (equalizes the spread)
- Adds a learnable rescale (γ) and shift (β)—allowing regrading if needed

This ensures every neuron learns steadily, without one batch overpowering another.

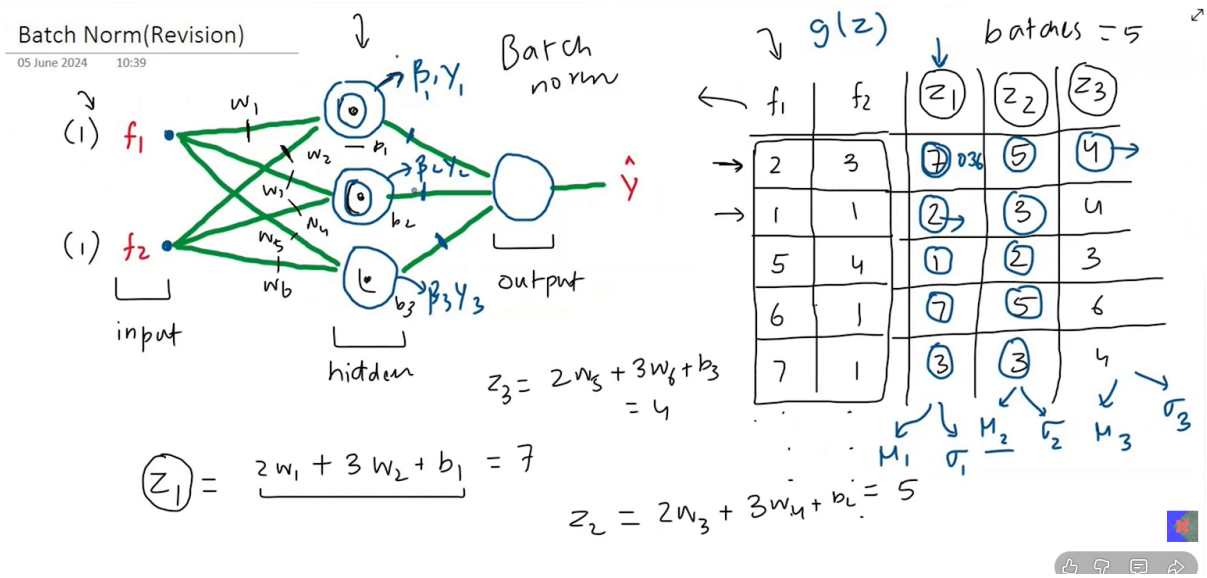

Figure 10: This is an example image

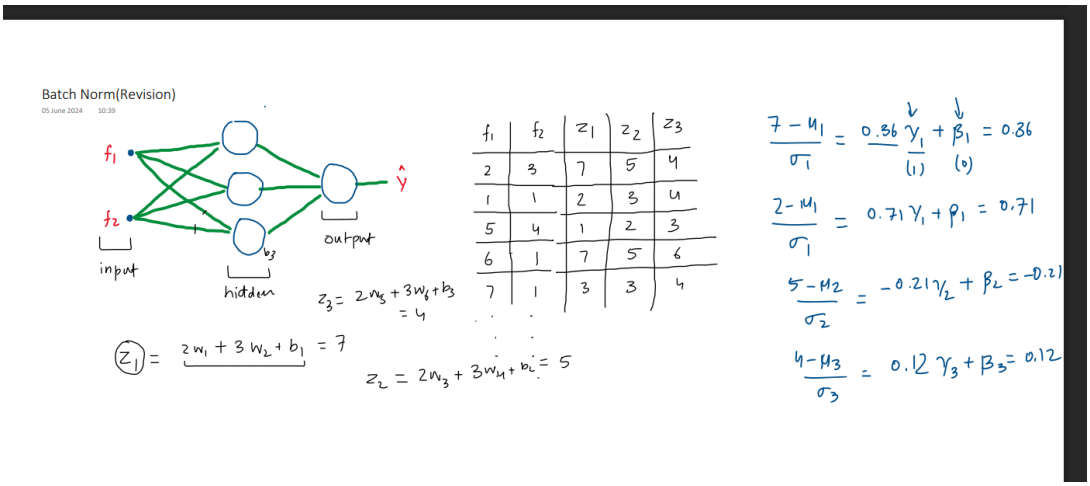

Figure 11: This is an example image

17.2 Layer Normalization in Transformers

17.2.1 Why Batch Normalization Fails in Transformers

While Batch Normalization works well for CNNs, it fails in Transformers and NLP tasks for three critical reasons:

1. Variable Sequence Lengths and Padding

When processing sentences:

- "Hi Deepak" → 2 tokens
- "How can I help you today" → 6 tokens

Transformers pad sequences to a fixed length (e.g., 10). Some positions contain real tokens, others are `<PAD>` tokens (typically zeros).

If we apply Batch Normalization:

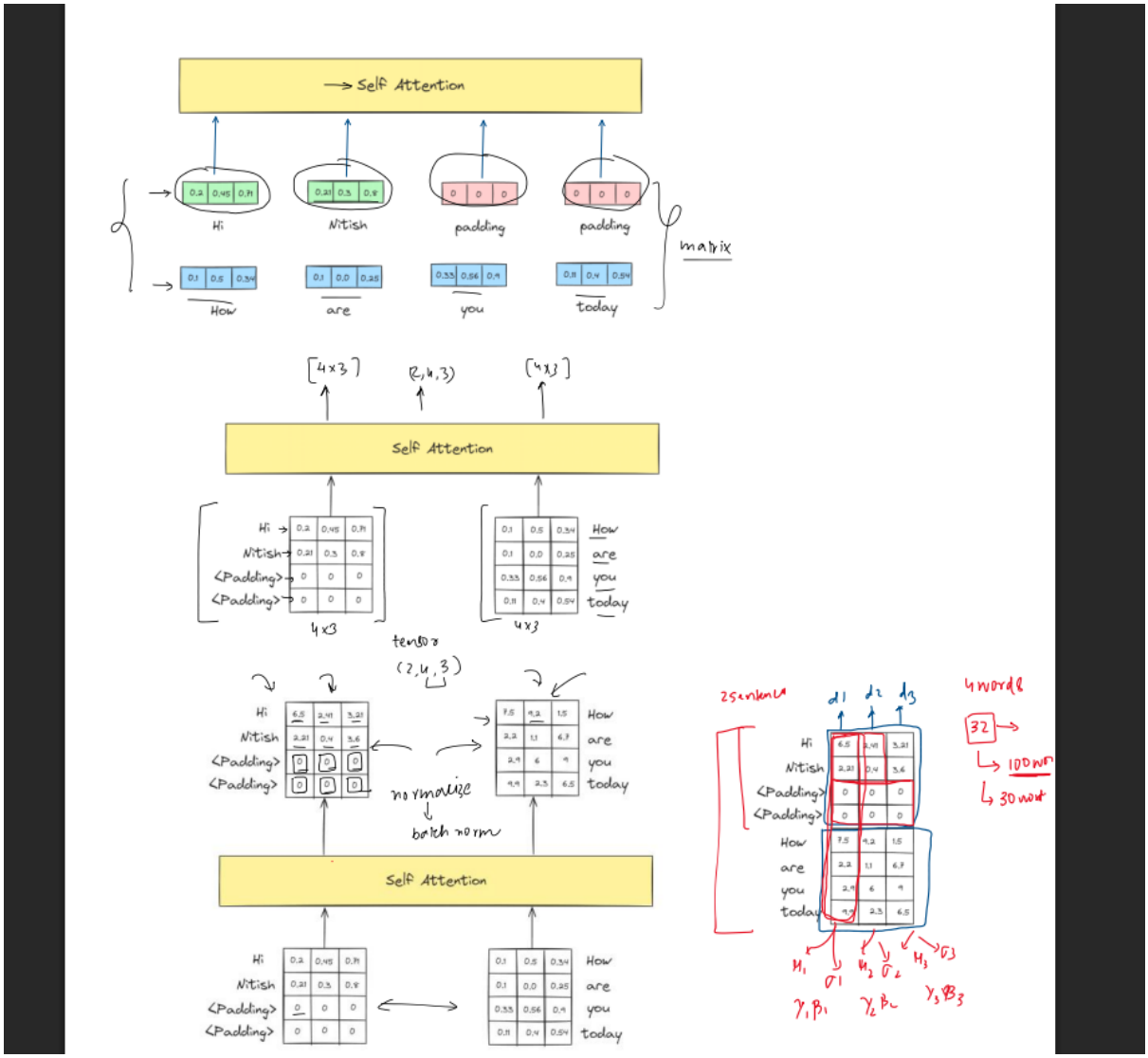

Figure 12: This is an example image

- It computes mean/variance *across the batch* (vertically across different sentences)
- Padding zeros dominate the statistics → mean shifts toward 0
- Model receives incorrect normalization statistics

Result: BN fails because statistics become meaningless with variable-length padded sequences.

2. Sequential and Autoregressive Processing

In RNNs and Transformers:

- Each token is processed independently
- Batch statistics vary across time steps

During inference, batch size might be 1 (single sentence). BN fails because it depends on a batch to estimate mean/variance.

3. Small Batch Sizes in NLP

Transformers often train with batch sizes of 4–8 (due to large memory requirements). BatchNorm's estimation of mean/variance becomes unstable for such small batches.

Conclusion: BatchNorm depends on batch-wide statistics, which breaks when there is padding, variable lengths, small batch sizes, or sequential dependencies.

17.2.2 Layer Normalization: The Solution

Instead of computing statistics *across the batch*, Layer Normalization computes **within each sample (token vector)**—across its feature dimension.

Mathematical Definition

For each sample i, normalize across its feature dimensions:

$$\hat{x}_i = \frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}}$$

where μ_i, σ_i^2 are computed **across features**, not across the batch.

Then apply learnable parameters:

$$y_i = \gamma \hat{x}_i + \beta$$

Complete Layer Normalization Formula:

$$\boxed{y_i = \gamma \left(\frac{x_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \right) + \beta}$$

17.2.3 Concrete Example

Consider two token embeddings (dimension $d = 4$):

Token	Embedding ($d = 4$)
Hi	[0.2, 0.4, 0.6, 0.8]
Deepak	[0.1, 0.3, 0.5, 0.7]

For token "Hi":

$$\mu = \frac{0.2 + 0.4 + 0.6 + 0.8}{4} = 0.5$$

$$\sigma^2 = \frac{(0.2 - 0.5)^2 + (0.4 - 0.5)^2 + (0.6 - 0.5)^2 + (0.8 - 0.5)^2}{4} = 0.05$$

Normalize:

$$\hat{x} = \frac{x - 0.5}{\sqrt{0.05 + \epsilon}} = [-1.34, -0.45, 0.45, 1.34]$$

Apply learnable parameters (assume $\gamma = 1, \beta = 0$):

$$y = \gamma \hat{x} + \beta = [-1.34, -0.45, 0.45, 1.34]$$

Key Advantage: No dependency on batch size, no problem with padding zeros, and stable across all tokens.

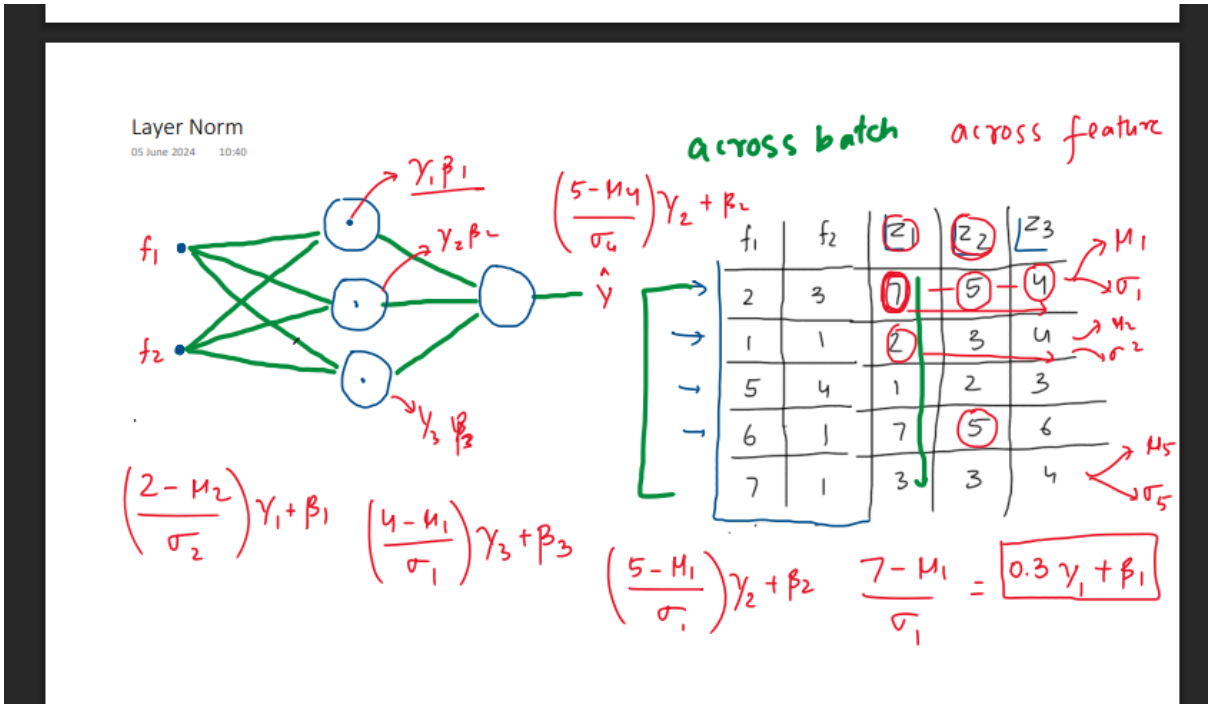

Figure 13: This is an example image

17.2.4 Layer Normalization in Transformer Architecture

1. Placement in Transformers

LayerNorm is used:

- Before and/or after Self-Attention and Feed-Forward blocks
- In Pre-LN and Post-LN architectures

Post-LN (original Transformer, Vaswani et al. 2017):

$$x = x + \text{Sublayer}(\text{LayerNorm}(x))$$

Problem: gradient instability for deep networks.

Pre-LN (used in GPT, BERT, etc.):

$$x = x + \text{Sublayer}(\text{LayerNorm}(x))$$

More stable training and better convergence.

2. LayerNorm in Multi-Head Self-Attention

When self-attention outputs a matrix of shape $[\text{batch}, \text{seq\_len}, \text{hidden\_dim}]$, LayerNorm is applied **independently to each token vector** of size hidden_dim.

For each word (e.g., "Hi", "Deepak", "today"), LayerNorm normalizes its embedding features.

3. Learnable Parameters

LayerNorm has learnable γ (scale) and β (shift), like BatchNorm. They have the same shape as the feature dimension:

$$\gamma, \beta \in \mathbb{R}^{d_{\text{model}}}$$

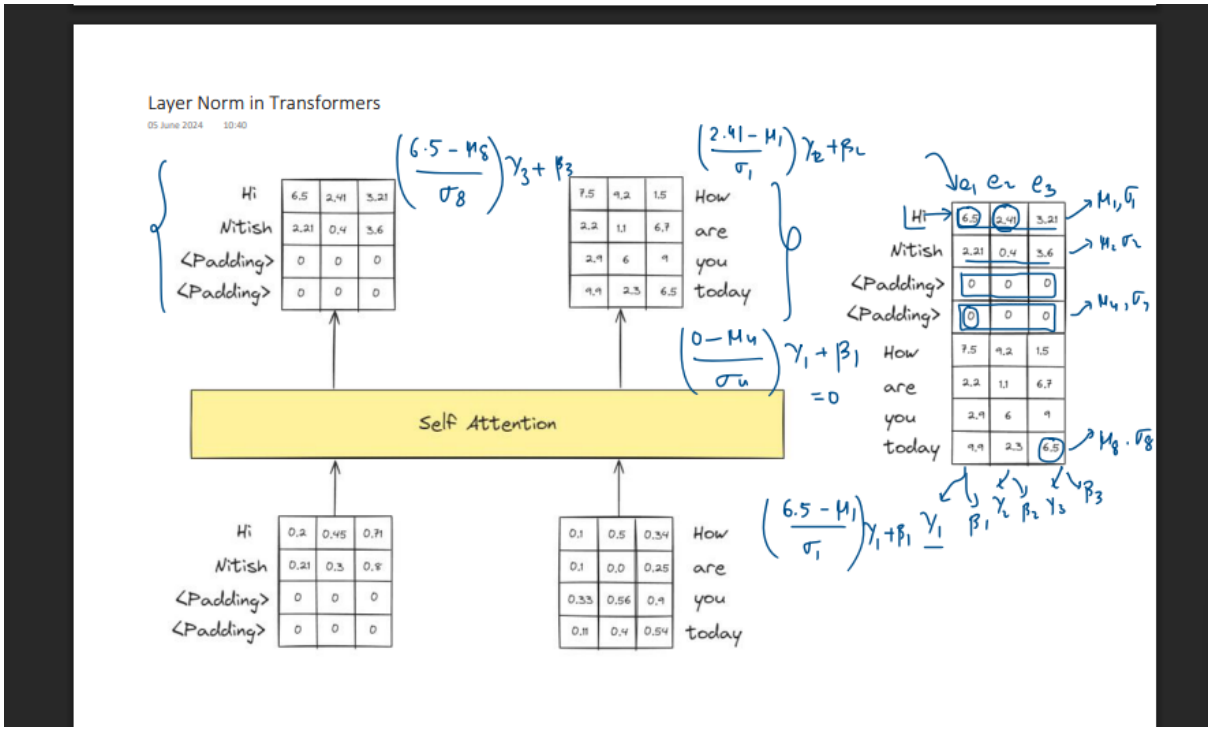

Figure 14: This is an example image

so every feature dimension learns its own rescaling.

4. Training vs Inference

Unlike BatchNorm, LayerNorm uses **the same mean and variance** at training and inference time (since they are computed per sample). This removes the need for running averages—another reason it's preferred in Transformers.

17.2.5 Comparison: Batch Normalization vs Layer Normalization

Feature	**Batch Normalization**	**Layer Normalization**
Normalizes	Across batch dimension	Across feature dimension
Formula	(μ, σ) per feature across batch	(μ, σ) per sample across features
Batch size dependency	High	None
Padding effect	Strongly affected	Unaffected
Used in	CNNs, MLPs	Transformers, RNNs, NLP
Learnable params	γ, β per feature	γ, β per feature dimension
Train/Test difference	Yes (uses running averages)	No (same statistics)
Typical shape	[Batch, Channels, H, W]	[Batch, Seq, Hidden]

Table 12: Comparison of Batch Normalization and Layer Normalization

17.2.6 Intuitive Analogy

Batch Normalization normalizes "across students in a class"—it compares how everyone performed relative to each other.

Layer Normalization normalizes "within a student's own subjects"—making sure their performance is internally balanced, regardless of other students.

17.2.7 Summary

Batch Normalization:

- Normalizes across batch for each feature
- Works well for CNNs with large, fixed-size batches
- Fails in NLP due to variable lengths, padding, and small batches

Layer Normalization:

- Normalizes across features for each sample
- Stable for any sequence length and batch size
- Consistent between training and inference
- Standard in Transformers (GPT, BERT, LLaMA)

18 Comparison Tables

18.1 RNN vs Transformer

Property	**RNN/LSTM**	**Transformer**
Processing	Sequential (word-by-word)	Parallel (all words at once)
Long-range dependencies	Struggles (vanishing gradients)	Excellent (direct connections)
Training speed	Slow (cannot parallelize)	Fast (full GPU utilization)
Memory bottleneck	Hidden state compresses everything	Attention to all tokens
Gradient flow	Degrades over long sequences	Direct paths via attention
Context window	Limited by hidden state size	Limited by quadratic complexity

Table 13: RNN vs Transformer Architecture Comparison

18.2 Final Summary: Static vs Dynamic

Aspect	**Static Embedding**	**Dynamic (Self-Attention)**
Representation	Fixed vector	Context-dependent vector
"bank" in "money bank"	Same as always	Financial institution
"bank" in "river bank"	Same as always	Geographical location
Training	Pre-trained separately	End-to-end with model
Quality	Lower for ambiguous words	Higher, disambiguation
Parameters	Fewer (vocabulary only)	More (vocab + W_Q/K/V)
Computation	Fast lookup only	Requires attention computation

Table 14: Final Comparison: Static vs Dynamic Embeddings

19 Transformer Encoder Architecture

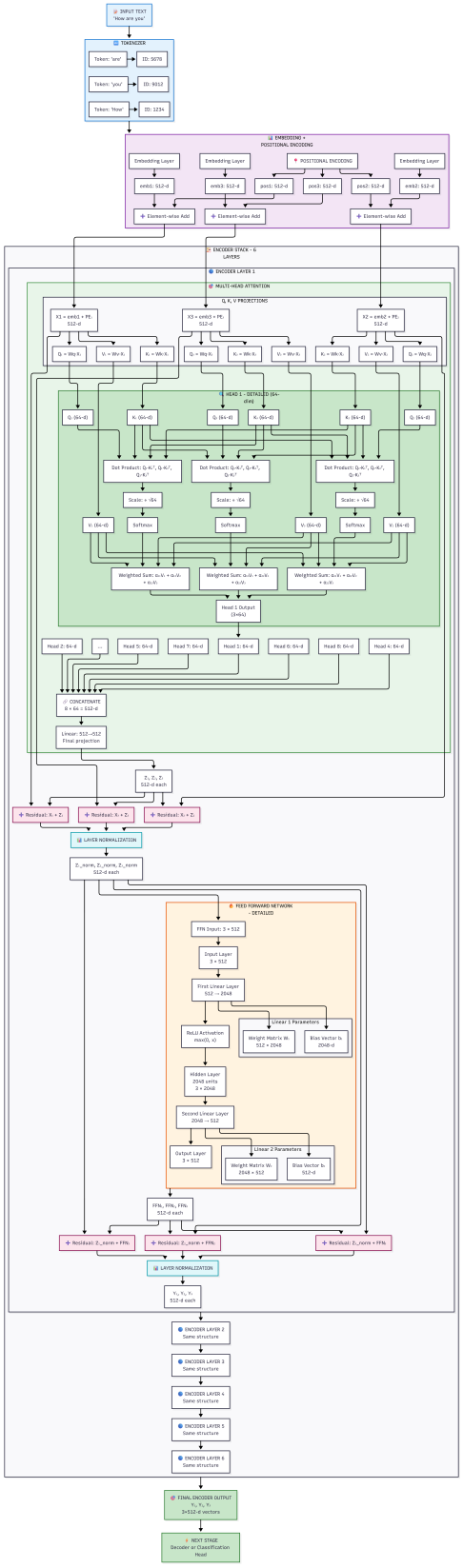

Figure 15: Complete Transformer Encoder Architecture

19.1 Why These Components?

19.1.1 1. Why Residual Connections?

Purpose: Enable training of deep networks by preserving gradient flow.

Key Points:

- **Gradient highway**: The +1 term in

$$\frac{\partial(x + f(x))}{\partial x} = 1 + \frac{\partial f}{\partial x}$$

  creates a direct path for gradients to flow backward through all layers, preventing vanishing gradients.

- **Information preservation**: The original input x is added to the transformation $f(x)$, ensuring important information is not lost even if a layer learns poorly.

- **Easy optimization**: The network can learn $f(x) = 0$ (identity mapping) when no transformation is needed, making training easier and more stable.

- **Deep networks possible**: Residuals enable training of deep stacks (6+ layers and far beyond) without vanishing gradients.

19.1.2 2. Why Feed-Forward Network (FFN)?

Purpose: Introduce non-linearity and increase per-token capacity for complex transformations.

Key Points:

- **Non-linearity via ReLU**: Self-attention is mostly linear; without non-linearity repeated linear layers collapse into one linear transform.

- **Dimensional expansion**: A $512 \to 2048 \to 512$ bottleneck allows the network to learn richer intermediate representations.

- **Position-wise independence**: Each token is processed independently through the same FFN, enabling full parallelization on GPUs.

- **Parameter share**: FFN typically contains the majority of layer parameters (e.g. $\sim$66% in the 512/2048 design).

19.1.3 3. Why 6 Encoder Layers?

Purpose: Build hierarchical representations from low-level patterns to high-level abstractions.

Key Points:

- **Hierarchical learning**: Lower layers capture local/syntactic patterns; middle layers capture semantics; higher layers represent abstract/task-relevant features.

- **Empirical sweet spot**: The original Transformer used 6 layers for the translation tasks; it balances performance and compute for base models.

- **Residual-enabled training**: Residuals (and normalization) make 6+ layers trainable and stable.
- **Scalability**: Larger models often use many more layers (e.g., BERT-base:12, GPT-3:96) depending on data and compute.

19.1.4 4. Why Multi-Head Self-Attention?

Purpose: Allow each token to gather context from all other tokens while learning multiple types of relations simultaneously.

Key Points:

- **Context awareness**: Each token attends to every other token, so contextual meaning (e.g., "bank" as financial vs river) is dynamically resolved.
- **Multiple perspectives (8 heads)**: Different heads can learn syntax, short-range dependencies, long-range relations, and semantic affinities.
- **Parallel processing**: Attention operates via matrix ops, allowing efficient GPU utilization compared to sequential RNNs.
- **Global information flow**: Direct paths exist between any two tokens (constant path length), improving long-range dependency modeling.

Summary: Multi-Head Attention captures context; FFN adds non-linear expressiveness; residuals enable deep stacking; and multiple layers build hierarchy. Together they form the Encoder.

19.2 Detailed Component Breakdown

19.2.1 Position-wise Feed-Forward Network (FFN)

Architecture:

- Layer 1: $\mathbb{R}^{512} \to \mathbb{R}^{2048}$ (expansion)
- ReLU activation
- Layer 2: $\mathbb{R}^{2048} \to \mathbb{R}^{512}$ (compression)

Mathematical definition:

$$\mathrm{FFN}(x) = \mathrm{ReLU}(xW_1 + b_1)W_2 + b_2,$$

where $W_1 \in \mathbb{R}^{512\times2048}, W_2 \in \mathbb{R}^{2048\times512}$ and biases $b_1 \in \mathbb{R}^{2048}, b_2 \in \mathbb{R}^{512}$.

Dimensions:

- Input: 512
- Hidden: 2048 (4×)
- Output: 512

Why this design?

- ReLU injects non-linearity (without it, stacked linear transforms collapse).
- Expansion provides an expressive intermediate space.
- Position-wise processing is fully parallelizable and keeps sequence length constant.

Parameter counts (per layer approx.)

$$\text{FFN params} \approx 512 \times 2048 + 2048 \times 512 = 2{,}097{,}152$$

$$\text{Attention params (rough)} \approx 1{,}048{,}576$$

FFN typically dominates the parameter budget.

19.2.2 Why 6 Encoder Layers? — The Depth Magic

Progressive refinement:

Layer	What It Learns	Example ("How are you")
Layer 1	Local dependencies	"How" ↔ "are"
Layer 2	Syntax	subject–verb links
Layer 3	Semantics	question pattern recognition
Layer 4	Pragmatics	expected replies / social context
Layer 5	Domain patterns	conversational styles
Layer 6	Abstractions	final contextual embedding

Table 15: Progressive refinement across 6 encoder layers.

Empirical findings:

- Too few layers (1–2): underfit complex language patterns.
- Too many layers (12+ for small models): diminishing returns and higher compute / overfit risk.
- Six layers: practical trade-off in original Transformer for translation tasks.

19.3 Complete Mathematical Formulation

For encoder layer ℓ:

Step 1: Multi-Head Attention + Residual + LayerNorm

$$Z_\ell = \text{LayerNorm}\big(X_\ell + \text{MultiHeadAttention}(X_\ell)\big)$$

Step 2: FFN + Residual + LayerNorm

$$X_{\ell+1} = \text{LayerNorm}\big(Z_\ell + \text{FFN}(Z_\ell)\big)$$

where X_ℓ is the input to layer ℓ and $X_{\ell+1}$ is the output (input to layer $\ell + 1$). Stack for $\ell = 1 \ldots 6$ to get final encoder outputs.

19.4 Complete PyTorch Implementation (clean)

Listing 1: Compact Transformer Encoder (PyTorch)

```
import torch
import torch.nn as nn

class EncoderLayer(nn.Module):
    def __init__(self, d_model=512, n_heads=8, d_ff=2048, dropout=0.1):
        super().__init__()
        self.self_attn = nn.MultiheadAttention(embed_dim=d_model, num_heads=
    n_heads, dropout=dropout)
        self.ffn = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(d_ff, d_model)
        )
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.drop = nn.Dropout(dropout)

    def forward(self, x):
        # x: (seq_len, batch, d_model) for nn.MultiheadAttention
        attn_out, attn_weights = self.self_attn(x, x, x)
        x = self.norm1(x + self.drop(attn_out))
        ff = self.ffn(x)
        x = self.norm2(x + self.drop(ff))
        return x, attn_weights

class TransformerEncoder(nn.Module):
    def __init__(self, num_layers=6, d_model=512, n_heads=8, d_ff=2048, dropout
    =0.1):
        super().__init__()
        self.layers = nn.ModuleList([EncoderLayer(d_model, n_heads, d_ff,
    dropout) for _ in range(num_layers)])

    def forward(self, x):
        # x: (seq_len, batch, d_model)
        attn_maps = []
        for layer in self.layers:
            x, attn = layer(x)
            attn_maps.append(attn)
        return x, attn_maps

# Usage example (seq_len=3, batch=1, d_model=512)
if __name__ == "__main__":
    seq_len, batch, d_model = 3, 1, 512
    x = torch.randn(seq_len, batch, d_model)
    encoder = TransformerEncoder()
    out, attn_maps = encoder(x)
    print("Output␣shape:", out.shape)
```

19.5 Key Insights Summary

1. **Multi-Head Attention**: Enables each token to attend to all others, yielding context-aware embeddings.
2. **Residual Connections**: Preserve gradients and let layers learn corrections rather than full transformations.
3. **Layer Normalization**: Stabilizes activations independent of batch size.
4. **Feed-Forward**: Injects non-linearity and increases per-token capacity.
5. **Six Layers**: Provides hierarchical feature extraction while remaining practical in compute.
6. **Position-wise Processing**: Preserves sequence length and allows efficient parallelism.

20 Conclusion

Self-attention is the revolutionary mechanism enabling transformers to:

- **Create context-aware representations**: Solving the "bank" ambiguity problem
- **Process sequences in parallel**: Dramatically faster than RNNs
- **Capture long-range dependencies**: Direct connections between any tokens
- **Scale to billions of parameters**: GPT-3 (175B), GPT-4 (rumored 1.8T)

The Mathematical Essence:

$$\boxed{y_{\text{word}} = \sum_{i=1}^{n} \underbrace{\text{softmax}(q_{\text{word}} \cdot k_i)}_{\text{attention weight}} \cdot v_i} \tag{118}$$

Each word becomes a **learned weighted combination** of all words in context. This simple yet powerful idea revolutionized artificial intelligence and enabled:

- ChatGPT and large language models
- Modern machine translation systems
- Code generation tools
- Protein structure prediction (AlphaFold)
- And countless other applications

All parameters (embeddings, W^Q, W^K, W^V) are learned via backpropagation during training on massive datasets, enabling the model to discover semantic relationships and contextual patterns automatically!