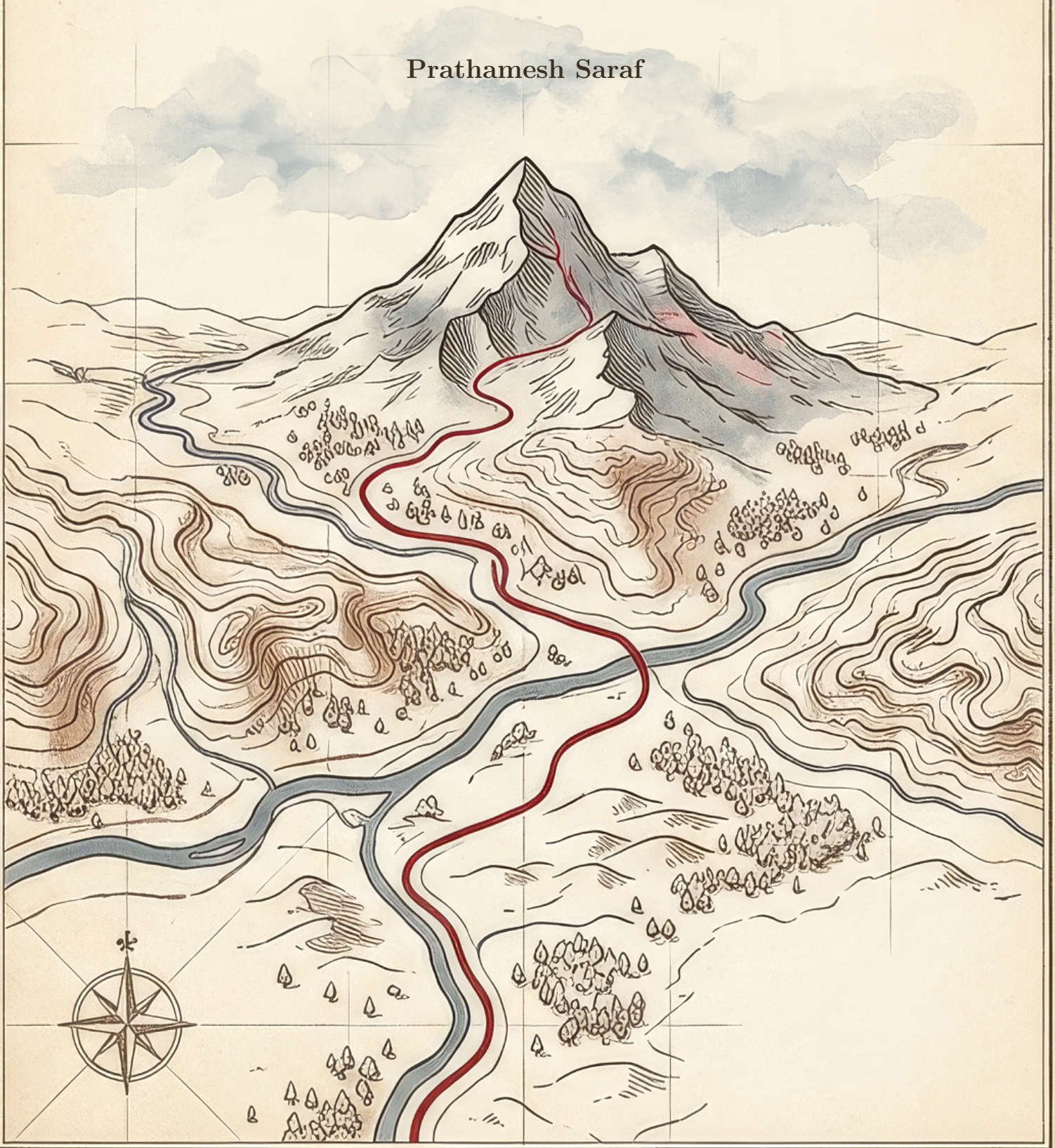
My Adventures with Large Language Models

Build foundational LLMs from Transformers to DeepSeek from scratch, in PyTorch

Prathamesh Saraf



Preface

Welcome to *My Adventures with Large Language Models.*

If you have read introductions to building LLMs before, you have probably seen the same destination over and over: a working GPT-2, trained on a small corpus, with a few hundred lines of PyTorch and a satisfying loss curve. That is a wonderful place to arrive. It is also, in 2026, only the beginning of the story.

The architectures actually powering frontier models today (Llama 3, DeepSeek V3, and the systems that quietly run inside ChatGPT and Claude) differ from GPT-2 in ways that matter: rotary positional embeddings instead of sinusoidal, RMSNorm instead of LayerNorm, SwiGLU instead of GELU, grouped-query attention instead of multi-head, and, most recently, multi-head latent attention with decoupled RoPE and a mixture-of-experts feed-forward block. None of these are exotic. All of them can be implemented from scratch on a laptop. Almost none of the available "build an LLM" resources cover them.

This book does. We will build a vanilla transformer to ground our intuitions, walk through GPT-2 to consolidate them, and then spend the rest of the book on the architectures that are actually shipping: Llama 3 in Chapter 3, the KV-cache optimisations that make modern inference feasible in Chapter 4, and DeepSeek's MLA and DeepSeekMoE in Chapter 5. By the end, you will not just know how an LLM works: you will have built one that looks recognisably like the systems running in production today.

What You'll Learn

The book is structured as a progressive build, with each chapter introducing the next generation of ideas:

- **Chapter 1: A Journey into the Heart of the Transformer.** We implement the original "Attention is All You Need" paper end-to-end, building a complete encoder–decoder transformer that translates English to Hindi. You will understand embeddings, sinusoidal positional encoding, scaled dot-product and multi-head attention, layer normalisation, residual connections, and the full training loop.
- **Chapter 2: LLMs are Unsupervised Multitask Learners.** We strip the transformer down to its decoder-only form and rebuild GPT-2 from scratch, including pretraining on a real corpus, tokenisation, and inference with greedy and sampling-based decoding.
- **Chapter 3: Llama 3: Herd of Models.** We implement Meta's Llama 3.2-3B, replacing each component of GPT-2 with its modern counterpart: RMSNorm, RoPE, SwiGLU, and Grouped-Query Attention. By the end of this chapter, you will have a working Llama 3 implementation that loads pretrained weights.
- **Chapter 4: Speeding Up Inference: KV Cache.** We turn from architecture to systems. Why is autoregressive inference so slow, what does the KV cache actually cache, and how do MQA and GQA trade memory for expressiveness? This chapter is the bridge between "it works" and "it works at scale."

- **Chapter 5: DeepSeek: The Final Adventure.** We close with the architecture that, as of this writing, sets the bar for open-weight efficiency: DeepSeek's Multi-Head Latent Attention (MLA), Decoupled RoPE, DeepSeekMoE with shared and fine-grained experts, auxiliary-loss-free load balancing, and Multi-Token Prediction.

Who This Book Is For

This book is written for:

- ML practitioners and engineers who want to understand modern LLMs at the level of code, not slides;
- Students and researchers in NLP who want a working implementation alongside the papers;
- Software engineers who learn best by building things from scratch;
- Anyone who has used ChatGPT or Claude and wondered, in detail, how the thing actually works.

You will need a working knowledge of Python and a passing familiarity with PyTorch and neural networks. You do not need a research background (every concept is built up from first principles, with analogies and worked examples), but you should be comfortable reading code.

What This Book Is *Not*

To save you time, here is what this book deliberately does *not* cover:

- It is not a survey of NLP history or a literature review.
- It is not a guide to using the OpenAI, Anthropic, or any other LLM API.
- It is not a fine-tuning, RLHF, or DPO cookbook (though the foundations here will make those much easier to learn).
- It is not a retrieval-augmented generation (RAG) or LLM-application engineering book.
- It is not a deployment, serving, or MLOps guide.

If those are what you are looking for, you will likely be happier with a different book. If you want to know what is happening inside the model when you call `model.generate()`, you are in the right place.

Hardware Requirements

Most of the code in this book will run on a modern laptop with 16 GB of RAM. Specifically:

- **Chapters 1–2**: A laptop CPU is sufficient. Training the small transformer in Chapter 1 takes minutes; pretraining the Chapter 2 GPT-2 to a meaningful loss benefits from a single GPU but is not strictly required.
- **Chapter 3**: Loading and running Llama 3.2-3B with pretrained weights needs roughly 8 GB of VRAM in half precision. A single consumer GPU (RTX 3090, 4090, or equivalent) is comfortable.
- **Chapters 4–5**: The KV-cache and MLA discussions are largely architectural; the accompanying code is small and runs on any modern machine. Running full DeepSeek-scale experiments is out of scope; we focus on understanding and on small reproductions.

If you do not have a GPU, the code is structured so that every chapter has a small-scale variant you can run on CPU and a full-scale variant that requires accelerated hardware.

The Code Repository

All code in the book is available at:

`https://github.com/S1LV3RJ1NX/mal-code`

The repository is organised by chapter (`ch01/`, `ch02/`, …, `ch05/`), with each chapter containing a runnable end-to-end script, a notebook walkthrough, and a `README` mapping book sections to files. Errata, updates, and reader-contributed extensions are tracked in the repository's `ERRATA.md` and `CHANGELOG.md`.

How to Read This Book

- **Read with the editor open.** Type the code as you go. The single biggest determinant of whether you finish this book is whether you actually run the code.
- **Break things.** Change the embedding dimension, swap RMSNorm back for LayerNorm, replace SwiGLU with GELU, and see what happens to the loss curves. Most of the insight in modern LLM design comes from negative experiments.
- **Read the papers alongside.** Each chapter cites the papers it builds on. The book is intended to be a working companion to those papers, not a replacement for them.
- **Be patient with the maths.** A handful of derivations (RoPE, MLA) repay slow reading. Skim them on the first pass if you must, but come back.

A Note on the Journey

Writing this book has been an education. The hardest parts were not the architectures themselves, but holding the chain of "why" all the way down: why RoPE, why GQA, why MLA, why now. Every time I thought I understood a technique, writing it up showed me I did not. I have tried to leave that texture in: the dead ends, the trade-offs, the reasons something elegant on paper turns out to be the wrong default in practice. The field will keep moving; this snapshot is what I wish I had had a year ago.

Let's begin our adventure.

Prathamesh Saraf
First Edition, 2026

Acknowledgements

This book stands on the work of many people.

I am indebted, technically and pedagogically, to **Andrej Karpathy**[1] for the Neural Networks: Zero to Hero series; to **Sebastian Raschka**[2] for *Build a Large Language Model (From Scratch)*, which set the standard this book has tried to extend; to **Umar Jamil**[3] for his transformer videos, which clarified many details I had glossed over; and to **Raj Dandekar**[4] for the DeepSeek lectures that informed Chapter 5.

Truefoundry[5] provided the compute that made the experiments in this book possible, and several engineers there reviewed drafts and caught errors I would have shipped.

To my mother, **Supriya**, and my uncle, **Sagar**: thank you, for everything, always.

To **Hima**, who lived through every chapter of this book and somehow still encouraged the next one: this would not exist without you.

To **Kunal**, who first suggested I write this and then refused to let me forget I had said yes: thank you for the push.

And to the researchers, engineers, and open-source contributors whose work this book describes: thank you for doing it in public.

[1] `https://x.com/karpathy`
[2] `https://x.com/rasbt`
[3] `https://x.com/hkproj`
[4] `https://x.com/raj_dandekar`
[5] `https://truefoundry.com/`

Contents

Chapter 1

A Journey into the Heart of the Transformer

Picture yourself trying to read a book in a dimly lit room. You squint, focusing on each word in turn, much like how early sequence models had to crank through language one token at a time. The trick that changed things was learning to shine a spotlight (a different one for each word being generated) on the parts of the input that mattered most. That trick is called *attention*, and this chapter is about building it from scratch.

Our goal: Build a complete transformer, end-to-end, in PyTorch. We'll take the architecture apart into its components, understand each one in isolation, and then put them back together.

Figure 1.1: Not so real architecture of a typical transformer

By the end of this chapter, you will have built a translation system, based on the transformer architecture, that translates English to Hindi (Fig 1.1). To get there, we will implement the paper ***Attention Is All You Need*** [1] in full. I will assume working familiarity with Python and PyTorch; everything else is built up as we go. The code for this chapter lives at `https://github.com/S1LV3RJ1NX/mal-code/tree/main/ch1`.

Architecture. The translation architecture, in full, looks like this:

We shall examine each component shown in Fig 1.2. Our path through the chapter is:

1. Embeddings: representing words as vectors a model can work with.
2. Positional Encoding: giving the model a sense of order.
3. Attention Mechanism: the heart of the transformer.
4. Layer Normalization: keeping activations well-behaved.
5. Residual Connections: preserving signal across depth.

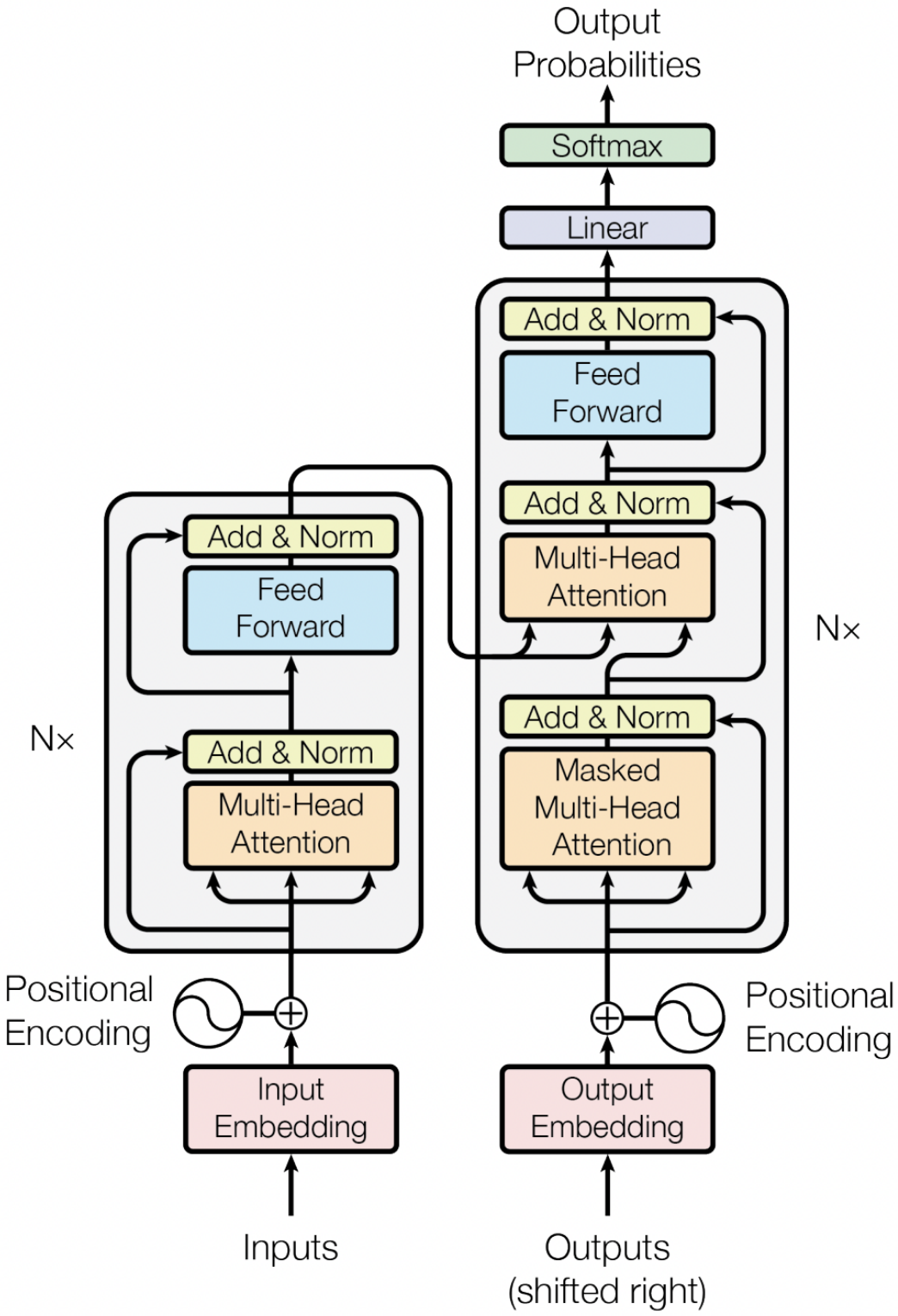


Figure 1.2: Actual architecture of the transformer [1]

6. Position-wise Feed-Forward Networks: per-token processing.
7. Building an Encoder: the comprehension half.
8. Building a Decoder: the generation half.
9. Projection Layer: mapping hidden states back to vocabulary logits.
10. Complete Transformer Assembly: wiring everything together.
11. Dataset and Tokenizer: preparing the bilingual corpus.
12. Training and Validation: fitting the model to the task.
13. Conclusion.

1.1 Embeddings

Imagine you're at a family gathering where your grandmother speaks only Hindi and your cousin speaks only English. When your grandmother says '**पानी**' *(paani)*, someone brings water, and when she says '**खाना**' *(khaana)*, food appears. Your cousin starts noticing these patterns - even without knowing Hindi, they're beginning to associate words across languages through context. This is exactly what we're teaching our machines to do!

The Challenge: Making Words Machine-Friendly Let's play a translation game. How would you explain to a computer that `water` and **पानी** *(paani)* mean the same thing? Machines only understand numbers, so we need a clever way to convert both English and Hindi words into numerical patterns that capture their meaning and relationships across languages. This is where embeddings come in. Think of embeddings as creating a universal language map. When you say `water` in English, it should be placed close to **पानी** *(paani)* in this map because they mean the same thing. Similarly, `cold water` with **ठंडा पानी** *(thanda paani)* and `hot water` with **गरम पानी** *(garam paani)* should be nearby because they're related concepts. This is how our transformer will learn to navigate between languages! But how do we create this numerical map? This is where Word2Vec [2] comes in: one of the earliest breakthroughs in word embeddings, introduced by Tomas Mikolov and his team at Google.

To see how it works, let's imagine a game. Picture a sentence with a missing word: "The _____ is red and juicy." Your mind might instantly supply "apple" or "cherry" because these words make sense in the context. Word2Vec works similarly. It's trained to predict a word based on its surrounding words (continuous bag of words model) or to predict the surrounding words given a specific word (this is called the skip-gram model).

To understand intuitively,

- Continuous Bag of Words (CBOW): In CBOW, the model predicts a target word based on its context (the surrounding words). For example, given the sentence "The cat sat on the mat", the model would try to predict 'cat' from 'the', 'sat', 'on', and 'mat.'
- Skip-gram: In Skip-gram, the model does the opposite: it predicts the context words given a target word. Using the same sentence, it would try to predict 'the', 'sat', 'on', and 'mat' given 'cat'.

Through this process, Word2Vec learns not just the meaning of individual words but also the relationships between them. Words that appear in similar contexts, like "king" and "queen" or "doctor" and "nurse", end up being mapped close together in the embedding space. Let's look at an example to better understand embeddings. Don't worry about understanding all the code: it's mainly for building intuition.

1.1.1 Example with Word2Vec

Code:

```
# Imports
import gensim.downloader as api
from gensim.models import Word2Vec

model = api.load("word2vec-google-news-300")
# Total size of the vocabulary
print(len(model))
```

Output:

```
3000000
```

Now, let's see how we can manipulate word vectors. We can perform vector arithmetic to find relationships between words. For example, let's see if `king - man + woman` is approximately equal to `queen`.

Code:

```
# Get word vectors
king_vector = model['king']
man_vector = model['man']
woman_vector = model['woman']

# Vector arithmetic
result_vector = king_vector - man_vector + woman_vector

# Find the most similar word to the result vector
most_similar_word = model.most_similar([result_vector], topn=2)
print(f"The most similar word to 'king - man + woman' is: {most_similar_word}")
```

Output:

```
The most similar word to 'king - man + woman' is: [('king', 0.8449392318725586),
↪  ('queen', 0.7300517559051514)
```

It turns out that for this example to work properly, there's a subtle detail we should note. The actual result of `King - Man + Woman` would be closest to `King`. The widely known example works because the implementation excludes the original input words from the possible results! This means while the resulting vector is actually closest to `King`, the algorithm returns `Queen` as it's the next closest match.

Code:

```
# Check most similar words to man
model.most_similar('man')
```

Output:

```
[('woman', 0.7664012908935547),
 ('boy', 0.6824871301651001),
 ('teenager', 0.6586930155754089),
 ('teenage_girl', 0.6147903203964233),
 ('girl', 0.5921714305877686),
 ('suspected_purse_snatcher', 0.571636438369751),
 ('robber', 0.5585119128227234),
 ('Robbery_suspect', 0.5584409832954407),
 ('teen_ager', 0.5549196600914001),
 ('men', 0.5489763021469116)]
```

Thus, we see that,

- Sentences are converted to words, that form your vocabulary
- Each vocabulary has an associated embedding with it.
- We have around 3000K words vectors with dimension of each vector being 300.

With that background, let's write an embedding class for our transformer model, replicating the details in the paper [1].

1.1.2 The Embedding Class

Imagine you're teaching a child to read. You start with a picture book where each word has an associated image. We're doing the same thing for our model, but instead of pictures, each word is associated with a list of numbers (a `vector`). Let's build that lookup together.
Code:

```python
# Imports
import torch
import torch.nn as nn
import numpy as np

class InputEmbeddings(nn.Module):
    """
    This class represents the Input Embeddings for our AI model, functioning as a
    ↪ personal dictionary:
    - Each word is assigned a unique code (embedding).
    - These codes are crafted to reflect relationships between words.
    - Words with similar meanings will have embeddings that are close to each other
    ↪ in the vector space.

    For example, in our minds, 'cat' and 'kitten' are closely related, just as
    ↪ their embeddings will be!
    """

    def __init__(self, vocab_size: int, d_model: int):
        """
        Initializes the InputEmbeddings class.

        Parameters:
        vocab_size (int): The total number of unique words the model can understand
        ↪ (akin to the number of words in a dictionary).
        d_model (int): The dimensionality of the word embeddings, which determines
        ↪ how detailed the word representations are (commonly set to 512).
```

```
        """
        super().__init__()
        self.d_model = d_model
        self.vocab_size = vocab_size
        # Each word gets assigned its own vector representation!
        self.embedding = nn.Embedding(vocab_size, d_model)

    def forward(self, x: torch.Tensor):
        """
        The forward method converts input word indices into their corresponding
        ↪  embeddings.

        Parameters:
        x (torch.Tensor): A tensor containing the indices of the words to be
        ↪  converted into embeddings.

        Returns:
        torch.Tensor: The scaled embeddings for the input words.
        """
        return self.embedding(x) * np.sqrt(self.d_model)
```

The embedding process is similar to looking up words in our dictionary. The paper suggests scaling the embeddings by $\sqrt{d_{model}}$. This scaling is a normalization technique that ensures the embeddings have an appropriate scale relative to the model's dimensionality. This scaling helps maintain the stability of the training process and prevents the gradients from becoming too large, especially when dealing with high-dimensional embeddings.

1.1.3 Sidetrack: Understanding nn.Embedding

Imagine you're in a huge library with thousands of books. Instead of keeping the actual books at the front desk, the librarian has a special card catalog where:

- Each book has a unique number (like an ID)
- Each card contains detailed information about the book (genre, themes, topics, etc.)

This analogy perfectly captures how `nn.Embedding` works!
Code: Simple Example of `nn.Embedding`

```
# Create a small embedding layer
vocab_size = 5  # We only have 5 words
embedding_dim = 3  # Each word is represented in 3 dimensions
embedding = nn.Embedding(vocab_size, embedding_dim)

# Initally the embeddings are random
print(embedding.weight.data)
```

Output:

```
tensor([[-0.1115,  0.1204, -0.3696],
        [-0.2404, -1.1969,  0.2093],
        [-0.9724, -0.7550,  0.3239],
        [-0.1085,  0.2103, -0.3908],
        [ 0.2350,  0.6653,  0.3528]])
```

During training, these values get updated to capture meaning. Similar words end up with similar vectors: 'cat' and 'dog' might have closer vectors than 'cat' and 'fish'.

Code:

```
words = ["cat", "dog"]
# Assume in the vocabulary, cat is at index 0 and dog is at index 1
word_ids = torch.tensor([0, 1])

# Get embeddings
vectors = embedding(word_ids)
print(f"Shape of output: {vectors.shape}")  # Will be [2, 3]
# 2 because we input 2 words
# 3 because each word becomes a vector of 3 numbers
print(f"Embedding for cat: {vectors[0]}")
print(f"Embedding for dog: {vectors[1]}")
```

Output:

```
Shape of output: torch.Size([2, 3])
Embedding for cat: tensor([-0.1115,  0.1204, -0.3696])
Embedding for dog: tensor([-0.2404, -1.1969,  0.2093])
```

Thus, we see that the embedding for cat is the 0^{th} row in the embedding layer, while dog is the 1^{st}.

1.1.4 Visual Intuition

Let's visualize the current embedding class in terms of its input and output shape.

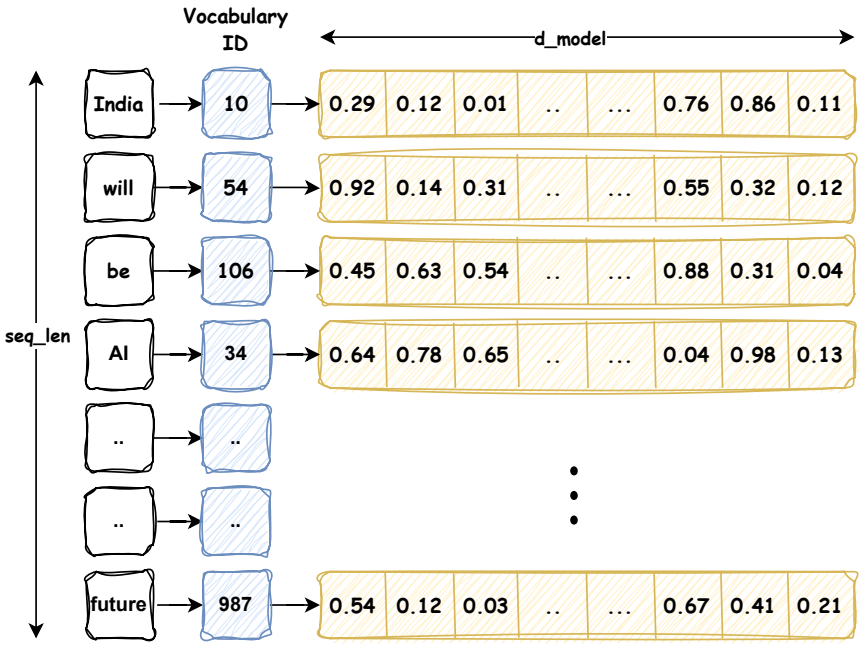


Figure 1.3: Embeddings Visualization

Fig 1.3 visually demonstrates how an input sequence of words is transformed into a sequence of dense vectors (embeddings) using an embedding layer. Each word is first converted to its corresponding ID from the existing vocabulary[1], and then the embedding layer maps these IDs to high-dimensional vectors that capture semantic relationships between words. The final output is a tensor of shape (seq_len, d_{model}) that can be further processed by downstream layers in the network.

In practice, often batches of data are sent to the network for processing. Hence, we add additional batch dimension, as shown in the Fig 1.4.

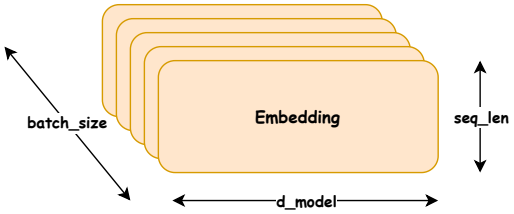


Figure 1.4: Embeddings with batches

1.1.5 Tensor Shape Table

We shall keep track of the shapes of the tensors flowing through the network, via a table. As we go on adding new components, we shall update this table.

Layer	Input Shape	Output Shape
Input Embeddings	$(b, \text{seq\_len})$	$(b, \text{seq\_len}, d_{\text{model}})$

Table 1.1: Tensor shape table

Table 1.1 shows input and output shapes for the `Input Embeddings` layer. Here, b is the batch size, seq_len is the sequence length (number of tokens/words in each input sequence), and d_{model} is the dimensionality of the word embeddings. The output embeddings are scaled by $\sqrt{d_{\text{model}}}$.

1.1.6 Summary

Key Takeaways: Embeddings

- Embeddings are dense vectors that represent words in a high-dimensional space.
- They capture semantic relationships between words/tokens.
- During training, embeddings are updated to better represent the data.
- The embedding layer is a learnable parameter that gets optimized during training.
- The embedding layer is a PyTorch `nn.Module`, so it can be used in a neural network.

With `InputEmbeddings` in place, we move to the next component: positional encoding.

[1]We shall look at the process of creating vocabulary from text corpus (a.k.a. `tokenization`) in the later sections.

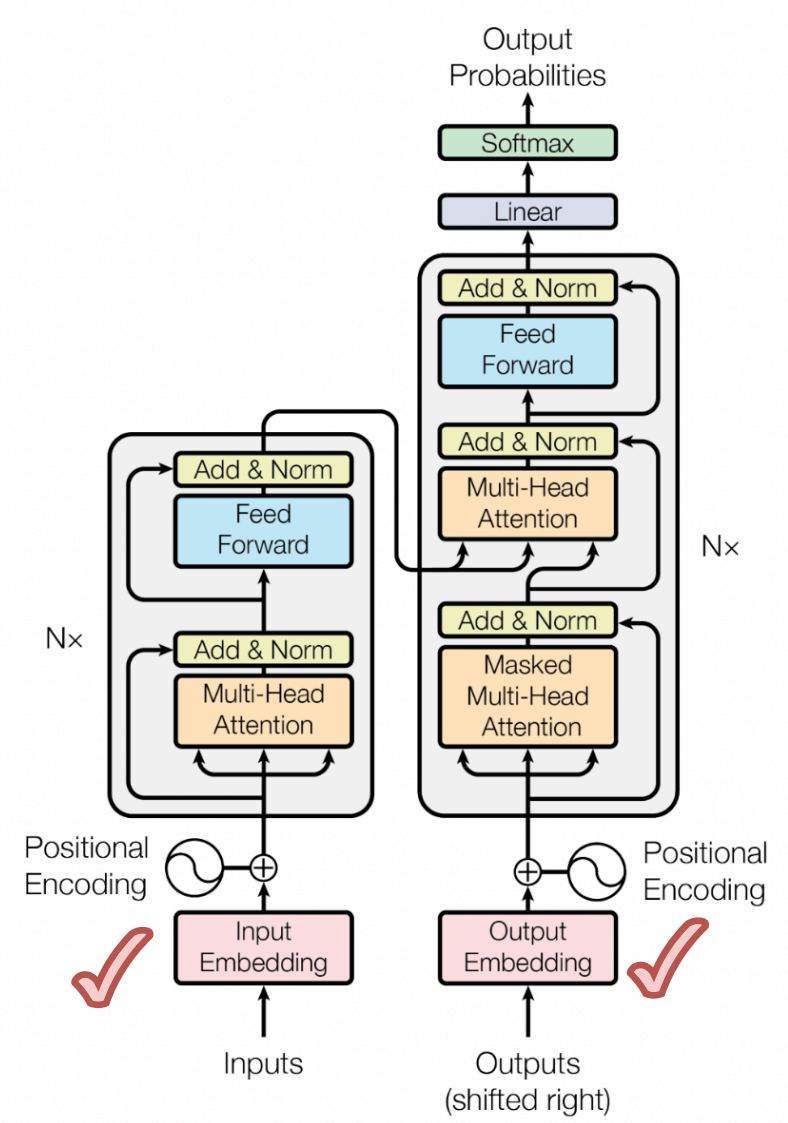


Figure 1.5: Embedding class completed. [1]

1.2 Positional Encoding

Imagine you're at your favorite concert. As the band plays, you instinctively know what comes next in the song - the chorus after the verse, the bridge after the chorus. This natural understanding of sequence and order is something we take for granted, but it's a capability we need to explicitly teach our AI models. Welcome to the world of **Positional Encoding (PE)**!

In the realm of language, the order of words can change the meaning entirely. Consider the sentences "The cat chased the mouse" and "The mouse chased the cat". The words are the same, but their order changes the story. To help our AI understand this, we introduce `Positional Encoding`.

When we convert words into embeddings, we lose the information about their position in a sentence. Unlike recurrent neural networks (RNNs[2]) and long short-term memory (LSTM[3])

[2] `https://aws.amazon.com/what-is/recurrent-neural-network/`
[3] `https://developer.nvidia.com/discover/lstm`

networks, which process sequences step-by-step and naturally maintain position information through their hidden states, transformers process all words simultaneously. This parallel processing (which we'll explore in detail when we discuss the attention mechanism) is powerful but means we need to explicitly add positional information. Positional encoding solves this by injecting position information directly into our embeddings. Thus, we can consider `Positional Encoding (PE)` like a musical score, providing a sense of timing and order to the notes (words) in our song (sentence).
Before looking at solutions, let's establish what makes an ideal Positional Encoding system:

- **R1: Unique encoding for each position:** Think of it like assigning unique ID badges to workers. Whether you have 10 or 100 employees, each person's ID should remain consistent. Similarly, position 2 should have the same encoding whether it's in a short sentence or a long one. This consistency helps the model learn stable patterns.
- **R2: Linear relation between positions:** Imagine a number line where you can easily jump forward or backward. If you know someone is at position p, you should be able to easily calculate where position p+k would be. This predictability makes it easier for our model to understand relative positions.
- **R3: Ability to handle unexpected lengths:** Just like how a GPS system should work whether you're traveling 1 mile or 1000 miles, our encoding should work for sequences longer than those seen during training. We can't let our model be limited by its training examples.
- **R4: Simple, learnable pattern:** The encoding should follow a straightforward formula that our model can easily grasp. Think of it like teaching multiplication tables - simple patterns are easier for the model to learn and apply.
- **R5: Works across different dimensions:** Like a Swiss Army knife, our encoding should be versatile enough to work whether we're dealing with 1D sequences (like text), 2D data (like images), or even higher dimensions.

Reading the above conditions the first thought that will come to anyone's mind will be, "let's add the position of the word". This naive solution will work for small sentences. But for longer sentences like let's say for some essay with 5000 words, adding position 5000 to the word embedding can lead to exploding or vanishing gradients [4]. With this new knowledge, a natural follow up might be to normalize the position value by $\frac{1}{N}$. This constrains the values between 0 and 1, but introduces another problem. If we choose $\frac{1}{N}$ to be the length of the current sequence, then the position values will be completely different for each sequence of differing lengths, violating R1. Another method would be, instead of adding our (potentially normalized) integer position to each component of the embedding, we could instead convert the word into its binary representation and add it to our embedding dimension, which will give us unique encodings irrespective of sequence lengths. But, the drawback being, the encoding vectors formed are not smooth, continuous or predictable. Hence, difficult to optimize. There is a beautiful blog by huggingface, which I would recommend to read for better animated visualizations [5].

Can you think of any functions that are smooth, continuous and repetitive? Right! `Sinusoidal` functions. These functions are also chosen by authors of the attention paper. Understanding sinusoidal positional encodings can indeed be a bit challenging at first, but let's break it down step by step. The goal of positional encodings in the transformer paper was to inject information about the position of tokens in a sequence into the model, since the self-attention mechanism itself does not inherently capture any notion of order.

[4] `https://en.wikipedia.org/wiki/Vanishing_gradient_problem`
[5] `https://huggingface.co/blog/designing-positional-encoding`

1.2.1 Understanding Sinusoidal Positional Encodings

One might think: why not just use a simple function like `sine` to encode positions? The problem with using just sine is that it does not provide enough information for the model to learn relative positions efficiently. The Transformer should be able to infer relative positions easily. If we use both `sine` and `cosine`, we can express shifts in position using a simple `linear transformation`. This means that for a given position shift k, we can express the new encoding in terms of the old encoding using a matrix multiplication:

$$M \ @ \ \begin{bmatrix} \sin(\omega_i p) \\ \cos(\omega_i p) \end{bmatrix} = \begin{bmatrix} \sin(\omega_i (p+k)) \\ \cos(\omega_i (p+k)) \end{bmatrix}$$

where:

- p is the original position
- k is the shift
- ω_i is the frequency of the sinusoid at dimension i.

This property ensures that the model can infer relative positions *without explicitly memorizing them*, making it *generalizable to longer sequences*. If we used only sine, this linear shift property wouldn't hold because shifting sine alone would require nonlinear operations to reconstruct the full position.

Finding the Transformation Matrix

Let's prove that relative positions can be captured through linear transformations - a key property that makes sinusoidal encodings so effective. To find this transformation matrix, we can express it as a general 2×2 matrix with unknown coefficients u_1, v_1, u_2, and v_2:

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \ @ \ \begin{bmatrix} \sin(\omega_i p) \\ \cos(\omega_i p) \end{bmatrix} = \begin{bmatrix} \sin(\omega_i (p+k)) \\ \cos(\omega_i (p+k)) \end{bmatrix}$$

By applying the trigonometric addition theorem to the right-hand side, we can expand this into:

$$\begin{bmatrix} u_1 & v_1 \\ u_2 & v_2 \end{bmatrix} \ @ \ \begin{bmatrix} \sin(\omega_i p) \\ \cos(\omega_i p) \end{bmatrix} = \begin{bmatrix} \sin(\omega_i p)\cos(\omega_i k) + \cos(\omega_i p)\sin(\omega_i k) \\ \cos(\omega_i p)\cos(\omega_i k) - \sin(\omega_i p)\sin(\omega_i k) \end{bmatrix}$$

This expansion gives us a system of two equations by matching coefficients:

$$u_1 \sin(\omega_i p) + v_1 \cos(\omega_i p) = \cos(\omega_i k)\sin(\omega_i p) + \sin(\omega_i k)\cos(\omega_i p)$$

$$u_2 \sin(\omega_i p) + v_2 \cos(\omega_i p) = -\sin(\omega_i k)\sin(\omega_i p) + \cos(\omega_i k)\cos(\omega_i p)$$

By comparing terms with $\sin(\omega_i p)$ and $\cos(\omega_i p)$ on both sides, we can solve for the unknown coefficients:

$$u_1 = \cos(\omega_i k) \quad v_1 = \sin(\omega_i k)$$

$$u_2 = -\sin(\omega_i k) \quad v_2 = \cos(\omega_i k)$$

These solutions give us our final transformation matrix M_k:

$$M_k = \begin{bmatrix} \cos(\omega_i k) & \sin(\omega_i k) \\ -\sin(\omega_i k) & \cos(\omega_i k) \end{bmatrix}$$

If you've done any game programming before, you might recognize this as the 2D Rotation Matrix! Interestingly, while the connection between sinusoidal encodings and rotations was implicit in the original Transformer paper [1], it wasn't until RoPE (Rotary Position Embedding) [3] that this connection was explicitly used for position encoding.

Mathematical Definition of Sinusoidal Encodings

Each position is assigned a unique vector based on sine and cosine functions applied at different frequencies:

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

where:

- pos is the token's position in the sequence.
- i is the dimension index of the encoding vector.
- d_{model} is the model's hidden dimension.
- 10000 is a constant scaling factor.

Each position gets a different `pattern of waves`, ensuring that no two positions are encoded identically. This leads us to the concept of geometric progression, which determines how these frequencies are selected.

Geometric Progression in Positional Encodings

To understand why these specific frequencies were chosen, we need to explore the concept of geometric progression and its role in capturing positional information at different scales. A `geometric progression` is a sequence where each term is obtained by multiplying the previous term by a constant factor. For example:

$$2, 4, 8, 16, 32, \ldots$$

where each term is obtained by multiplying the previous one by 2.
The frequencies used in the encoding are defined as:

$$\omega_i = \frac{1}{10000^{\frac{2i}{d_{model}}}}$$

Since the wavelength is the inverse of frequency:

$$\lambda_i = 10000^{\frac{2i}{d_{model}}}$$

This means that the sequence of wavelengths follows a geometric progression with a ratio:

$$\frac{\lambda_{i+1}}{\lambda_i} = 10000^{\frac{2}{d_{model}}}$$

Since this ratio remains `constant`, the sequence of wavelengths `scales smoothly`, ensuring that different encoding dimensions capture both `local` and `global` positional patterns.

How Does This Affect the Encodings?

Now that we understand why the encoding follows a geometric progression, let's look at how this affects the Transformer's ability to understand positions:

1. **Different Scales of Information:**
   - Higher dimensions capture low-frequency (long-distance) relationships.
   - Lower dimensions capture high-frequency (local) relationships.
2. **Positional Shift Invariance:**
   - Since the encoding is based on sine and cosine, relative shifts in position can be captured by simple linear transformations.
3. **Efficient Learning:**
   - The model does not need to learn positional embeddings from scratch.
   - The structured, wave-like encodings help the attention mechanism generalize well to unseen sequence lengths.

Thus, Sinusoidal positional encodings are a simple yet powerful method to provide transformers with positional awareness. By using both sine and cosine functions, they allow linear transformations to infer relative positions easily. Their geometrically progressing frequencies ensure that different encoding dimensions capture short-term and long-term dependencies, making them well-suited for NLP tasks.

1.2.2 Coding Positional Encoding

Code:

```
class PositionalEncoding(nn.Module):
    """
    This class implements positional encoding for transformer models, where:
    - Each position in a sequence is assigned a unique encoding.
    - These encodings enable the model to understand the order of words.
    - The encoding is derived from sine and cosine functions of varying
    ↪  frequencies.
    - The encoding is added to the word embeddings to provide positional
    ↪  information.
    - The encoding remains fixed and does not change during training.
    """

    def __init__(self, d_model: int, seq_len: int, dropout: float = 0.1):
        """
        Initializes the PositionalEncoding class.

        Parameters:
        d_model (int): The dimension of the word embeddings.
        seq_len (int): The maximum length of a sequence expected.
        dropout (float): The dropout rate applied to the positional encodings.
        """
        super().__init__()

        self.d_model = d_model
        self.seq_len = seq_len
```

```
        self.dropout = nn.Dropout(dropout)

        # Create a matrix of shape (seq_len, d_model) to hold the positional
        ↪  encodings
        # We need vectors of d_model dimension and we need a total of seq_len such
        ↪  vectors
        pe = torch.zeros(seq_len, d_model)
        # Create a sequence of positions from 0 to seq_len
        # We need to unsqueeze it to make it a column vector
        position = torch.arange(0, seq_len, dtype=torch.float).unsqueeze(1) #
        ↪  shape: (seq_len, 1)

        # Create a sequence of frequencies for the sine and cosine waves
        # We use a geometric progression from 1 to 10000
        # Value calculated in log space for numerical stability
        div_term = torch.exp(torch.arange(0, d_model, 2).float() *
        ↪  (-np.log(10000.0) / d_model))

        # Apply sine to even indices and cosine to odd indices
        # We use a slice to apply the sine function to every second element
        # The first slice `:` means we apply it to all rows
        # The second slice `0::2` means we apply it to every second column (even
        ↪  indices)
        pe[:, 0::2] = torch.sin(position * div_term)
        # Apply cosine to odd indices
        # The second slice `1::2` means we apply it to every second column (odd
        ↪  indices)
        pe[:, 1::2] = torch.cos(position * div_term)

        # Add a batch dimension
        pe = pe.unsqueeze(0) # shape: (1, seq_len, d_model)
        # Register the positional encodings as a buffer
        # Buffers are not parameters and are not updated during training
        # They are used for storing constants or precomputed values
        self.register_buffer('pe', pe)

    def forward(self, x: torch.Tensor):
        """
        Adds the positional encodings to the input embeddings.

        Parameters:
        x (torch.Tensor): The input tensor containing word embeddings.

        Returns:
        torch.Tensor: The input embeddings with positional encodings added.
        """
        # Add the positional encodings to the word embeddings
        # We use a slice to get the encodings for the current sequence length
        # pe is expanded to match batch size
        # x shape: (batch_size, seq_len, d_model)
        # pe shape: (1, seq_len, d_model)
        x = x + (self.pe[:, :x.shape[1], :]).requires_grad_(False) # shape:
        ↪  (batch_size, seq_len, d_model)
        # Apply dropout to the positional encodings
        return self.dropout(x)
```

One thing to note is that positional encoding is not updated during training; hence, it is

registered as a buffer.

1.2.3 Visualisation

The following plots show positional encodings of varying dimensions. Fig 1.6 illustrates how different dimensions capture different frequency patterns, while Fig 1.7 shows how positional encodings are combined with word embeddings to create position-aware representations.

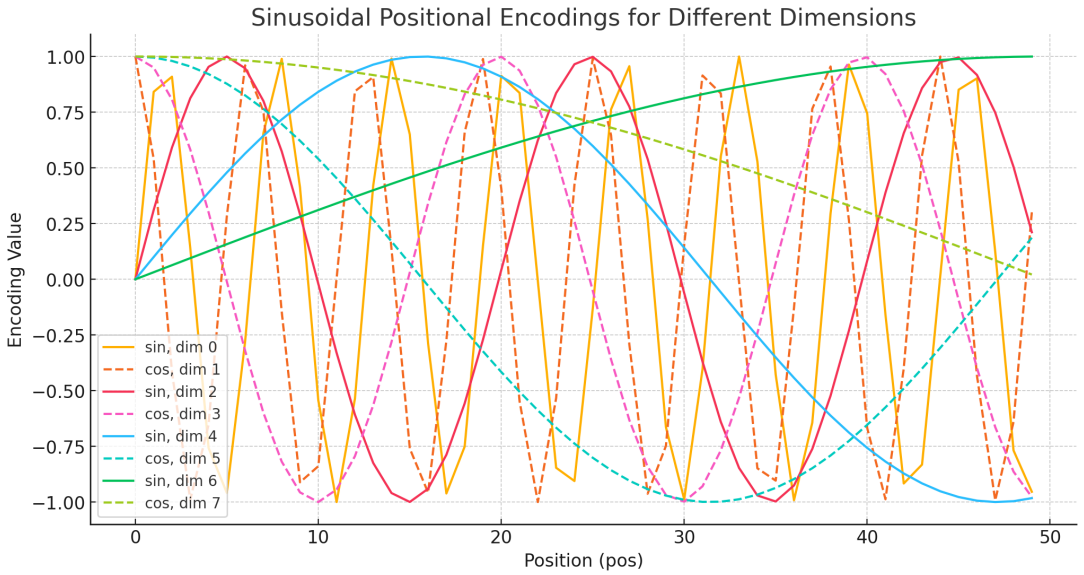


Figure 1.6: Positional encodings for different dimensions

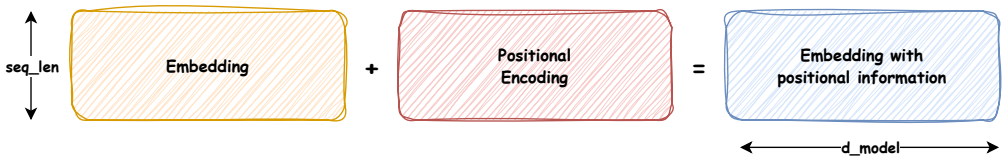


Figure 1.7: Position aware embedding vector

1.2.4 Tensor Shape Table

Let's add positional encoding to our previous table 1.1.

Layer	Input Shape	Output Shape
Input Embeddings	$(b, \text{seq\_len})$	$(b, \text{seq\_len}, d_{\text{model}})$
Positional Encodings	$(b, \text{seq\_len}, d_{\text{model}})$	$(b, \text{seq\_len}, d_{\text{model}})$

Table 1.2: Tensor shape table

1.2.5 Summary

Key Takeaways: Positional Encoding

- Positional encodings are added to word embeddings to give each word a unique position in the sequence
- They use sine and cosine functions of different frequencies to create unique position signatures
- The geometric progression of frequencies enables capturing both short and long-range dependencies
- The sinusoidal design allows relative positions to be captured through simple linear transformations
- These encodings are fixed (not learned) yet provide reliable positional information across sequence lengths

With positional encoding in place, we turn to the most important component of the chapter: the attention mechanism.

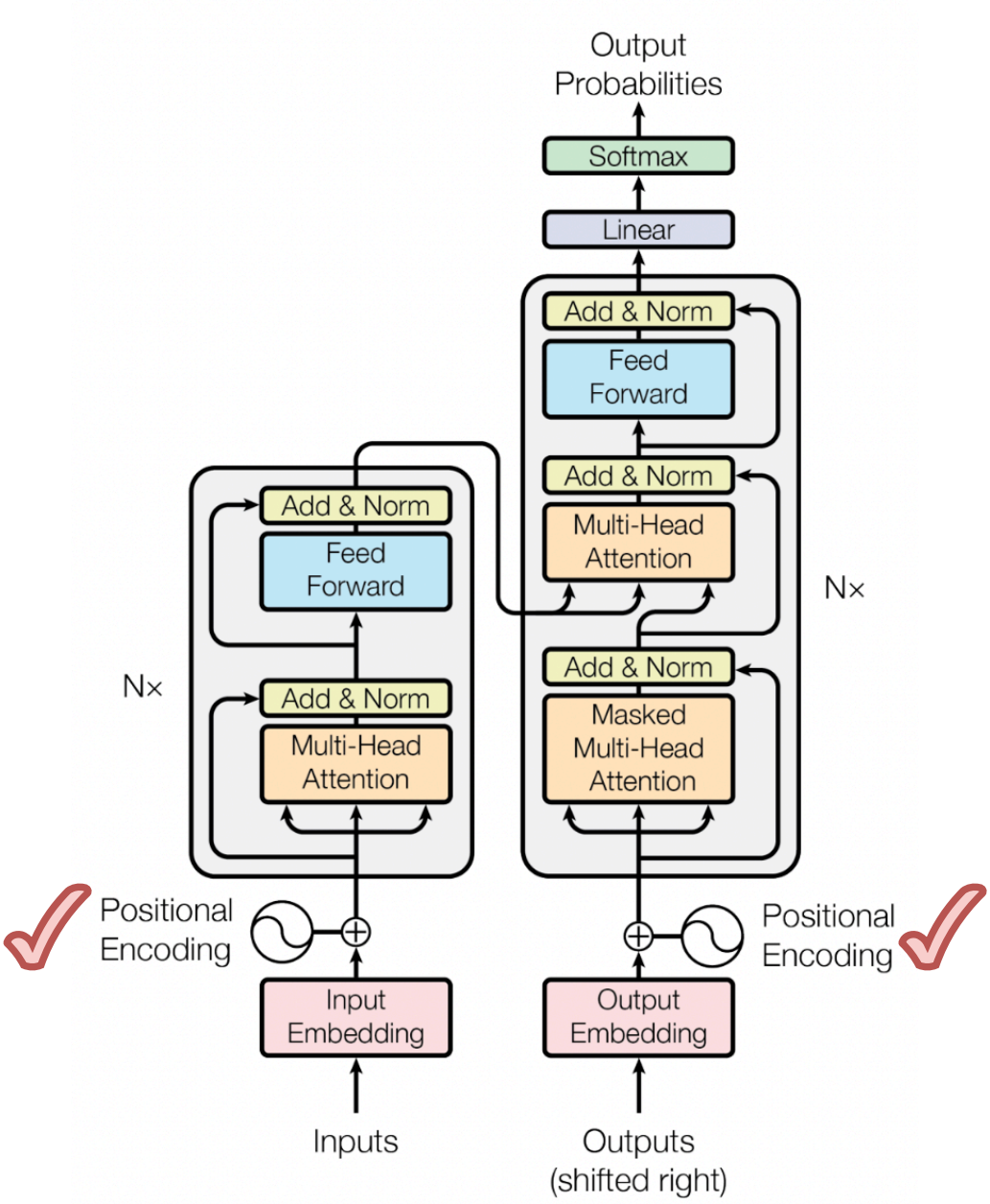


Figure 1.8: Positional Encoding class completed. [1]

1.3 Attention Mechanism - The Heart of Our Transformer

Imagine you're in a room full of people speaking different languages. As a translator, you need to understand not just individual words, but how they connect to form meaning. This was exactly the challenge that early AI translation systems faced! As shown in Fig 1.9, accurate translation requires attending to different words in the original sentence when generating each word in the target language.

Think of early translation AI (RNNs) as someone trying to remember a long story by memorizing one sentence at a time. By the end, they might forget important details from the beginning! This is known as the long-term dependency problem in RNN encoder-decoder systems.

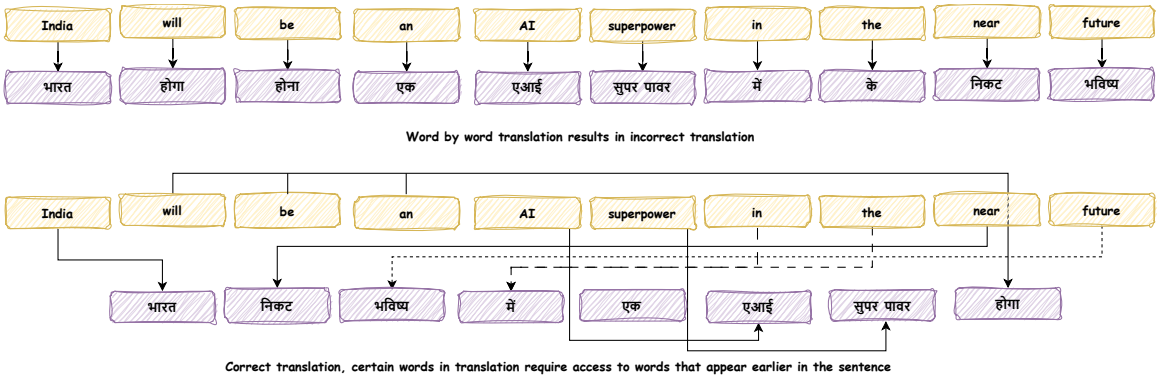


Figure 1.9: Accurate translation requires attending to different words in the original sentence

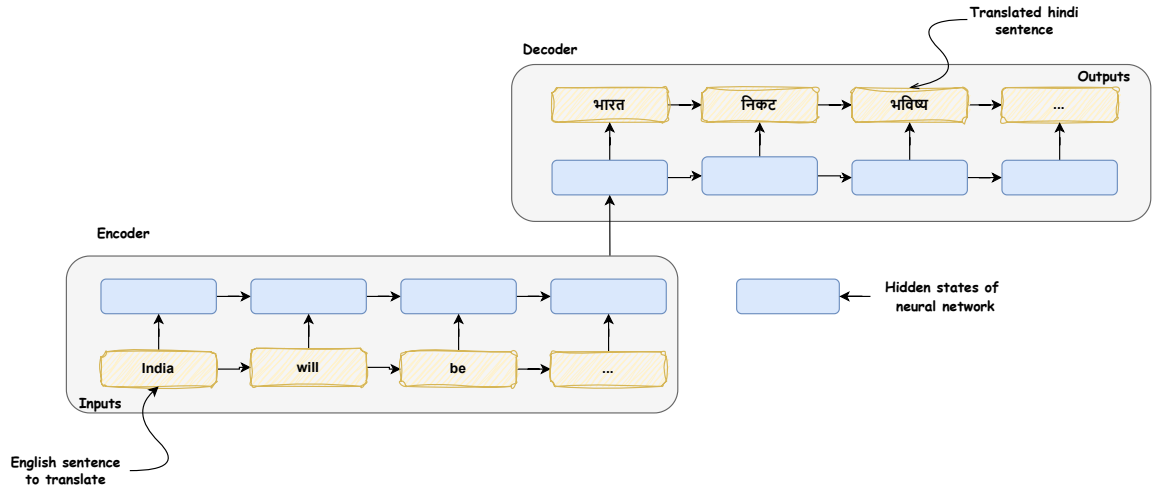


Figure 1.10: Encoder-Decoder RNN architecture for English to Hindi translation

As shown in Fig 1.10, the encoder would read the input sentence and compress all the information into a fixed-length hidden state - like trying to squeeze a whole book into a single page summary. The decoder would then generate the translation using only this compressed information. Naturally, this led to significant information loss, especially for longer sequences!

Then came a key insight: what if, instead of compressing everything into one hidden state, the decoder could look back at the original text whenever needed? This was the Bahdanau attention mechanism [4]. Just as a human translator glances back at different parts of the source while writing the translation, this mechanism let the decoder "pay attention" to relevant parts of the input as it generated each output token. The next leap came in 2017 with the paper "Attention Is All You Need" [1], which questioned whether the recurrence in RNNs was needed at all. The transformer, built around self-attention, let the model look at all input tokens at once and learn their relationships directly.
This mattered for three reasons:

1. It enabled parallel processing of all input tokens, dramatically improving computational efficiency.

2. It could directly model relationships between any tokens, regardless of their positional distance.

3. It learned attention patterns dynamically, mimicking human reading comprehension.

In the following section, we'll explore how self-attention works, but remember - it's all about helping our AI understand context, just like how you naturally understand the meaning of words based on their surroundings! We'll learn the attention mechanism in increasing levels of complexity: `Self-attention with no trainable weights` → `Self-attention with trainable weights` → `Causal attention` → `Multi-head attention`.

1.3.1 Self-attention with no trainable weights

Imagine you're at a party following a conversation. When someone speaks, your mind naturally connects their words to relevant points made earlier. This is exactly how self-attention works! In self-attention, the term 'self' emphasizes the mechanism's ability to compute attention weights by relating different positions within a single input sequence. It evaluates and learns the relationships and dependencies among various parts of the input itself. This differs from traditional attention mechanisms, which focus on relationships between elements of two distinct sequences, such as in sequence-to-sequence models where attention might occur between an input sequence and an output sequence. **Our objective**: To create a context-aware understanding of each word by considering all other words in the sentence. Let's start with a simple scenario. Imagine we have 4 words/tokens[6] (i.e., $seq\_len = 4$) in our sentence, and each word is represented by a rich description (an embedding) of 512 features (i.e., $d_{model} = 512$).
Code:

```
torch.manual_seed(123)
inputs = torch.rand(4, 512)
print(f"Shape of inputs: {inputs.shape}")
print(f"Inputs: {inputs}")
```

Output:

```
Shape of inputs: torch.Size([4, 512])
Inputs: tensor([[0.2961, 0.5166, 0.2517,  ..., 0.9959, 0.6785, 0.3981],
        [0.5921, 0.0056, 0.5577,  ..., 0.9115, 0.5589, 0.8239],
        [0.1299, 0.0250, 0.6655,  ..., 0.5465, 0.4532, 0.7598],
        [0.6945, 0.2478, 0.4111,  ..., 0.7706, 0.8131, 0.1889]])
```

Think of this as having 4 words, where each word is described by 512 different characteristics!

Computing Attention Scores: How Words Connect

Now to the central question. Just as you find connections between different parts of a conversation, we need our model to find connections between words. We do this through `attention scores`.

Imagine comparing two books - you might look at their genres, themes, and writing styles to determine their similarity. That's exactly what we do with words using a `dot product`! A dot product is a mathematical operation that multiplies two vectors element-wise and sums the products. It serves as a measure of similarity because it quantifies how closely two vectors align: a higher dot product indicates greater alignment or similarity between the vectors. In self-attention mechanisms, the dot product determines how much each element in a sequence should focus on, or 'attend to,' other elements: the higher the dot product, the stronger the attention between two elements. Let's see this in action by using our input vector at position 2 (`inputs[2]`) as the query vector.

[6] We'll use words and tokens interchangeably. In upcoming sections, we'll discuss tokenization briefly. For now, understand that tokens are simply a way to split sentences into smaller units.

Code:

```
query = inputs[2]
print(f"Shape of query: {query.shape}")
# Print the first 10 and last 10 values of the query vector
print(f"Query: {query[:10]}...")
```

Output:

```
Shape of query: torch.Size([512])
Query: tensor([0.1299, 0.0250, 0.6655, 0.2683, 0.9916, 0.5703, 0.3389, 0.9356,
↪  0.9790, 0.8712])...
```

Now, let's compute the attention scores, of query with every other input.
Code:

```
# Compute the attention scores
attention_scores = torch.empty(inputs.shape[0])

# Iterate over all the input vectors
for i in range(inputs.shape[0]):
    key = inputs[i]
    attention_scores[i] = torch.dot(query, key)

print(f"Attention scores: {attention_scores}")
```

Output:

```
Attention scores: tensor([134.1449, 129.8451, 177.2513, 130.5724])
```

You can also see that the dot product of `query` is maximum with itself, i.e., `inputs[2]`.

Normalizing Scores: Making Comparisons Fair

Think of normalization like converting different currencies to a common one. We need all our attention scores to be on the same scale (between 0 and 1) and sum up to 1. This helps in controlling the amount of attention each element receives, preventing any one element from dominating the attention distribution.

We use a special function called `softmax` - think of it as a friendly referee that makes sure everyone plays fair. The `softmax` function takes a vector of real numbers and transforms them into probabilities that sum to 1. Mathematically, for a vector x with n elements, the softmax function is defined as:

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

where,

- x_i is the input value at position i
- e is Euler's number (approximately 2.71828)

This approach is better at managing extreme values and offers more favorable gradient properties during training. In addition, the softmax function ensures that the attention weights are always positive. This makes the output interpretable as probabilities or relative importance, where higher weights indicate greater importance.

Let's see the softmax function in action with our attention scores:

Code:

```
normalized_attention_scores = torch.softmax(attention_scores, dim=0)
print(f"Normalized attention scores: {normalized_attention_scores}")
print(f"Sum of normalized attention scores:
↪  {torch.sum(normalized_attention_scores)}")
```

Output:

```
Normalized attention scores: tensor([1.9016e-19, 2.5808e-21, 1.0000e+00,
↪  5.3406e-21])
Sum of normalized attention scores: 1.0
```

Notice how the softmax function has transformed our raw attention scores into probabilities. The highest score (corresponding to the query matching itself) receives almost all the attention weight (since the dot product is maximum here), while other positions receive very small weights. This is exactly what we want - the model can now focus most of its attention on the most relevant parts of the input!

Sidetrack: Understanding Dimensions

Let's pause to clarify what summing across a particular dimension means. When you compute a sum in PyTorch, you sum all the elements along the specified dimension (axis), collapsing that dimension and reducing the tensor's rank.

The meaning of $dim = 0$ in PyTorch (or other tensor libraries) depends on the rank (number of dimensions) of the tensor, as it always refers to the first dimension of the tensor. Let's break it down for `1D`, `2D`, and `3D` tensors.

1D Tensor: A 1D tensor is essentially a vector, with only one dimension (a single list of elements).

```
x = torch.tensor([1, 2, 3]) # Shape: (3,) → One dimension with size 3.
x.sum(dim=0)  # Output: tensor(6) = 1 + 2 + 3
```

Here, $dim = 0$ is the only dimension, so it's effectively summing the entire tensor.

2D Tensor: A 2D tensor is a matrix, with two dimensions: rows (dim=0) and columns (dim=1).

```
# Shape: (2, 3) → 2 rows and 3 columns.
x = torch.tensor([[1, 2, 3],
                  [4, 5, 6]])
```

1. Summing along $dim = 0$:
    - Collapse the rows, summing elements column-wise.

- ```
  x.sum(dim=0)  # Output: tensor([5, 7, 9])
  ```

- This corresponds to summing vertically: $[1+4, 2+5, 3+6]$ to $[5, 7, 9]$

2. Summing along $dim = 1$:

   - Collapse the columns, summing elements row-wise.

   - ```
     x.sum(dim=1)  # Output: tensor([6, 15])
     ```

   - This corresponds to summing horizontally: $[1+2+3, 4+5+6]$ to $[6, 15]$

3D Tensor: A 3D tensor can be thought of as a stack of 2D matrices (like a 3D array).

```
# Shape: (2, 2, 3) → 2 'matrices', each with 2 rows and 3 columns.
x = torch.tensor([[[1, 2, 3],
                   [4, 5, 6]],

                  [[7, 8, 9],
                   [10, 11, 12]]])
```

1. Summing along $dim = 0$:

   - Collapse the first dimension (the stack of matrices), summing elements across the 'stack' for each corresponding position in the matrix.

   - ```
     x.sum(dim=0)
     ```

   - Sum:

     ```
     tensor([[1+7, 2+8, 3+9],
             [4+10, 5+11, 6+12]])
     ```

   - Result:

     ```
     tensor([[8, 10, 12],
             [14, 16, 18]])
     ```

2. Summing along $dim = 1$:

   - Collapse the second dimension (rows), summing elements row-wise for each matrix in the stack.

   - ```
     x.sum(dim=1)
     ```

   - Sum:

```
tensor([[1+4, 2+5, 3+6],
        [7+10, 8+11, 9+12]])
```

- Result:

```
tensor([[5, 7, 9],
        [17, 19, 21]])
```

3. Summing along $dim = 2$:

- Collapse the third dimension (columns), summing elements column-wise for each row in each matrix.

- ```
  x.sum(dim=2)
  ```

- Sum:

```
tensor([[1+2+3, 4+5+6],
        [7+8+9, 10+11+12]])
```

- Result:

```
    tensor([[ 6, 15],
            [24, 33]])
```

Summary Table

Table 1.3: Summary Table for Summing Across `dim=0`

Tensor Shape	**dim=0 Summation Description**	**Resulting Shape**
1D $(n,)$	Sum all elements in the vector	Scalar $(1,)$
2D (m, n)	Sum elements column-wise (across rows)	$(n,)$
3D (p, m, n)	Sum elements across the stack of matrices	(m, n)

Table 1.4: Summary Table for Summing Across `dim=1`

Tensor Shape	**dim=1 Summation Description**	**Resulting Shape**
1D $(n,)$	Not applicable	Not applicable
2D (m, n)	Sum elements row-wise (across columns)	$(m,)$
3D (p, m, n)	Sum elements across rows for each matrix	(p, n)

- $dim = 0$ always refers to the outermost dimension (the first one in the shape).
- $dim = -1$ always refers to the last dimension of the tensor
```

Table 1.5: Summary Table for Summing Across `dim=2`

Tensor Shape	**dim=2 Summation Description**	**Resulting Shape**
1D $(n,)$	Not applicable	Not applicable
2D (m, n)	Not applicable	Not applicable
3D (p, m, n)	Sum elements across columns for each matrix	(p, m)

Table 1.6: Summary Table for Summing Across `dim=-1` (Last Dimension)

Tensor Shape	**dim=-1 Summation Description**	**Resulting Shape**
1D $(n,)$	Sum all elements in the vector	Scalar $(1,)$
2D (m, n)	Sum elements row-wise (across columns)	$(m,)$
3D (p, m, n)	Sum elements across columns for each matrix	(p, m)

- The dimension being summed gets collapsed, reducing the tensor's rank by 1. Other dimensions remain intact.
- For a 1D tensor: $dim = -1$ is equivalent to $dim = 0$
- For a 2D tensor: $dim = -1$ is equivalent to $dim = 1$
- For a 3D tensor: $dim = -1$ is equivalent to $dim = 2$

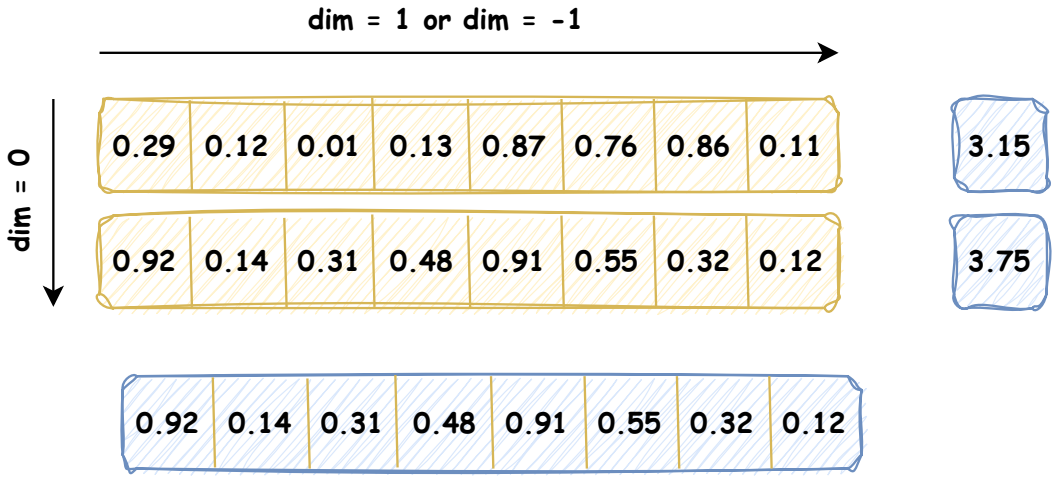


Figure 1.11: Visual depiction of summing across $dim = 0$ or $dim = 1$ *or* -1 for a 2D tensor

Creating Context: Bringing It All Together

Now we put the pieces together. We create a `context vector` by combining all input vectors according to their attention scores. Think of it like a weighted playlist: each song (word) contributes to the overall feel (meaning) in proportion to how well it fits the query. The context vector for our query is a weighted combination of all input vectors. Mathematically:

$$\text{context} = \sum_{i=1}^{n} \alpha_i v_i$$

where,

- α_i is the normalized attention score for position i
- v_i is the input vector at position i
- n is the number of input vectors

Let's look at the shapes of our tensors:

```
Shape of inputs: torch.Size([4, 512])
Shape of normalized attention scores: torch.Size([4])
```

This weighted combination can be efficiently computed using matrix multiplication. The normalized attention scores act as weights that determine how much each input vector contributes to the final context vector.
Code:

```
context_vector = normalized_attention_scores @ inputs
print(f"Shape of context vector: {context_vector.shape}")
print(f"Context vector: {context_vector[:10]}...")
```

Output:

```
Shape of context vector: torch.Size([512])
Context vector: tensor([0.1299, 0.0250, 0.6655, 0.2683, 0.9916, 0.5703, 0.3389,
↪  0.9356, 0.9790, 0.8712])...
```

Notice how the resulting context vector has the same dimensionality as our input vectors (512), but now contains information from all input vectors, weighted by their relevance to our query. Since our normalized attention scores were heavily focused on the query vector itself (as we saw in the previous section), the context vector is very similar to our original query vector.

Computing Context Vectors for all Queries

Imagine you're hosting a party where everyone (each word) needs to understand their relationship with every other guest (other words). Instead of doing this one person at a time, we can efficiently do this for everyone simultaneously! So far we have:

- Take a single query vector
- Computed its attention scores with all input vectors
- Normalized the attention scores
- Created its context vector

Let's scale this up efficiently! Our inputs shape is:

```
Shape of inputs: torch.Size([4, 512])
```

Now, using matrix multiplication, attention-scores can be calculated as:
Code:

```
# inputs is of shape (seq_len, d_model) @ (d_model, seq_len) = (seq_len, seq_len)
attention_scores = inputs @ inputs.T
print(f"Shape of attention scores: {attention_scores.shape}")
print(f"Attention scores: {attention_scores}")
```

Output:

```
Shape of attention scores: torch.Size([4, 4])
Attention scores: tensor([
        [172.0800, 126.5615, 134.1449, 126.5499],
        [126.5615, 169.7253, 129.8451, 128.4887],
        [134.1449, 129.8451, 177.2513, 130.5724],
        [126.5499, 128.4887, 130.5724, 168.5083]])
```

Notice how the attention scores form a square matrix where each row represents how one word attends to all others. The diagonal values are typically highest because words have the strongest attention with themselves. We then normalize the attention scores to get the normalized attention scores.

Code:

```
normalized_attention_scores = torch.softmax(attention_scores, dim=-1)
print(f"Shape of normalized attention scores: {normalized_attention_scores.shape}")
# Verify that the sum of the normalized attention scores is 1
print(f"Sum of normalized attention scores: {torch.sum(normalized_attention_scores,
↪  dim=-1)}")
```

Output:

```
Shape of normalized attention scores: torch.Size([4, 4])
Sum of normalized attention scores: tensor([1., 1., 1., 1.])
```

Now, as in previous section, we can directly multiply the normalized attention scores with the input vectors to get the context vectors.

Code:

```
context_vectors = normalized_attention_scores @ inputs
print(f"Shape of context vectors: {context_vectors.shape}")
```

Output:

```
Shape of context vectors: torch.Size([4, 512])
```

And there we have it! Each word now has a rich, context-aware representation that takes into account all other words in the sentence. Its shape is the same as that of our embedding vector.

Summary:

1. Self-attention helps words 'understand' their context by learning from their neighbors
2. Each word computes attention scores with every other word, creating a web of relationships

3. Softmax normalization ensures these relationships are proportional and meaningful
4. The resulting context vectors capture rich, bidirectional relationships between words
5. Matrix operations make this process computationally efficient

Next, we add trainable weights so the model can learn what to attend to.

1.3.2 Self-attention with trainable weights

Remember our party conversation analogy? Now imagine you're not just listening to conversations, but you're actually learning how to be a better listener over time. That's what we're adding now - the ability for our AI to learn and adapt!

So far, we've been computing attention using the raw input vectors directly. While this works, it's quite limited. The real power comes when we let the model learn how to transform these vectors to better capture relationships. Our next step will be to implement the self-attention mechanism used in the original transformer architecture. This self-attention mechanism is also called `scaled dot-product attention`.

We will implement this step by step by introducing three trainable weight matrices W_q, W_k, and W_v. These three matrices are used to project each embedded input vector into query, key, and value vectors, respectively. Think of these transformations as teaching our model to ask better questions (queries), create better indexes (keys), and extract better information (values) from the input.

Why query, key, and value?

The terms `key`, `query`, and `value` in the context of attention mechanisms are borrowed from the domain of information retrieval and databases, where similar concepts are used to store, search, and retrieve information. Think of this like a sophisticated search engine. Where,

- Query (Q): Search query denoting what you're looking for (like your Google search)
- Key (K): The index cards in a library catalog
- Value (V): The actual books on the shelves

Generally, the weight matrices have dimensions same as that of the embedding dimension of the input vectors. As our inputs are of dimension (seq_len, d_{model}) and the dimension of the weight matrices should be (d_{model}, d_{model}). In our example, d_{model} is `512`, and the dimension of the inputs is `(4, 512)`. For understanding purposes, instead of using the same input and output dimensions for the weight matrices, we will use different dimensions for the weight matrices, denoted by d_{in} and d_{out}. For our example, let's consider d_{in} as 512 i.e, dimension of our embedding, d_{out} as 64.

Code:

```
d_in = 512 # inputs.shape[1]
d_out = 64

# Initialize the weight matrix
torch.manual_seed(123)
W_query = torch.nn.Parameter(torch.rand(d_in, d_out))
W_key   = torch.nn.Parameter(torch.rand(d_in, d_out))
W_value = torch.nn.Parameter(torch.rand(d_in, d_out))

print(f"Shape of weight matrix: {W_query.shape}")
```

Output:

```
Shape of weight matrix: torch.Size([512, 64])
```

Now, `Query (Q)`, `Key (K)`, and `Value (V)` vectors are computed by matrix multiplication of `inputs` with respective weight matrices. Here, the shape of the `inputs` is $(seq\_len, d_{model}) \equiv (4, 512)$, while the shape of the weight matrices is $(d_{in}, d_{out}) \equiv (512, 64)$, with $d_{model} = d_{in}$.

Code:

```
# Compute the query, key and value vectors for inputs
# here, d_model = d_in
# (seq_len, d_model) @ (d_in, d_out) -> (seq_len, d_out)

query = inputs @ W_query
print(f"Shape of query vector: {query.shape}")

# Compute the key vector for inputs
key = inputs @ W_key
print(f"Shape of key vector: {key.shape}")

# Compute the value vector for inputs
value = inputs @ W_value
print(f"Shape of value vector: {value.shape}")
```

Output:

```
Shape of query vector: torch.Size([4, 64])
Shape of key vector: torch.Size([4, 64])
Shape of value vector: torch.Size([4, 64])
```

Weight parameters vs. attention scores

Important Distinction

In the weight matrices W, the term `weight` is short for `weight parameters`, the values of a neural network that are optimized during training. This is not to be confused with the attention scores. As we already saw, attention scores determine the extent to which a context vector depends on the different parts of the input (i.e., to what extent the network focuses on different parts of the input). In short, weight parameters are the fundamental, learned coefficients that define the network's connections, while attention scores are dynamic, context-specific values.

Computing Attention Scores

Earlier, we computed the attention scores by taking the dot product among the input vectors. Now, we will compute the attention scores by taking the dot product between the query and key vectors. We then need to normalize the attention scores by using the softmax function.

Code:

```
# (d_in, d_out) @ (d_out, d_in) -> (d_in, d_in)
attention_scores = query @ key.T
normalized_attention_scores = torch.softmax(attention_scores, dim=-1)
print(f"Shape of normalized attention scores: {normalized_attention_scores.shape}")
```

Output:

```
Shape of normalized attention scores: torch.Size([4, 4])
```

The authors suggest scaling the attention scores by the square root of the embedding dimension of the key vectors before applying the softmax function. This is done to prevent the attention scores from becoming too large or too small, which can cause numerical instability during training. The scaling by the square root of the embedding dimension is the reason why this self-attention mechanism is also called `scaled dot-product attention`.

Code:

```
scaled_and_normalized_attention_scores = torch.softmax(attention_scores /
↪  torch.sqrt(torch.tensor(d_out)), dim=-1)
print(f"Shape of scaled and normalized attention scores:
↪  {scaled_and_normalized_attention_scores.shape}")
print(f"Sum of scaled and normalized attention scores:
↪  {torch.sum(scaled_and_normalized_attention_scores, dim=-1)}")
```

Output:

```
Shape of scaled and normalized attention scores: torch.Size([4, 4])
Sum of scaled and normalized attention scores: tensor([1., 1., 1., 1.])
```

The Scaling Secret: Why Divide by $\sqrt{d_{model}}$

Understanding Attention Scaling

As the embedding dimension grows, dot products grow with it, which pushes softmax toward extreme values. A concrete example:
Suppose we have two words represented by vectors of dimension d:

- With $d = 2 : [0.1, 0.1] \ @ \ [0.1, 0.1] = 0.02$
- With $d = 100 : [0.1, 0.1, ..., 0.1] \ @ \ [0.1, 0.1, ..., 0.1] = 1.0$
- With $d = 1000 : [0.1, 0.1, ..., 0.1] \ @ \ [0.1, 0.1, ..., 0.1] = 10.0$

As dimension grows, dot products explode. When these large numbers go through softmax, the distribution collapses to a near-one-hot vector and gradients vanish. The fix: divide by $\sqrt{d_{model}}$.
By dividing by $\sqrt{d_{model}}$, we:

- Keep dot products in a stable range regardless of dimension
- Ensure softmax produces meaningful probabilities
- Maintain healthy gradients during training

In modern language models (where $d_{model} > 1000$):

- Without scaling: Attention collapses to a hard "winner-takes-all" switch
- With scaling: Attention remains a smooth, learnable weighting

Code:

```
import math
d_model = 512
q = torch.randn(4, d_model)
k = torch.randn(4, d_model)
scores = q @ k.T

print("Without scaling:")
print(f"Score range: [{scores.min():.2f}, {scores.max():.2f}]")
print("\nWith scaling:")
scaled_scores = scores / math.sqrt(d_model)
print(f"Score range: [{scaled_scores.min():.2f}, {scaled_scores.max():.2f}]")
```

Output:

```
Without scaling:
Score range: [-69.67, 40.90]

With scaling:
Score range: [-3.08, 1.81]
```

Computing the context vector

Now, we will compute the context vector by taking the weighted sum of the value vectors, using the scaled and normalized attention scores.

Code:

```
context_vector = scaled_and_normalized_attention_scores @ value
print(f"Shape of context vector: {context_vector.shape}")
```

Output:

```
Shape of context vector: torch.Size([4, 64])
```

Calculation Summary

Table 1.7 shows the step-by-step shape changes for intermediate steps in attention with trainable weights.

Table 1.7: Summary Table for Calculation of Self-attention with Trainable Weights

Sr.	Step	Resulting Shape
1	Input	$(seq\_len, d_{model}) \equiv (seq\_len, d_{in})$
2	Initialize weight matrices for Query (Q), Key (K) and Value (V) i.e, W_q, W_k, W_v respectively	(d_{in}, d_{out})
3	Compute Query vector: $Q = input \ @ \ W_q \rightarrow (seq\_len, d_{in}) \ @ \ (d_{in}, d_{out})$	(seq_len, d_{out})
4	Compute Key vector: $K = input \ @ \ W_k \rightarrow (seq\_len, d_{in}) \ @ \ (d_{in}, d_{out})$	(seq_len, d_{out})
5	Compute Value vector: $V = input \ @ \ W_v \rightarrow (seq\_len, d_{in}) \ @ \ (d_{in}, d_{out})$	(seq_len, d_{out})
6	Compute attention score, scale and normalize: $attn\_scores = Q \ @ \ K^T \rightarrow (seq\_len, d_{out}) \ @ \ (d_{out}, seq\_len)$	(seq_len, seq_len)
7	Compute context vector: $context\_vec = attn\_scores \ @ \ V \rightarrow (seq\_len, seq\_len) \ @ \ (seq\_len, d_{out})$	(seq_len, d_{out})

1.3.3 Causal Attention

Imagine you're reading a mystery novel. As you read each word, you can only use the information from the words you've already read to make predictions about what might happen next. You can't peek ahead! This is exactly how causal attention works in Large Language Models. For many LLM tasks, we want the self-attention mechanism to consider only the previous tokens when predicting the next token in a sequence. This is where `causal attention` (also known as `masked attention`) comes in - it's a specialized form of self-attention that ensures the model only looks at past and present inputs when processing any given token. Think of it as giving your model good reading habits - no peeking ahead!

Looking at the transformer architecture in Figure Fig 1.2, we use two different types of attention:

- The encoder block uses standard self-attention since it needs to process the entire input sequence at once
- The decoder block uses causal attention to prevent any "future" information from leaking into its predictions

Let's explore how we can modify the standard self-attention mechanism to create this causal attention behavior that's essential for LLMs.

Masking the Attention Scores

The key to implementing causal attention is creating a clever mask that prevents the model from accessing future tokens. We do this by setting attention scores for future tokens to negative infinity ($-\infty$). When these scores go through the softmax function (which converts inputs into probabilities), they effectively become zero - making those future tokens invisible to the current position.

Fig 1.12 illustrates this transformation. We can implement this masking efficiently by creating a matrix with 1's above the diagonal and replacing these values with negative infinity ($-\infty$).

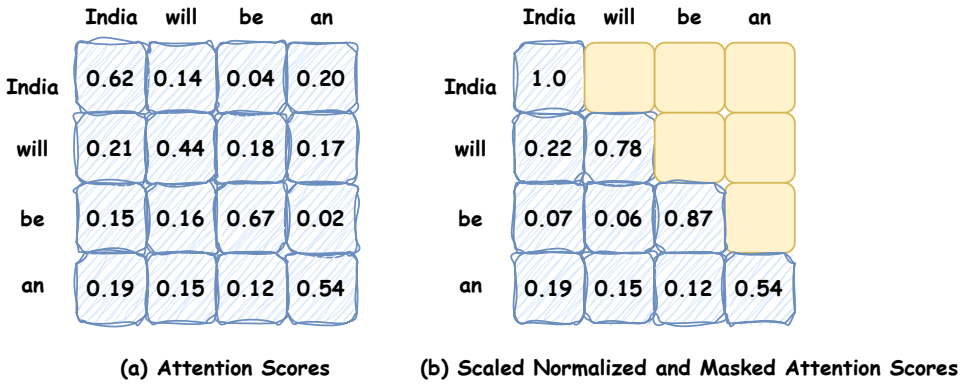


Figure 1.12: Comparison of unmasked and masked attention scores.

Code:

```
# triu -> triangular upper
# diag -> 0 : Diagonal is retained, 1: diagonal is made zero
mask = torch.triu(torch.ones(attention_scores.shape[0], attention_scores.shape[0]),
↪  diagonal=1)
print(mask)
```

Output:

```
tensor([[0., 1., 1., 1.],
        [0., 0., 1., 1.],
        [0., 0., 0., 1.],
        [0., 0., 0., 0.]])
```

Code:

```
# 0 -> False, 1 -> True
# Replace true values with -inf
masked_attention_scores = attention_scores.masked_fill(mask.bool(), -torch.inf)
scaled_and_normalized_masked_attention_scores =
↪  torch.softmax(masked_attention_scores / torch.sqrt(torch.tensor(d_out)),
↪  dim=-1)
print(f"Shape of masked attention scores: {masked_attention_scores.shape}")
print(f"Shape of scaled and normalized masked attention scores:
↪  {scaled_and_normalized_masked_attention_scores.shape}")
```

Output:

```
Shape of masked attention scores: torch.Size([4, 4])
Shape of scaled and normalized masked attention scores: torch.Size([4, 4])
```

Dropout: A Simple Trick to Prevent Overfitting

Think of dropout as randomly turning off some neurons during training, similar to how studying different subjects with varying combinations of friends helps you understand the material better, rather than always relying on the same study group. This technique prevents the model from becoming too dependent on specific patterns and helps it learn more generalisable features.

In transformer architectures, we typically apply dropout at two key points in the attention mechanism:

- After calculating the attention weights
- After applying the attention weights to the value vectors

In our implementation, we'll focus on the first approach - applying dropout after computing the attention weights, as it's more commonly used in practice.

Code:

```
# In the following code example, we use a dropout rate of 50%
# which means masking out half of the attention weights.
# We apply PyTorch's dropout implementation first to a 6 × 6 tensor consisting of
↪  1s for simplicity:

torch.manual_seed(123)
dropout = torch.nn.Dropout(0.5)
example = torch.ones(6, 6)
print(f"Shape of example: {example.shape}")
print(f"Example before dropout: {example}")
print(f"Example after dropout: {dropout(example)}")
```

Output:

```
Shape of example: torch.Size([6, 6])
Example before dropout: tensor([
        [1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.],
        [1., 1., 1., 1., 1., 1.]])

Example after dropout: tensor([
        [2., 2., 2., 2., 2., 2.],
        [0., 2., 0., 0., 0., 0.],
        [0., 0., 2., 0., 2., 0.],
        [2., 2., 0., 0., 0., 2.],
        [2., 0., 0., 0., 0., 2.],
        [0., 2., 0., 0., 0., 0.]])
```

`Important Note:` When we apply dropout with a 50% rate, half of the attention weights are randomly set to zero. To maintain the overall influence of the attention mechanism, the remaining weights are automatically scaled up by a factor of $1/(1-0.5) = 2$. This scaling is handled automatically by PyTorch's `torch.nn.Dropout` implementation, ensuring consistent behavior during both training and inference phases.

Combining Attention into a Compact Class

Let's now combine all the steps we have seen so far into a compact class. While we are at it, also make the class more flexible by making it capable of handling batch of inputs rather than just a single input. This is because in practice, we will be dealing with batches of inputs, and we want our class to be able to handle this.

Code:

```
class Attention(nn.Module):
    """
    A self-attention mechanism that computes attention scores and context vectors
    for a given input tensor. This class is designed to handle batches of inputs
    and incorporates dropout for regularization.

    Attributes:
        d_in (int): The dimensionality of the input features.
        d_out (int): The dimensionality of the output features.
        W_query (nn.Linear): Linear transformation for the query vectors.
        W_key (nn.Linear): Linear transformation for the key vectors.
        W_value (nn.Linear): Linear transformation for the value vectors.
        dropout (nn.Dropout): Dropout layer to prevent overfitting.

    Args:
        d_in (int): Input feature dimension.
        d_out (int): Output feature dimension.
        dropout_rate (float): The dropout rate to apply to the attention weights.
        qkv_bias (bool, optional): Whether to include a bias term in the linear
        ↪  layers.
                                   Defaults to False.
    """

    def __init__(self, d_in: int, d_out: int, dropout_rate: float, qkv_bias: bool =
    ↪  False):
        super().__init__()
        self.d_in = d_in
        self.d_out = d_out

        # utilizing PyTorch's nn.Linear layers, which effectively perform matrix
        ↪  multiplication when the bias units are disabled.
        # Additionally, a significant advantage of using nn.Linear instead of
        ↪  manually implementing nn.Parameter(torch.rand(...))
        # is that nn.Linear has an optimized weight initialization scheme,
        ↪  contributing to more stable and effective model training.
        self.W_query = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_key = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.W_value = nn.Linear(d_in, d_out, bias=qkv_bias)
        self.dropout = nn.Dropout(dropout_rate)

    @staticmethod
    def calculate_attention_scores(queries: torch.Tensor, keys: torch.Tensor, mask:
    ↪  torch.Tensor = None,
                                   dropout: nn.Dropout = None) -> torch.Tensor:
        """
        Calculate attention scores and weights for given queries and keys.

        Args:
            queries (torch.Tensor): Query tensor of shape (batch_size, seq_len,
            ↪  d_out)
```

```
            keys (torch.Tensor): Key tensor of shape (batch_size, seq_len, d_out)
            mask (torch.Tensor, optional): Attention mask tensor. Defaults to None.
            dropout (nn.Dropout, optional): Dropout layer to apply. Defaults to
            ↪ None.

        Returns:
            torch.Tensor: Attention weights after softmax and dropout
        """
        # Compute the attention scores
        attention_scores = queries @ keys.transpose(1, 2)

        # Apply the mask if provided
        if mask is not None:
            attention_scores.masked_fill_(mask == 0, -torch.inf)

        # Compute the attention weights using softmax
        attention_weights = torch.softmax(attention_scores /
        ↪ np.sqrt(queries.shape[-1]), dim=-1)

        # Apply dropout if provided
        if dropout is not None:
            attention_weights = dropout(attention_weights)

        return attention_weights

    def forward(self, query: torch.Tensor, key: torch.Tensor, value: torch.Tensor,
                mask: torch.Tensor = None) -> torch.Tensor:
        """
        Forward pass for the attention mechanism.

        Args:
            query (torch.Tensor): Input tensor of shape (batch_size, seq_len,
            ↪ d_in).
            key (torch.Tensor): Input tensor of shape (batch_size, seq_len, d_in).
            value (torch.Tensor): Input tensor of shape (batch_size, seq_len,
            ↪ d_in).
            mask (torch.Tensor, optional): Attention mask tensor. Defaults to None.

        Returns:
            torch.Tensor: The context vector computed from the attention weights
                          and the value vectors, of shape (batch_size, seq_len,
                          ↪ d_out).
        """
        # Generate keys, queries and values for the input x via matrix
        ↪ multiplication
        keys = self.W_key(key)
        queries = self.W_query(query)
        values = self.W_value(value)

        # Calculate attention weights using the static method
        attention_weights = Attention.calculate_attention_scores(queries, keys,
        ↪ mask, self.dropout)

        # Compute the context vector as a weighted sum of the values
        context_vec = attention_weights @ values
        return context_vec
```

The chapter continues with Multi-Head Attention, Layer Normalisation, Residual Connections, the complete Encoder and Decoder assembly, and a working English→Hindi translation model trained end-to-end.

And this is just Chapter 1.

The book goes on to build GPT-2, Llama 3, and DeepSeek — from scratch.

Get the full book:

https://leanpub.com/adventures-with-llms

Code repository:

https://github.com/S1LV3RJ1NX/mal-code