

Self-attention as a directed graph!

Self-attention is at the heart of transformers, the architecture that led to the LLM revolution that we see today.

In this post, I'll clearly explain self-attention & how it can be thought of as a directed graph.

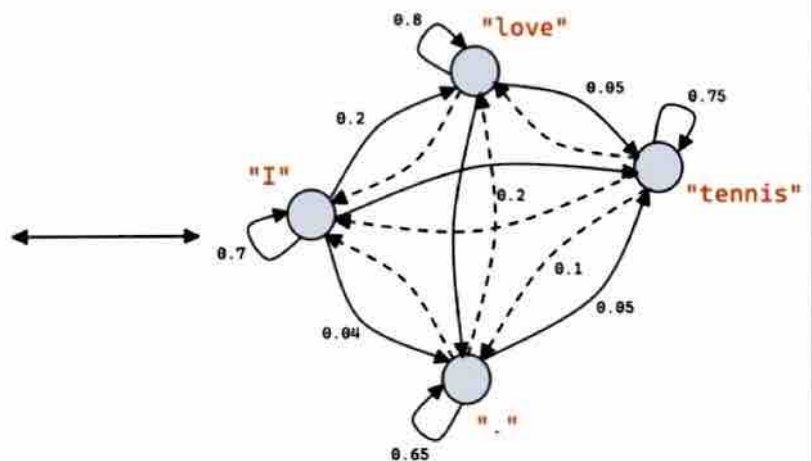
Read more... 🖱️

Attention: A communication mechanism

Attention probability scores:
how much a token should pay attention
to itself & the neighboring token

	"I"	"love"	"tennis"	"."
"I"	0.7	0.2	0.06	0.04
"love"	0.1	0.8	0.05	0.05
"tennis"	0.05	0.1	0.75	0.1
"."	0.1	0.2	0.05	0.65

Visualizing attention as a directed graph




Wondering where these numbers come from!? 🤔
Continue reading ... 📖

Before we start a quick primer on tokenization!

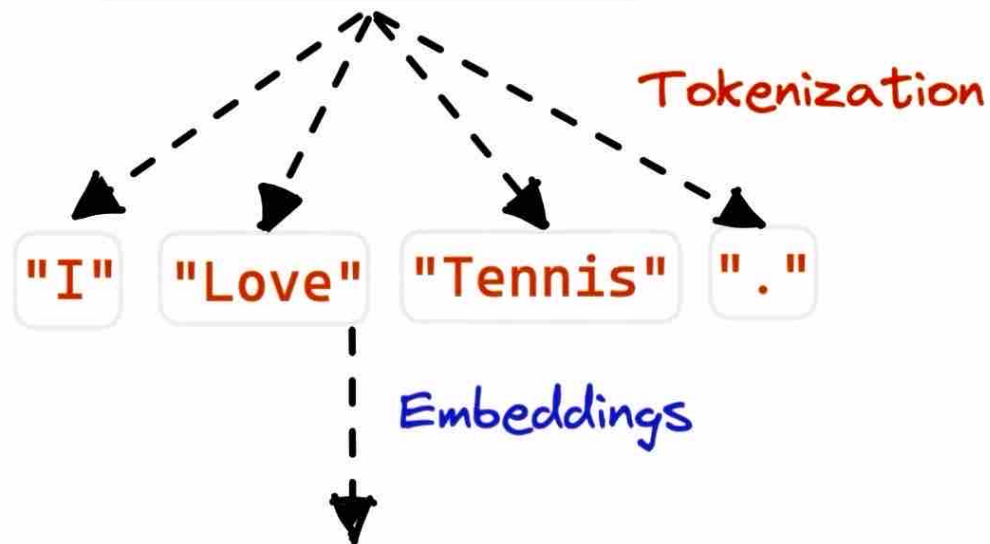
Raw text → Tokenization → Embedding
→ Model

Embedding is a meaningful representation of each token (roughly a word) using a bunch of numbers.

This embedding is what we provide as an input to our language models.

Check this 

I love Tennis.



"I"

[0.89, 0.45, 0.67, ..., 0.32, 0.04]

"Love"

[0.59, 0.35, 0.75, ..., 0.12, 0.24]

"Tennis"

[0.99, 0.48, 0.27, ..., 0.52, 0.18]

"."

[0.16, 0.55, 0.97, ..., 0.79, 0.84]



@akshay_pachaar

The core idea of Language modelling is to understand the structure and patterns within language.

By modeling the relationships between words (tokens) in a sentence, we can capture the context and meaning of the text.



I love Tennis & I am a big fan of Rafael Nadal.

A language model must see the entire context,
It should be aware of the relative positions and
relationships among the tokens.

Let's see how it's done 📺



Now self attention is a communication mechanism that help establish these relationships, expressed as probability scores.

Each token assigns the highest score to itself and additional scores to other tokens based on their relevance.

You can think of it as a directed graph

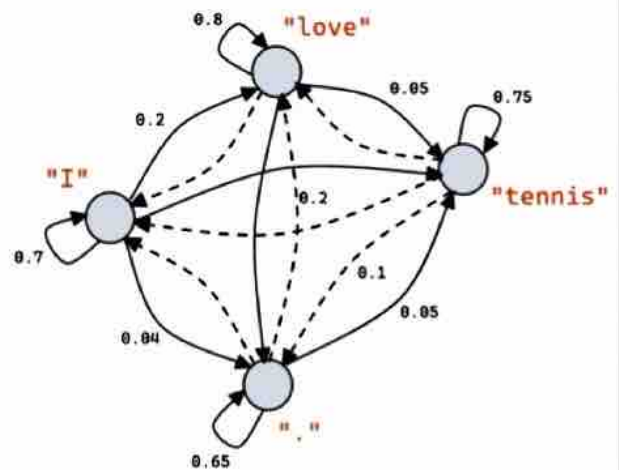


Attention: A communication mechanism

Attention probability scores:
how much a token should pay attention
to itself & the neighboring token

	"I"	"love"	"tennis"	","
"I"	0.7	0.2	0.06	0.04
"love"	0.1	0.8	0.05	0.05
"tennis"	0.05	0.1	0.75	0.1
","	0.1	0.2	0.05	0.65

Visualizing attention as a directed graph



Wondering where these numbers come from!? 😊
Continue reading ... 📖

To understand how these probability/attention scores are obtained:

We must understand 3 key terms:

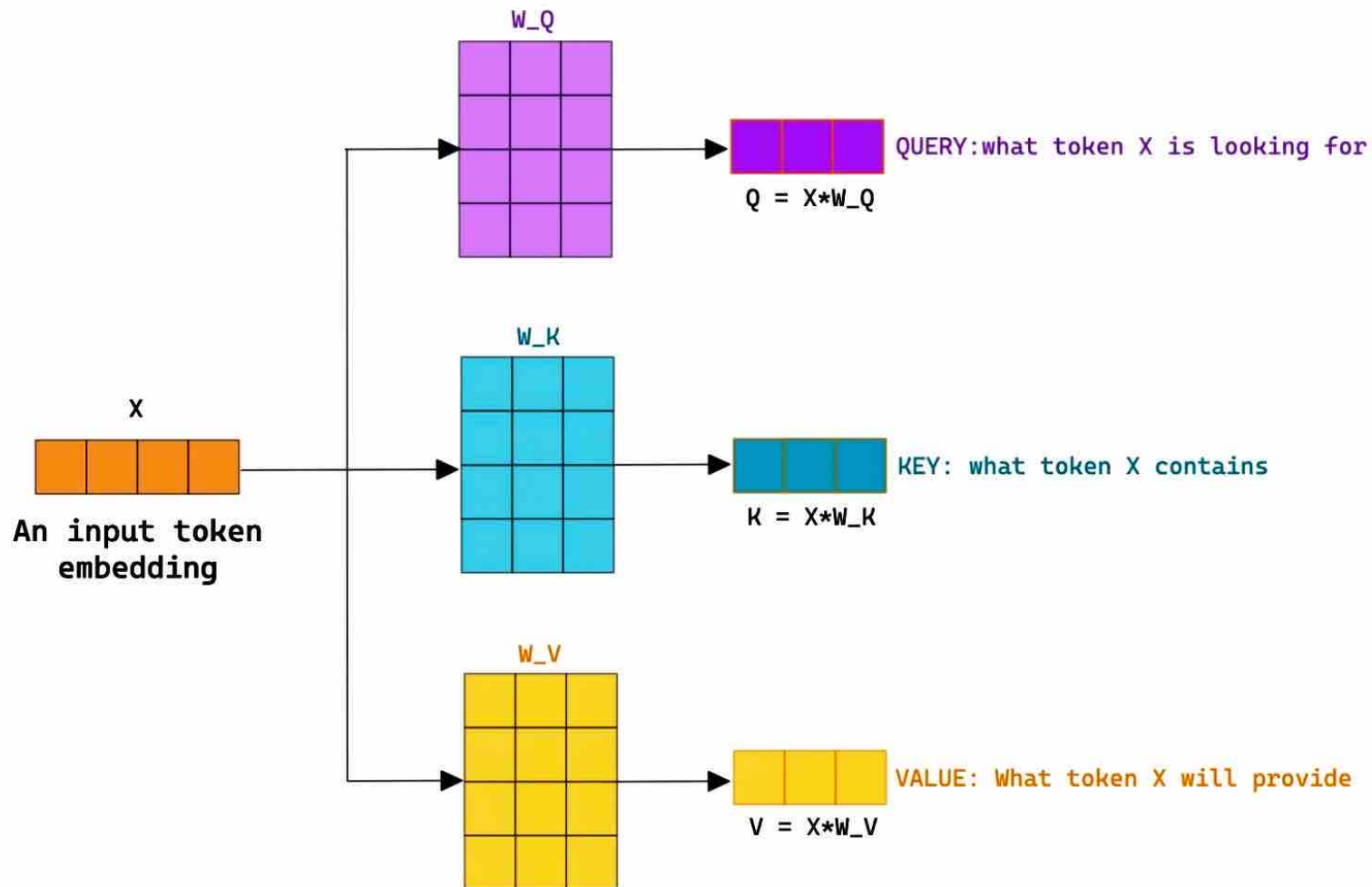
- Query Vector
- Key Vector
- Value Vector

These vectors are created by multiplying the input embedding by three weight matrices that are trainable.

Check this out 

Understanding Keys, Queries & Values

W_Q , W_K & W_V are Trainable weight matrices.

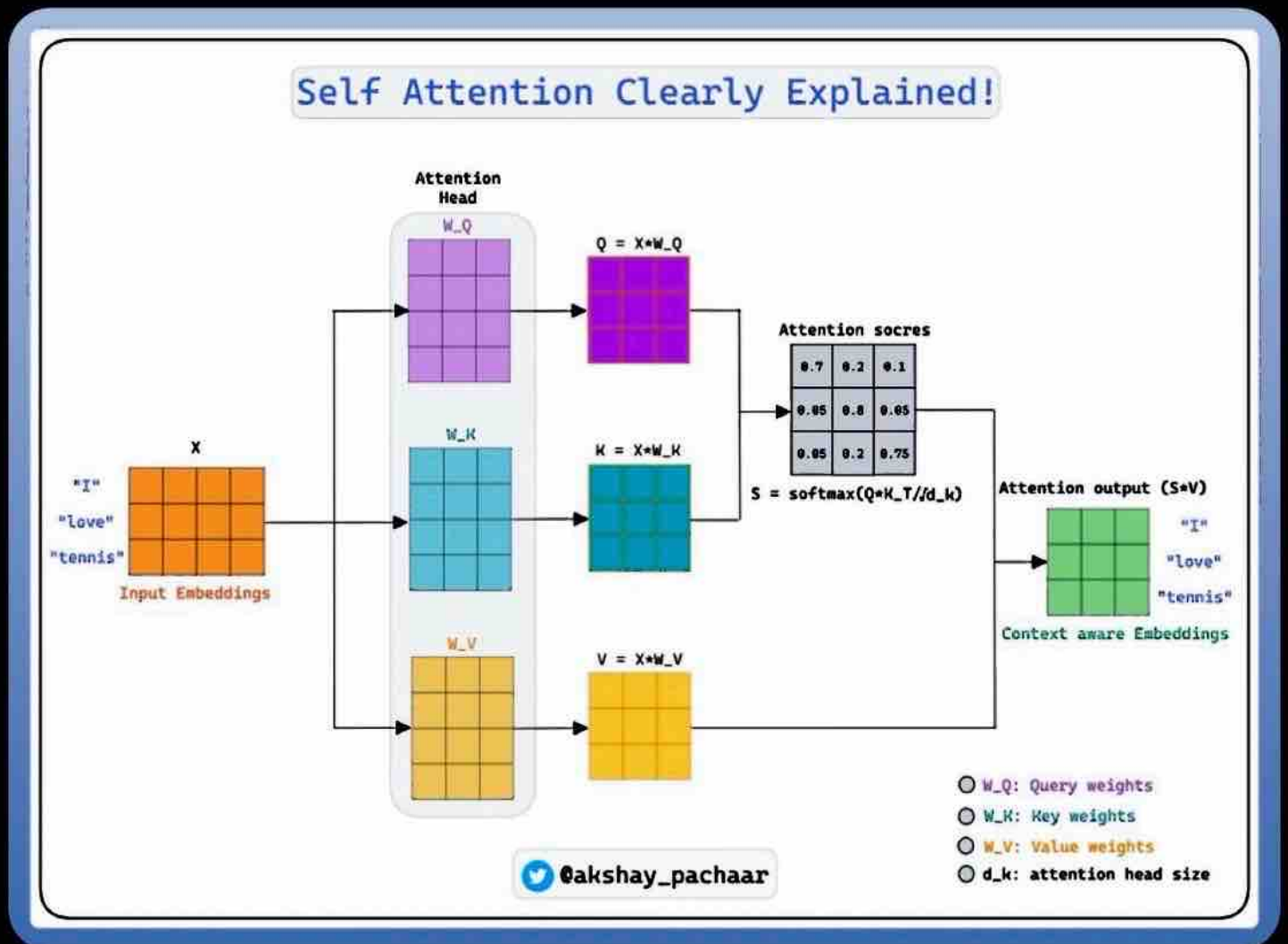


@akshay_pachaar

Now here's a broader picture of how input embeddings are combined with Keys, Queries & Values to obtain the actual attention scores.

After acquiring keys, queries, and values, we merge them to create a new set of context-aware embeddings.

Check this out 🙌



Implementing self-attention using PyTorch, doesn't get easier! 🚀

It's very intuitive! 💡

Check this out 🙌

```
self_attention.py

import torch
import torch.nn as nn
from torch.nn import functional as F

class SelfAttention(nn.Module):
    """ Single head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)
        # compute attention scores
        wei = q @ k.transpose(-2, -1) * k.shape[-1]**-0.5 (divide by root of d_k)
        wei = F.softmax(wei, dim=-1)
        v = self.value(x)
        out = wei @ v
        return out
```



Akshay 🚀

🐦 @akshay_pachar