Transformer Architecture

My goal was simple: understand the Transformer block by building a mini model from scratch with small numbers (sequence length = 4, embedding dimension = 8, heads = 2). I wanted to see the core mechanism: the shapes, the attention probabilities, and how the block produces a new representation for each token.
What I implemented and what each part means:

Token embeddings or input vector:

In my notebook, I started with random token embeddings, in real models, embeddings come from a learned embedding table, and similar words can end up with similar vectors after training.

Q, K, V projections:
Then I created W_Q, W_K, W_V matrices and computed:

- Q = X @ W_Q
- K = X @ W_K
- V = X @ W_V

The correct mindset for me was:

- Q (query) = what this token is looking for
- K (key) = what this token offers / how it should be matched
- V (value) = what content will be mixed into the output

Important learning from my output: they are just different linear projections of the same token vector, learned during training in real models.

Scaled dot product attention:

scores = (Q @ K.T) / sqrt(d_k)
And weights:
weights = softmax(scores)
Then attention output:
output = weights @ V
This step was the most important for me. Because after printing:

- scores shape (T,T)
- weights shape (T,T)
- row sums almost was 1
  I could see the mechanism is just:
  each token creates a distribution over all tokens, how much to attend, then it mixes values.

Multi head attention:

After I understood single head attention, I implemented multi head attention:

- I set num_heads = 2
- head_dimension = embedding_dimension // num_heads = 4
- Split Q, K, V into heads

Then I computed attention per head , two separate attention matrices , and finally:

- concatenated heads back to (T, d_model)
- applied output projection W_OutPut

What I saw here, from my own attention matrices:

- each head can produce different attention patterns
- so the model can focus on different relationships in parallel

Causal mask:

Then I applied a causal mask to block attention to future tokens:

- positions above the diagonal set to -inf
- after softmax, probability becomes 0 for future tokens

This is a key difference between:

- bidirectional self-attention (encoder, BERT-style)
- causal self-attention (decoder, GPT)

In decoder language modeling, masking is needed because otherwise the model would see the answer token. The original transformer uses masked self-attention in the decoder part.

Residual + Layer normalization

After attention output projection, I did:
x1 = layer_normalization(token_embeddings_multi_heads + Output_projection)
This is the classic Transformer block idea:

- residual connection: keep original info and add the new transformation
- layer normalization: stabilize the hidden states

Feed-forward network:
Then I built the FFN:

- latent_dimension = 16
- Hidden = relu(x1 @ W1 + b1)
- OutPut = Hidden @ W2 + b2
- then another residual + layer norm:

x2 = layer_normalization(x1 + OutPut)
The best way I understood this:
Self-attention is mainly mixing information across tokens.
FFN is applied per token, to add non linearity and transform features.

Output layer (logits - probabilities - prediction):
Finally I did the output projection:

- Weight_vocab shape = (d_model, vocab_size)
- Logits = x2 @ Weight_vocab + b_vocab = shape (T, vocab_size)
- softmax across vocab = probabilities
- argmax = predicted token id

This matches the idea of next-token prediction: the last position is used to predict the next token. In my setup vocab_size=20.

The most important line in my notebook conceptually is:

Attention_output = attention_weights @ V

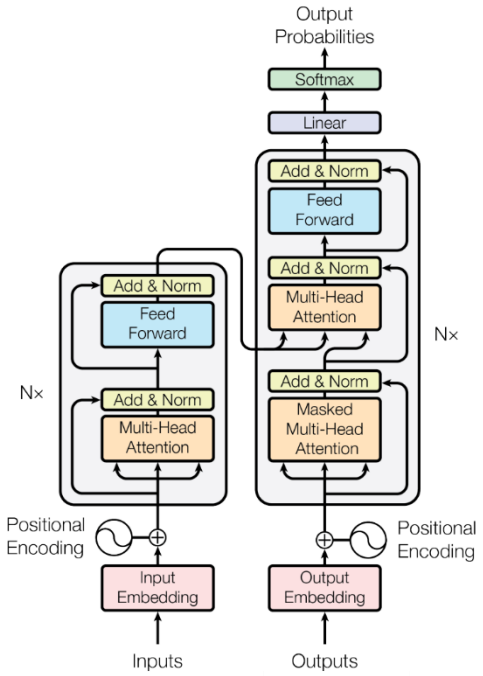
This is where the model builds a new representation for each token by mixing information from other tokens:

- attention_weights[i, j] decides how much token *i* will take from token *j*
- V[j] is what token *j* contributes its content
- the result is a weighted sum

In a real language model, this is how ambiguity gets resolved: bank attends differently depending on whether surrounding context emphasizes account or river. The specific numbers in my output are not meaningful, they are random weights, but the meaning of the operation is exactly what real Transformers do.

BERT

Encoder



GPT

Decoder

Source: How Transformers solve tasks

When I look back at the diagram now, it makes more sense because I can mentally translate each box into operations I coded:

- Multi-Head Attention = split Q/K/V into heads, score matrices, softmax, weighted sum, merging, output projection
- Residual + Layer normalization, I literally saw mean, std normalize
- Feed Forward = 2 layer applied per token.
- Masked Multi-Head Attention (decoder) = same attention, but with causal mask
- Encoder–Decoder Attention = same formula again, but Q comes from decoder,