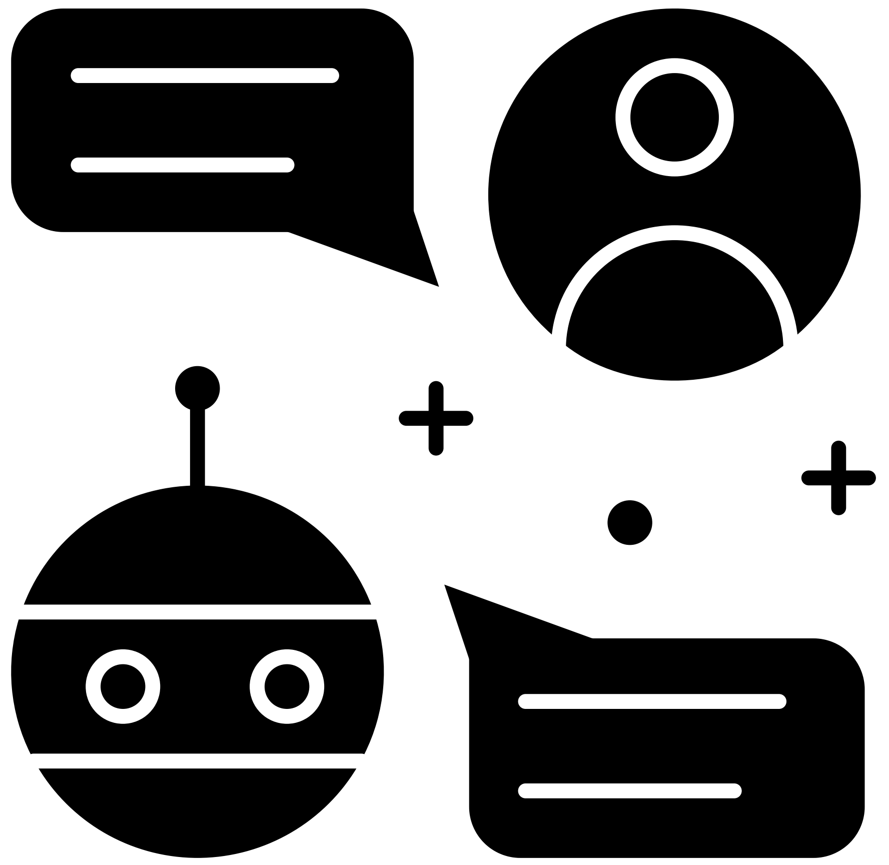
Course: Build a Small Language Model from scratch

Module 3: Transformer Architecture



Instructor: Nusrat Jahan Lia

Learning Objectives

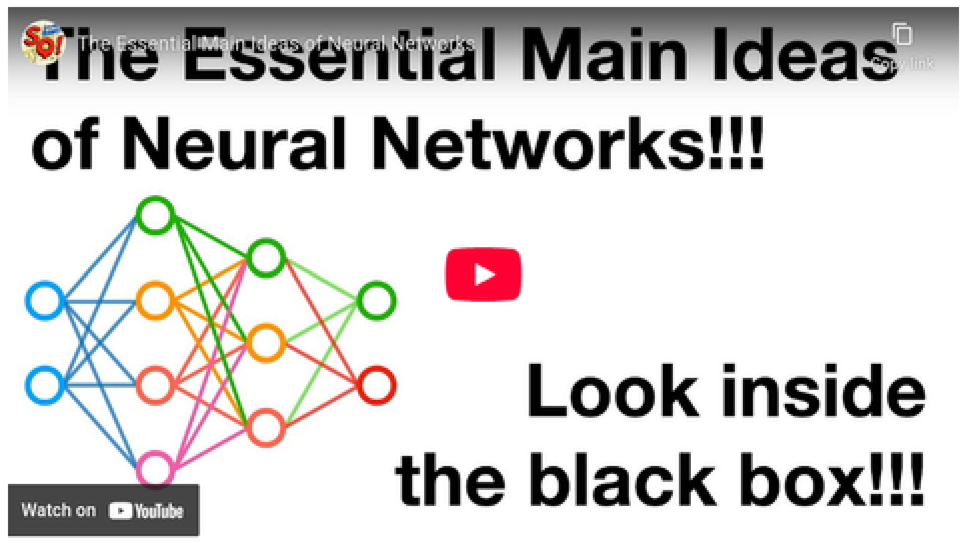
By the end of this module, you will:

- Understand why Transformers revolutionized language modeling
- Implement token and positional embeddings from scratch
- Build a complete self-attention mechanism with causal masking
- Construct transformer blocks with residual connections
- Assemble a complete GPT model architecture

Prerequisites Review

Before diving into Transformers, you need to understand Neural Network, Recurrent Neural Network (RNN), Long-short Term Memory (LSTM)

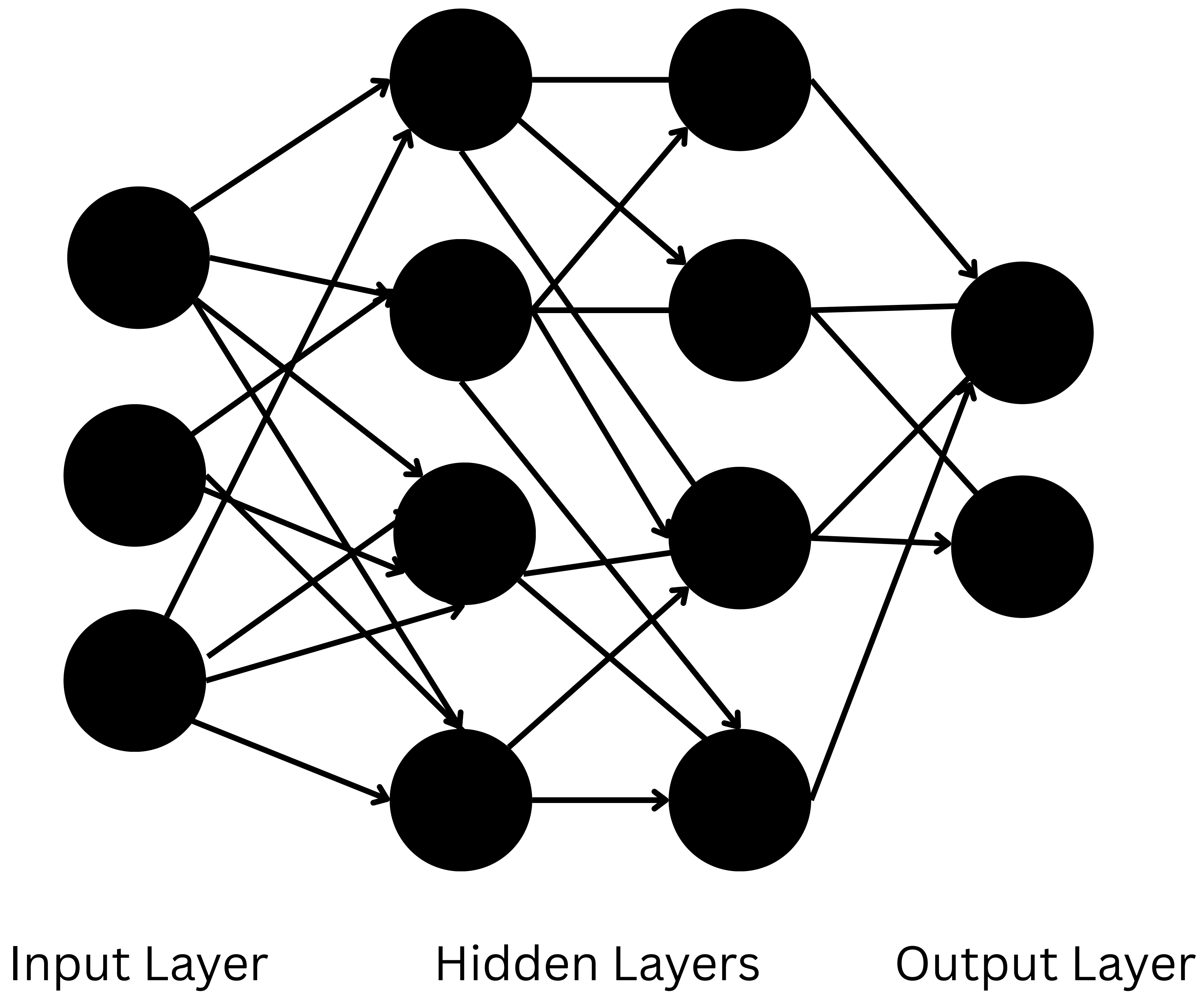
I didn't dive deep into traditional deep learning topics like NN, RNNs or LSTMs in this course because our main focus is on transformers. That said, understanding neural networks is super important before jumping into transformers — so I've linked some foundational NN content that you should definitely review first!


The Essential Main Ideas of Neural Networks
The Essential Main Ideas of Neural Networks!!!
Look inside the black box!!!
Watch on YouTube

In short, What is a Neural Network?

A neural network is a computational model, which has:

- **Neurons:** Processing units that receive inputs and produce outputs
- **Weights:** Parameters that determine the strength of connections
- **Layers:** Groups of neurons that process information sequentially



The Transformer Revolution (2017)

"Attention Is All You Need" - The Paper That Changed Everything

Key Insight: Instead of processing words sequentially, let each word directly "attend" to all other words simultaneously.

Core Transformer Principles

1. Parallel Processing:

```
Traditional RNN: word₁ → word₂ → word₃ → word₄
Transformer:     word₁ ↕ word₂ ↕ word₃ ↕ word₄
                         ↖  ↗  ↖  ↗  ↖  ↗
               All words interact simultaneously
```

2. Self-Attention Mechanism

- Each word can **"look at"** and **"attend to"** every other word
- Weights determine how much attention to pay to each word

3. Positional Encoding

- Since we process all words simultaneously, we need to tell the model about word order

Mathematical Foundation of Attention

Attention Formula (we'll implement this later):

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$$

Where:
Q = Queries (what each word is looking for)
K = Keys (what information each word provides)
V = Values (the actual information content)

Attention Mechanism Introduction

When you read "**রহিম একজন ভাল ছাত্র**, **সে পড়াশোনায় মনোযোগী**", your brain automatically connects "**সে**" back to "**রহিম**".

This is attention!

GPT Architecture Overview

What Are Model Parameters?

Parameters are the learnable weights and biases in a neural network that get updated during training. Think of them as the "knobs" the model adjusts to learn patterns in data.

- **Weight matrices:** Store learned relationships between inputs and outputs
- **Bias vectors:** Allow the model to shift outputs (though often omitted in modern transformers)
- **Embedding tables:** Convert discrete tokens into continuous vectors

Imagine a massive equalizer with millions of sliders. Each slider (parameter) can be adjusted to fine-tune how the system processes information. **More sliders = more control, but also more complexity**.

Our Model Configuration:

```
class GPTConfig:
    def __init__(self):
        self.vocab_size = meta["vocab_size"]  # ~32,100 for BanglaT5
        self.block_size = 128                 # Context length
        self.n_layer = 8                      # Number of transformer blocks
        self.n_head = 8                       # Number of attention heads
        self.n_embd = 512                     # Embedding dimension
        self.dropout = 0.2                    # Regularization
```

vocabulary size refers to the number of unique tokens (words, subwords, or characters) that the model can understand and generate. It's a crucial parameter that impacts both the model's ability to handle diverse language and its computational efficiency during inference.

Token Embeddings

Purpose: Convert discrete tokens into continuous vectors

1. Token: '▁আমি' → [3.8904e-02, -5.4560e-03, 1.5565e-01, -2.2271e-01, 2.1094e-01,
, ..., ..., ..., ..., ..., ..., ..., ..., ...,
-1.7417e-02, 2.8590e-01, 2.7366e-01, 7.4568e-02, -2.2345e-01] **(512 dimensions)**

This allows the model to understand the context and meaning of the tokens based on their vector representations

so here, parameters: vocab_size × n_embd = 32,100 × 512

```python
from transformers import AutoTokenizer, T5Model
import torch

# Load tokenizer and encoder-decoder model
tokenizer = AutoTokenizer.from_pretrained("csebuetnlp/banglat5")
model = T5Model.from_pretrained("csebuetnlp/banglat5")

# Input text
text = "আমি বাংলায় কথা বলি"

# Tokenize
inputs = tokenizer(text, return_tensors="pt")

# Pass through encoder to get token embeddings
with torch.no_grad():
    encoder_outputs = model.encoder(**inputs)

# Extract token-level vectors
token_embeddings = encoder_outputs.last_hidden_state.squeeze(0)  # Shape: (seq_len, 512)
tokens = tokenizer.convert_ids_to_tokens(inputs["input_ids"].squeeze())

# Show first 10 dims of each token's embedding
for i, (token, vector) in enumerate(zip(tokens, token_embeddings)):
    print(f"{i+1}. Token: '{token}' → {vector}")
```

Why 512 Dimensions?

```
def analyze_embedding_dimensions():
    """Understand the trade-off between embedding size and model capacity"""

    vocab_size = 32000
    embedding_dims = [128, 256, 512, 1024]

    print("Embedding Dimension Analysis:")
    print("-" * 50)

    for dim in embedding_dims:
        params = vocab_size * dim
        print(f"Embedding dim: {dim}")
        print(f"  Parameters: {params:,} ({params/1e6:.2f}M)")
        print(f"  Memory (float32): {params*4/1024/1024:.2f} MB")
        print(f"  Expressiveness: {'Low' if dim < 256 else 'Medium' if dim < 512 else 'High'}")
        print()

analyze_embedding_dimensions()
```

```
Embedding Dimension Analysis:
--------------------------------------------------
Embedding dim: 128
  Parameters: 4,108,800 (4.11M)
  Memory (float32): 15.67 MB
  Expressiveness: Low  // Limited ability to capture subtle differences in words.
                       //May work for simple or small-scale tasks (e.g., text classification).


Embedding dim: 256
  Parameters: 8,217,600 (8.22M)
  Memory (float32): 31.35 MB
  Expressiveness: Medium  // Balanced. Can represent most key features of language.
                          //Suitable for mid-sized models or tasks like sentiment or intent detection.

Embedding dim: 512
  Parameters: 16,435,200 (16.44M)
  Memory (float32): 62.70 MB
  Expressiveness: High         // Rich representation. Better at capturing fine-grained syntax, semantics, morphology,
                               //and long-range dependencies. Ideal for complex tasks (translation, summarization, QA).


Embedding dim: 1024
  Parameters: 32,870,400 (32.87M)
  Memory (float32): 125.39 MB
  Expressiveness: High
```

Consider two words:

- **"পড়ছে" (reading)**
- **"পড়েছে" (has read)**

- **A low-dimensional embedding** might map both to a similar vector because it lacks space to encode tense.
- **A high-dimensional embedding** can preserve that distinction, helping the model understand tense, aspect, and context more accurately**.**

Token Embedding Matrix

Let

- V = vocabulary size
- d = embedding dimension

Then:

$$E_{token} \in R^{V \times d}$$

Each row $E_{token}[i]$ is the embedding for token ID i.

Given input token indices:

$$t=[i_0,i_1,i_2,\ldots,i_T] \text{ where each } i_t \in [0,V)$$

We fetch:

$$X = \begin{bmatrix} E_{token}[i_0] \\ E_{token}[i_1] \\ \vdots \\ E_{token}[i_T] \end{bmatrix} \in R^{T\times d}$$

Positional Embedding Matrix (Learned)

Let:

- T = max sequence length (e.g., 512 or 1024)
- d = embedding dimension

Then:

$$E_{pos} \in R^{T \times d}$$

Each row $E_{pos}[t]$ is a vector representing position t.

These are also trainable parameters, initialized randomly and learned over time.

Combined Embedding

At each position t, final input is:

$$z_t = x_t + p_t = E_{token}[i_t] + E_{pos}[t]$$

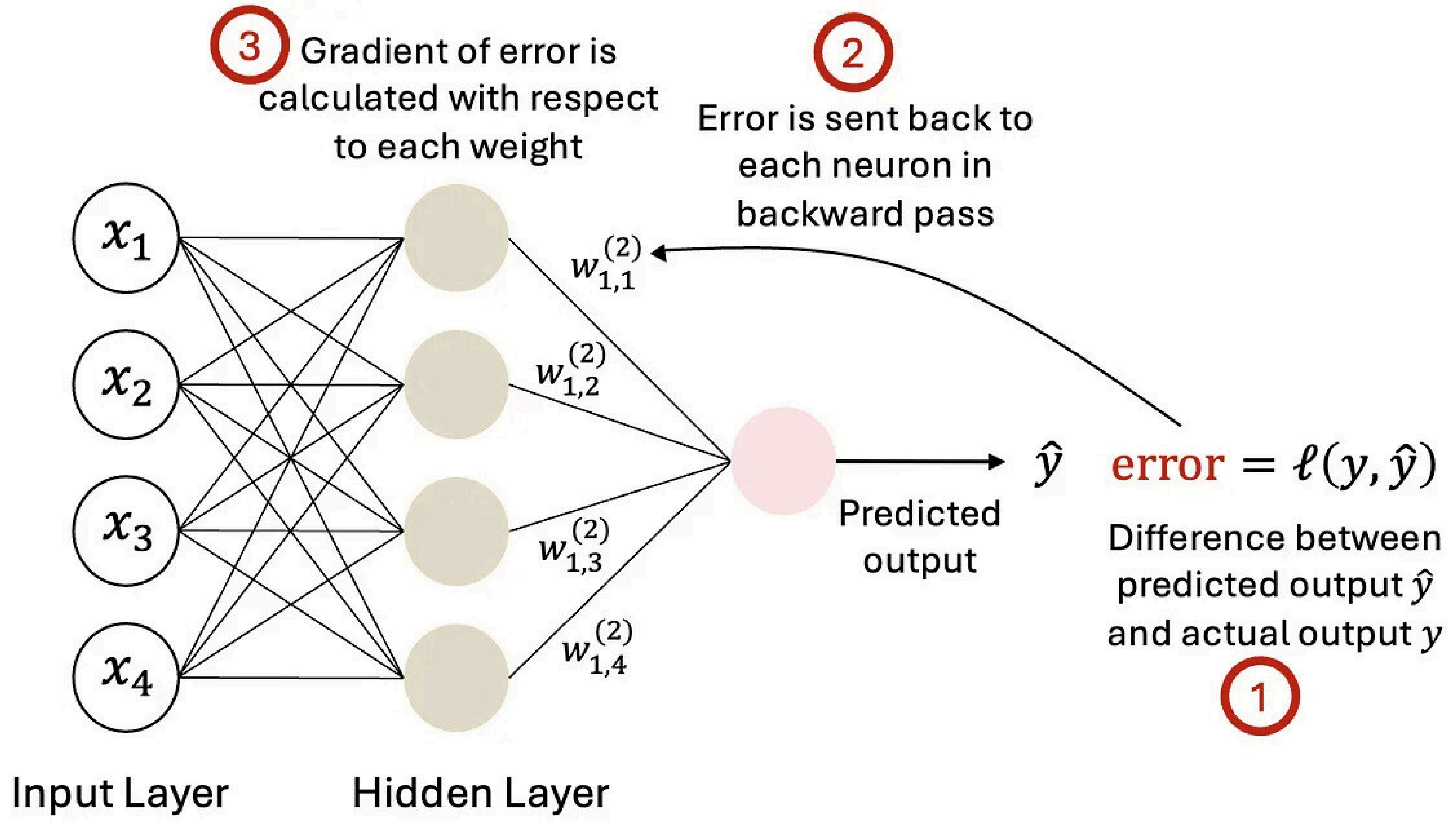
Where:

- $\mathbf{x_t}$ is the token embedding
- $\mathbf{p_t}$ is the positional embedding

Backpropagation (How Embeddings Are Learned)

During Training:

- Loss L (e.g., cross-entropy) is computed at the output.
- Gradients are propagated back through the transformer layers.



3 Gradient of error is calculated with respect to each weight
2 Error is sent back to each neuron in backward pass
x_1
x_2
x_3
x_4
$w_{1,1}^{(2)}$
$w_{1,2}^{(2)}$
$w_{1,3}^{(2)}$
$w_{1,4}^{(2)}$
$\hat{y}$
error $= \ell(y, \hat{y})$
Predicted output
Difference between predicted output $\hat{y}$ and actual output y
1
Input Layer
Hidden Layer

Take a look at

Eventually, we compute:

$$\frac{\partial L}{\partial E_{token}[i_t]}$$ for every token used in the batch.

These gradients update the relevant rows of E_{token} and E_{pos}

So only the rows corresponding to used tokens/positions get updated.....just like **sparse** updates.

Why Addition? Why Not Concatenation?

We combine token and positional embeddings using:

$$z_t = x_t + p_t$$

instead of:

$$[x_t; p_t] \in R^{2d}$$

Why?

- Keeps dimensionality manageable.
- Forces model to integrate position into semantic meaning.
- Addition is computationally simpler and works empirically well.

Dropout?

Randomly **"turns off"** neurons during training. That means some neurons are ignored (set to zero), forcing the network to generalize better to unseen data which prevents overfitting.

For Dropout (rate = 0.1)

- Each neuron has a 10% chance of being dropped

Dropout Rate	Effect
0	❌ No regularization — high overfitting risk
0.1	✅ Light regularization — good default choice
0.2	🔁 Medium regularization — for complex models
0.5	⚠️ Heavy regularization — risk of underfitting

Attention in Transformers: A Detective Solving a Case

Imagine an AI model is like a detective trying to solve a mystery ... say, "Who stole the missing gem?"

Query: "Who stole the gem?"

Our **context** is the detective's notebook : full of statements from different witnesses:

Witness	What they said
Witness A	“I saw someone in a red coat running away.”
Witness B	“The security alarm went off at midnight.”
Witness C	“I was asleep the whole time.”
Witness D	“Red coat? That matches the thief’s usual disguise.”

Now, **the detective (our model) needs** to weigh each clue in relation to the query — "Who stole the gem?"

The model gives attention scores to each clue:

Witness	Relevance to Query	Attention Weight	Why?
A	High	0.85	Saw someone escape!
B	Medium	0.5	Timing is useful.
C	Low	0.1	Not helpful — no info.
D	High	0.9	Matches known thief pattern!

These weights help the model combine the most relevant clues (statements) to figure out the answer.

Query, Key, Value Concept

- **Q:** Query — what I'm looking for
- **K:** Key — what each word offers
- **V:** Value — the information content

Let’s say the input sentence is:

X = ["রহিম", "গত", "সপ্তাহে", "ঢাকায়", "গিয়েছিল"]

Each word is first embedded as a vector in $\mathbb{R}$^d_model

We collect these into a matrix:

X $\in$ $\mathbb{R}$^(T × d_model), where T = number of tokens

Wait!

Do we remember what’s the final goal?

It is to predict what word comes after the previous one

"রহিম", "গত", "সপ্তাহে", "ঢাকায়", "গিয়েছিল"

$E_0 \quad E_1 \quad E_2 \quad E_3 \quad E_4$

$$\begin{tikzcd} & E_0 \arrow[dl] \arrow[d] \arrow[dr] & \\ Q & K & V \end{tikzcd}$$

$Q_i = E_i \times W_q$

$K_i = E_i \times W_k$

$V_i = E_i \times W_v$

matrix multiplication

Initialized randomly (like weights in a neural network layer) and updated during training using backpropagation, $W_q\ ,\ W_k\ ,\ W_v \in \mathbb{R}^{d\_model \times d\_k}$

d_k= d_model// n_head

Step	Meaning
Random Initialization	Starts with random numbers to break symmetry
Linear Transformation	Learns how to convert embeddings into query, key, and value roles
Backpropagation	Learns by adjusting weights to minimize task loss

Given:

Q, K, V all have shape: [tokens, d_k] (after projection),

$$\text{Attention}(Q,K,V) = \text{softmax}(QK^T / \sqrt{d\_k}) \times V$$

Q vs K → Similarity Scores

Each query vector is compared to every key vector using a dot product:

$$e\_ij \quad = \quad (Q_i K_j \,/\, \sqrt{}d\_k)$$

This tells us:

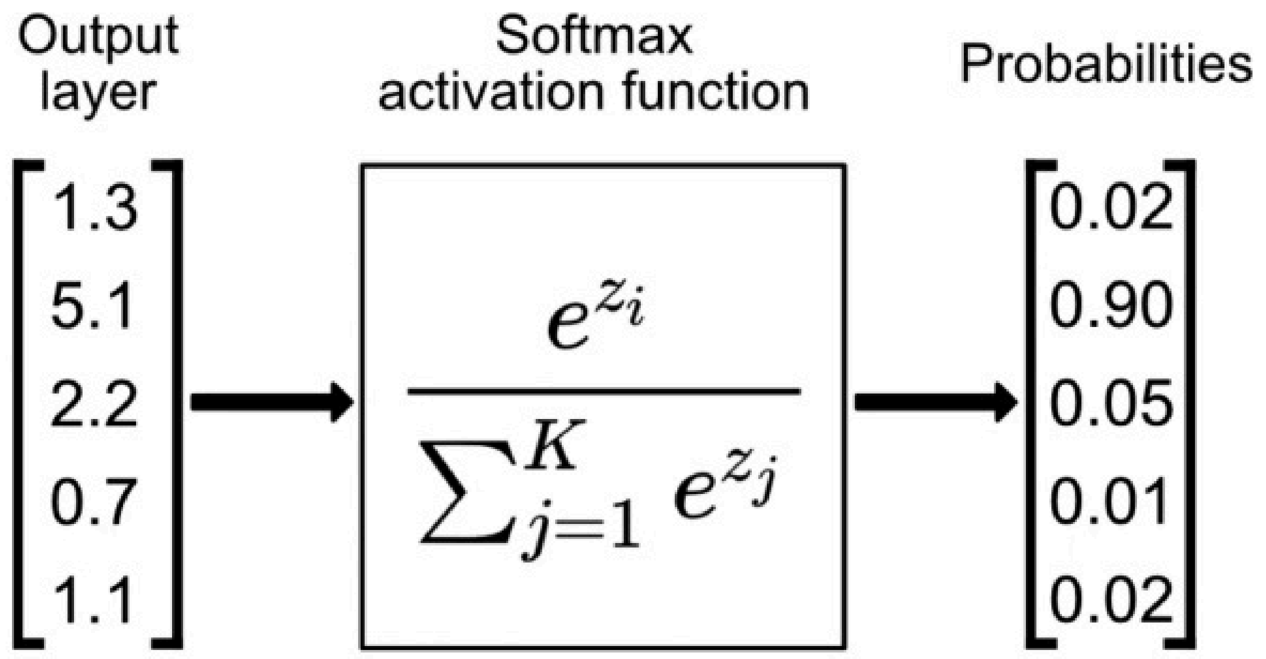
How relevant is token j (the key) to token i (the query)?

Result: a similarity matrix of shape [tokens × tokens]
→ one row per query, one column per key.

Softmax → Normalize into Probabilities

We apply softmax to each row of the similarity matrix:

α_ij = softmax(e_ij)


Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

Now we have attention weights : how much focus each query puts on each token in the sequence.

$$\text{attention}_i = \sum \alpha_{ij} * v_j$$

Concept	**Meaning**
Query (Q)	What this token is asking about ("what do I need to know?")
Key (K)	What each other token offers as context ("what info do I contain?")
Value (V)	The actual content/information that can be retrieved
Q vs K	Gives attention weights (how relevant each token is)
Weights × V	Produces the output (a mix of relevant token values)

Multi-Head Attention

Instead of computing one attention, we split the input into h heads:

Each head uses independent projections:

$$\text{head}_i = \text{Attention}(QW_Q^i, KW_K^i, VW_V^i)$$

Output:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h) \times W_O$$

Visual Metaphor: Orchestra 🎼

Think of the full sentence as a musical composition:

- Violin focuses on melody → Subject (Head 1)
- Drums focus on rhythm → Time (Head 2)
- Trumpet highlights harmony → Location (Head 3)
- Piano enriches the feel → Verb nuances (Head 4)

Each instrument (head) hears the same music, but listens differently.

Concept	**Real World Analogy**	**Code**
Attention	Focused listening	attention_weights
Multi-head	Multiple perspectives	heads = {...}
Embedding slicing	Splitting tasks	head_dim = n_embd // n_head
Parallelism	Orchestra sections	Heads working together

Insight 0: Self-Attention is Not Symmetric

$$QK^T \neq KQ^T \text{ unless } W\_Q = W\_K$$

Therefore, attention can be seen as a **directed graph.**

Insight 1: Attention = Routing Mechanism

The attention matrix routes local information from keys/values into global representations.

Insight 2: Shared Projections

Heads aren’t fully independent.
Concatenated projections show low-rank structure → heads learn to focus on similar subspaces.

Insight 3: Specialized Heads

According to Voita et al., heads fall into:

- Positional
- Syntactic
- Rare-word targeting

Pruning unused heads doesn’t degrade performance much.

Cross-Attention

- The encoder processes the source language (e.g., French).
- The decoder generates the target language (e.g., English).
- Cross-attention is how the decoder accesses the encoder’s output to align and generate correct translations.

What Happens When You Prune Cross-Attention Heads:

- The decoder loses its ability to properly focus on relevant parts of the encoder's output.
- This harms alignment between source and target sequences.
- The model may generate grammatically correct but semantically incorrect translations.

Why It's More Harmful than Pruning Self-Attention:

- Self-attention (within encoder or decoder) mostly models local dependencies (e.g., grammar, syntax). These are often redundant across heads , Voita et al. showed some can be pruned with minimal loss.

Cross-attention, on the other hand:

- Handles global mapping between languages.
- Fewer heads mean less expressive capacity to model complex source-target relationships.

Causal Self-Attention

When predicting a token at position t, the model only "attends" to tokens from positions ≤ t.

This is critical for **autoregressive generation**,
where we predict the next token based only on the previous ones : **no cheating by looking ahead!**

	আমি	বাংলায়	কথা	বলি
আমি	1	0	0	0
বাংলায়	1	1	0	0
কথা	1	1	1	0
বলি	1	1	1	1

Let:

$$X \in R^{B \times T \times d\_model}$$

B: batch size, T: sequence length, d_model: embedding dim

We then compute raw attention matrix:

Each row A_i: contains similarity scores between token i's query and all keys.

Apply Causal Mask

We use a lower triangular mask: $$M_{ij} = \begin{cases} 0 & \text{if } j \le i \\ -\infty & \text{if } j > i \end{cases}$$

Apply it:

A_masked=A+M; A = Previously calculated attention matrix (not the normalized attention score)

Why? So that token i cannot see future tokens j>i

Next Step : Softmax Normalization

We apply softmax along the last axis:

$$\alpha\_ij = \exp(A\_ij)/\sum_k \exp(A\_ik)$$

This gives us attention weights: how much token i attends to token j.

Then, weighted Sum Over Values:

$$output\_i = \sum_j \alpha\_ij \times V\_j$$

This produces the attention output for each head. Then:Concatenate all head

Why This Matters

Without Causal Masking:

Model sees all tokens, including future ones — learns to cheat:
Predict "বাংলায়" using "বলি" (which appears after it).

With Causal Masking:

Model must predict "বলি" using only "আমি”

It learns true sequential patterns, essential for:

- **Language modeling**
- **Code generation**
- **Music generation**

Code Walkthrough

Initialize Layers:

```
self.n_head = cfg.n_head
self.key   = nn.Linear(cfg.n_embd, cfg.n_embd, bias=False)
self.query = nn.Linear(cfg.n_embd, cfg.n_embd, bias=False)
self.value = nn.Linear(cfg.n_embd, cfg.n_embd, bias=False)
self.proj  = nn.Linear(cfg.n_embd, cfg.n_embd, bias=False)
```

cfg : object that represent our model config dict

Each of these nn.Linears is a fully-connected layer:

- W_Q, W_K, W_V, W_O, all with shape [d_model × d_model]
- These are shared across all heads, and are later **reshaped per head**

```
self.attn_drop = nn.Dropout(cfg.dropout)
self.resid_drop= nn.Dropout(cfg.dropout)
```

Two dropout layers:

- One applied to the attention weights (attn_drop)
- One applied to the final output projection (resid_drop)

Causal Mask:

```
mask = torch.tril(torch.ones(cfg.block_size, cfg.block_size))
mask = mask.view(1, 1, cfg.block_size, cfg.block_size)
self.register_buffer("mask", mask)
```

- Lower-triangular matrix with shape (1, 1, T, T)
- Ensures token at position i can only attend to ≤ i
- register_buffer ensures the mask is part of the model but not learnable

```
def forward(self, x):
    B, T, C = x.shape
```

- B: batch size
- T: sequence length
- C: embedding size (should be cfg.n_embd)

```
k = self.key(x).view(B,T,self.n_head,C//self.n_head).transpose(1,2)
q = self.query(x).view(B,T,self.n_head,C//self.n_head).transpose(1,2)
v = self.value(x).view(B,T,self.n_head,C//self.n_head).transpose(1,2)
```

- Project x to key/query/value using Linear(C, C) → shape: [B, T, C]
- Reshape for multi-head: [B, T, h, d_head], where d_head = C // h
- Transpose to [B, h, T, d_head]

So now: q, k, v ∈ ℝ^{B × h × T × d_head}

att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))

- q @ k^T: batched dot-product attention
- Result shape: [B, h, T, T] → attention scores between every token pair
- Scale by 1/√d_k : stabilize gradients

Apply Causal Mask:

```
att = att.masked_fill(self.mask[:, :, :T, :T] == 0, float('-inf'))
```

- Prevents token i from attending to any future tokens j > i
- Fills upper triangle with -infinity, so they become 0 after softmax

Normalize with Softmax:

```
att = F.softmax(att, dim=-1)
att = self.attn_drop(att)
```

Apply Attention to Values

```
y = att @ v
y = y.transpose(1, 2).contiguous().view(B, T, C)
```

- transpose(1, 2) → [B, T, h, d_head]
- .view(B, T, C) flattens back to original embedding dim (C = h × d_head)

Output Projection + Dropout

```
y = self.resid_drop(self.proj(y))
```

- Final linear layer W_O ∈ ℝ^{C × C}
- Adds a learnable mix of all heads
- Followed by dropout

Why transpose(1, 2)?

After reshaping Q, K, V like this:

```
q = q.view(B, T, n_head, head_dim)
```

- You get shape:
  → [batch_size, seq_len, n_heads, head_dim]
- But PyTorch expects batch-first and head-first format for efficient multi-head attention computation, like:
  → [batch_size, n_heads, seq_len, head_dim]

So we do:

```
q = q.transpose(1, 2)  # swaps dim 1 (seq_len) and dim 2 (n_heads)
```

transpose(1, 2) rearranges dimensions so that each attention head is grouped and can compute its scores independently in [B, n_heads, T, head_dim] format.

Assessment and Summary

- Why do we need positional embeddings in transformers?
- Explain the difference between encoder and decoder-only architectures
- What would happen if we removed causal masking from our model?
- How many parameters are in the attention mechanism of one layer?
- Why do we use multiple attention heads instead of one large head?
- What is the purpose of the residual connections in transformer blocks?

code