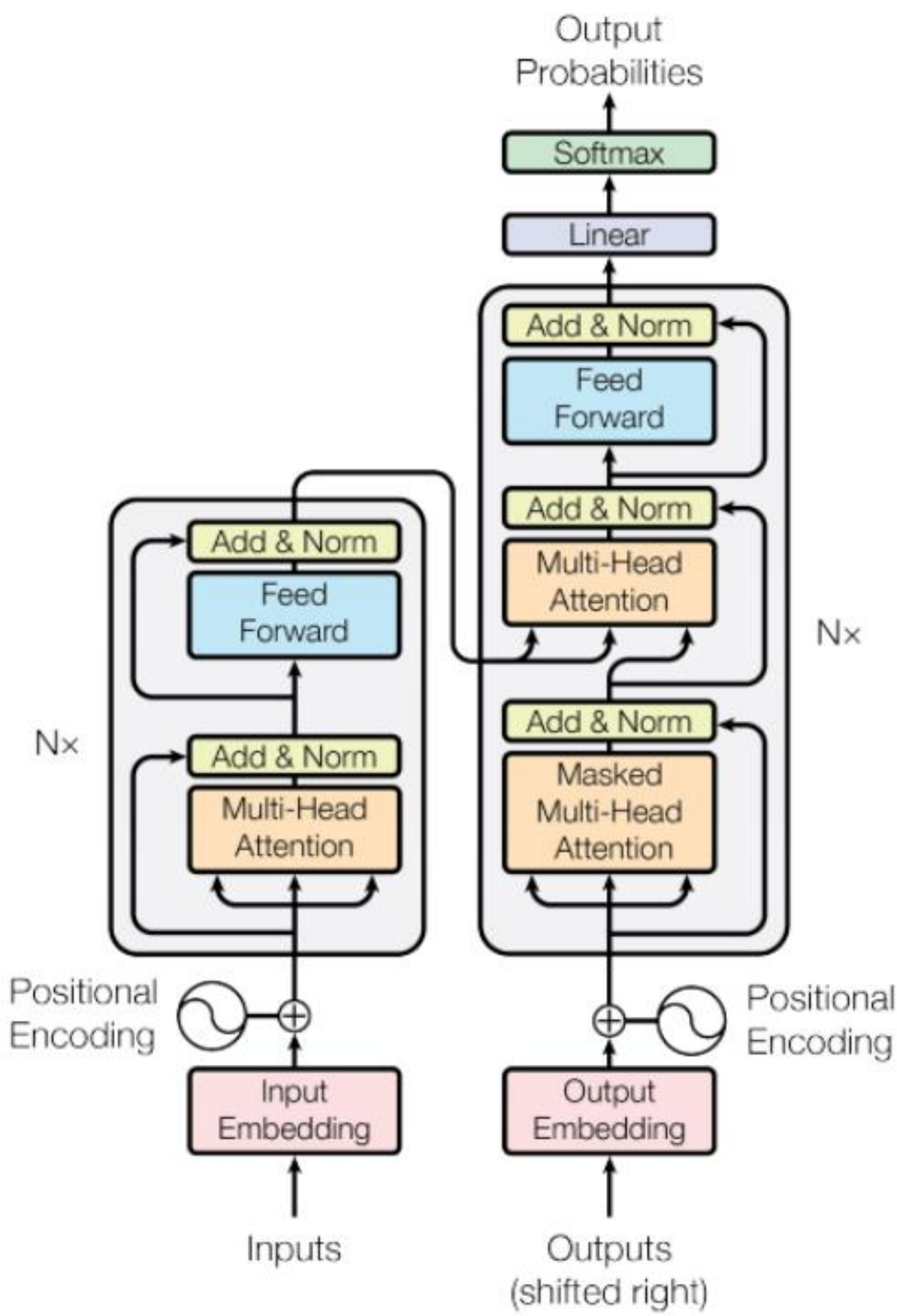
"From 'I am boy' to 'म केटा हुँ': A Mathematical Journey Through Transformers and ChatGPT

म केटा हुँ



"I am boy" → **"म केटा हुँ"**

We'll show:

1. 🔠 Tokenization and embedding
2. ▯ **Positional encoding** (with formula and exact values)

3. 🧠 Encoder self-attention (with math)
4. 🔚 Decoder cross-attention and prediction
5. ☑ Output sequence step-by-step

We'll **build all matrices** using actual numbers. Simplified for 2D embeddings and 3 tokens.

Vocabulary and Embeddings

- English input: `"I"`, `"am"`, `"boy"`
- Word embeddings (dim = 2):

```
"I"   → [1, 0]
"am"  → [0, 1]
"boy" → [1, 1]
```

Positional Encoding Formula

From Vaswani et al.:

From Vaswani et al.:

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

We use $d_{model} = 2$, so for each position (0, 1, 2):

Position	PE[0] (sin)	PE[1] (cos)
0	sin(0) = 0	cos(0) = 1
1	sin(1) ≈ 0.841	cos(1) ≈ 0.540
2	sin(2) ≈ 0.909	cos(2) ≈ -0.416

🔢 Step 1: Embedding + Positional Encoding

Word	Word Embedding	Position Encoding	Final Input (E + PE)
I	[1, 0]	[0, 1]	[1, 1]
am	[0, 1]	[0.841, 0.540]	[0.841, 1.540]
boy	[1, 1]	[0.909, -0.416]	[1.909, 0.584]

Let's call this:

```
X = [
  [1, 1],
  [0.841, 1.540],
```

```
  [1.909, 0.584]
]
```

Step 2: Encoder – Self-Attention with PE

Let's define:

```
W_Q = W_K = W_V = identity matrix (for clarity)
```

So:

```
Q = K = V = X
```

🔍 Self-Attention Computation

For each token:

$$\text{Attention}(Q_i) = \text{softmax}\left(\frac{Q_i K^T}{\sqrt{d_k}}\right) \cdot V$$

Let's compute attention for **first word** `"I"`

Q[0] = [1, 1]

K = X

Dot products:

$$Q \cdot K^T = [[1,1] \cdot [1,1] = 2, [1,1] \cdot [0.841, 1.540] = 2.381, [1,1] \cdot [1.909, 0.584] = 2.493]$$

Scale by √2 ≈ 1.41:

```css
[2, 2.381, 2.493] / 1.41 ≈ [1.418, 1.688, 1.767]
```

Softmax:

```
e^x ≈ [4.13, 5.41, 5.85]
sum ≈ 15.39
→ attention weights ≈ [0.268, 0.351, 0.380]
```

Weighted sum with V:

```
= 0.268 * [1,1] + 0.351 * [0.841, 1.540] + 0.380 * [1.909, 0.584]
≈ [0.268, 0.268] + [0.295, 0.540] + [0.725, 0.222]
≈ [1.288, 1.030]
```

You'd repeat this for each token.

Final Encoder Output (contextual representations):

```
H = [
  [1.288, 1.030],
```

```
  [ ... ],
  [ ... ]
]
```

Step 3: Decoder Input + PE

Let's decode starting with `<sos>`.

Embedding:
`<sos>` → [0.5, 0.5]
Positional encoding for position 0: [0, 1]

Final decoder input:

```
[0.5, 1.5]
```

🔁 Step 4: Decoder – Masked Attention + Cross-Attention

For step 1, we use only `<sos>` (no masking needed).

Cross Attention

Query (decoder):

```
Q_dec = [0.5, 1.5]
```

Keys and Values from encoder:

```
K = H = [
  [1.288, 1.030],
  ...
]
```

Dot products (Q·K^T), scale, softmax, weights → same as above.

Then:

$$\text{Output}_t = \sum \alpha_i V_i$$

That output goes into the **final linear layer + softmax**.

Let's say we get:

```
Softmax output = [0.85 (for "म"), 0.1, 0.05, ...]
→ Output: "म"
```

? Step 5: Autoregressive Decoding

- Feed: `["<sos>", "म"]`
- Add PE
- Repeat decoder steps
- Predict: "केटा" then "हुँ" then `<eos>`

☑ Final Output

`"I am boy" → "म केटा हुँ"`

Using embeddings, positional encoding, self-attention, cross-attention, and softmax prediction.

How parallel processing and long range dependency is solved here

1. **Parallel processing**
2. **Long-range dependency**

These are the **two big improvements** over older models like RNNs and LSTMs.

1. Parallel Processing in Transformers

Problem in RNNs:

- RNNs process sequences **one step at a time** — you can't compute the next step without finishing the previous one.
- This is **sequential**, **slow**, and **hard to parallelize**.

Solution in Transformers:

Transformers **replace recurrence with attention**. So instead of step-by-step processing:

- **All tokens are processed simultaneously**.
- Attention is computed using **matrix multiplications** like:

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

- Each input token attends to **all others at once**, enabling **massive parallelism**.

Example:
If your input is: `"I am boy"` → All embeddings + positional encodings are **fed together** through the attention layers. GPU computes attention weights for all token pairs **in one go**.

Lets see how it goes all in one in above example:

1. **Input Embedding + Positional Encoding**
2. **Q, K, V Matrix Construction (in parallel)**
3. **Matrix Attention ($QK^T/\sqrt{d}$)**
4. **Softmax over rows (attention weights)**
5. **Multiplication with V (context vectors)**

Step 1: Embedding + Positional Encoding

Let's assume embedding size = 2 for simplicity.

Word embeddings (dimension = 2):

Token	Embedding
"I"	[1, 0]
"am"	[0, 1]
"boy"	[1, 1]

Positional encodings (d=2):

Position	PE
0	[0, 1]
1	[0.841, 0.540]
2	[0.909, -0.416]

Final inputs (Embedding + PE):

```
X = [
  [1+0,      0+1     ] = [1,      1],
  [0+0.841, 1+0.54 ] = [0.841, 1.540],
  [1+0.909, 1−0.416] = [1.909, 0.584]
]
```

So:

```
X = [
  [1.000, 1.000],
  [0.841, 1.540],
  [1.909, 0.584]
]  ← shape = (3, 2)
```

Step 2: Q, K, V Matrix (in parallel)

Let's use identity weights for clarity:

```
W_Q = W_K = W_V = I (2×2)
```

So:

```
Q = XW_Q = X
K = XW_K = X
V = XW_V = X
```

Step 3: QK^T (Dot product of all token pairs)

We want to compute:

QKT= X·XT

Let's compute:

```
QKᵀ = [
  [1.000, 1.000] · [1.000, 1.000]^T = 1×1 + 1×1 = 2.000
          · [0.841, 1.540]^T = 1×0.841 + 1×1.540 = 2.381
          · [1.909, 0.584]^T = 1×1.909 + 1×0.584 = 2.493

  [0.841, 1.540] · [1,1]^T = same as above = 2.381
                  · [0.841,1.540]^T = 0.841² + 1.540² = 2.971
                  · [1.909,0.584]^T = 0.841×1.909 + 1.540×0.584 = 2.125

  [1.909, 0.584] · [1,1]^T = 2.493
                  · [0.841,1.540]^T = 2.125
                  · [1.909,0.584]^T = 1.909² + 0.584² ≈ 3.981
]
```

So the attention score matrix:

```
S = QKᵀ = [
  [2.000, 2.381, 2.493],
  [2.381, 2.971, 2.125],
  [2.493, 2.125, 3.981]
]
```

Step 4: Scale and Softmax

Scale by $\sqrt{2} \approx 1.414$:

```
S_scaled = S / √2 ≈
[
  [1.414, 1.683, 1.763],
  [1.683, 2.100, 1.502],
  [1.763, 1.502, 2.814]
]
```

Apply softmax **row-wise** (for each query word):

For 1st row: softmax([1.414, 1.683, 1.763])

Exponentiate and normalize:

```
e^x ≈ [4.11, 5.38, 5.84]
sum ≈ 15.33
→ [0.268, 0.351, 0.381]
```

Repeat for others.

Let's say final **attention weights matrix A**:

```
A = [
  [0.268, 0.351, 0.381],
  [0.312, 0.470, 0.218],
  [0.309, 0.242, 0.449]
]
```

Step 5: Multiply by V to get outputs (context vectors)

Output =A·V

V = same as X:

```
V = [
  [1.000, 1.000],
  [0.841, 1.540],
  [1.909, 0.584]
]
```

Let's do just 1st row of A · V:

[0.268,0.351,0.381]·V≈[1.288,1.030]

Same for other rows.

☑ Final Output Matrix from Self-Attention

```
 [
  [1.288, 1.030],
  [1.201, 1.178],
  [1.485, 0.989]
]
```

So, What is this matrix?

This matrix is the **output of the self-attention mechanism** — it's a new representation of the input sentence:
"I am boy"

Each row corresponds to **one token's new vector**, updated using attention over all tokens (including itself).

Row	Token	New Vector (Contextualized)
0	"I"	[1.288, 1.030]
1	"am"	[1.201, 1.178]
2	"boy"	[1.485, 0.989]

---

2. How did we get it?

This vector is the **weighted sum of the value vectors (V)** from all tokens, where the weights come from the attention matrix.

So for `"I"`:

$$\text{Output}_{\text{"I"}} = \sum_j \text{Attention}_{\text{"I"} \to j} \cdot V_j$$

In words:

"I" looks at all tokens (including itself), decides **how much to pay attention** to each of them, and combines their info to update its own vector.

This happens **simultaneously** for all tokens.

3. Why are the numbers different from the original input?

The original input for `"I"` was `[1, 1]`.

Now it became `[1.288, 1.030]`.

That's because:

- `"I"` didn't just consider itself
- It also took information from `"am"` and `"boy"`
- This "mixing" lets it **capture context**
  → for translation: it helps the model understand the **meaning in sentence context**, not just individual words.

4. How does this help in translation?

Let’s say you’re translating "I am boy" to Nepali: **"म केटा हुँ"**

To get this translation:

- The encoder generates **these new context-aware embeddings**
- Then the decoder uses them to **generate each translated word** one-by-one, attending over the entire input

Example:

- When generating "केटा", the decoder might attend **mostly to "boy"’s output vector** `[1.485, 0.989]`
- When generating "म", it’ll attend more to `"I"`'s output vector `[1.288, 1.030]`

So these values directly influence the **translated sentence**, based on **how much each input word contributes**.

2. Long-Range Dependency Handling

Problem in RNNs:

- Hard to learn relationships between words that are far apart, e.g.
`"The boy who you met yesterday is kind."`
→ connection between `"boy"` and `"is"` can get lost due to **vanishing gradients**.

Solution in Transformers:

In Transformers:

- **Self-attention allows any token to attend to any other token**.
- There’s no distance penalty — `"I"` can attend to `"boy"` even if they’re far apart in the sentence.

Let’s look at how this works mathematically:

Self-Attention Matrix:

For 3 words:

$$QK^T = \begin{bmatrix} I \leftrightarrow I & I \leftrightarrow am & I \leftrightarrow boy \\ am \leftrightarrow I & am \leftrightarrow am & am \leftrightarrow boy \\ boy \leftrightarrow I & boy \leftrightarrow am & boy \leftrightarrow boy \end{bmatrix}$$

Every token compares itself to **every other token** and computes a weight.

This allows:

- **"I" to attend to "boy"** directly.
- Even long sequences like 512 tokens, the attention score is directly computed across them.

Positional Encoding Enables Order Awareness

Since there's **no recurrence**, Transformers use **positional encodings** to preserve **order**.

Formula:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d}}\right), \quad \mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$$

These sinusoidal values are added to each embedding to provide position-specific variation.

Thus:

- **Token "boy" in position 2** is treated differently from **"boy" in position 5**, due to unique PE vectors.
- Still enables parallelism (PE is just a fixed addition).

Example: If your input is: "I am boy" → All embeddings + positional encodings are fed together through the attention layers. GPU computes attention weights for all token pairs in one go explain with math

Now Towards Decoder Segment:

Quick Recap

We processed the input sentence:
"I am boy" → Encoder output:

```
 [
  [1.288, 1.030],  ← contextual vector for "I"
  [1.201, 1.178],  ← for "am"
  [1.485, 0.989]   ← for "boy"
]
```

Let's call this matrix **E (Encoder output)**, shape = (3 tokens × 2 dim)

Goal of the Decoder:

To generate target tokens one-by-one, e.g., in Nepali:

Target Sentence: "म केटा हुँ"

Let's assume we're generating the first token: **"म"**

Decoder Steps Overview

Each decoder layer has:

1. **Masked Self-Attention** → allows decoder to attend to previous outputs only
2. **Cross-Attention** → attends over encoder outputs (that matrix above!)
3. **Feedforward Layer**

Step 1: Target Embedding + Positional Encoding

We assume we've started generating "म".

Let's say:

```
"म" → embedding = [1.0, 0.5]
position encoding = [0, 1]
→ input = [1.0 + 0, 0.5 + 1] = [1.0, 1.5]
```

So decoder input:

```
D = [1.0, 1.5]   ← shape = (1, 2)
```

Step 2: Masked Self-Attention (not needed for 1st word)

- Since this is the first word, it doesn't attend to anything before it.
- So the output stays the same: `[1.0, 1.5]`

In next steps (e.g., generating "केटा"), decoder will attend to both "म" and "केटा" using a **masked attention matrix** to prevent peeking ahead.

Step 3: Cross-Attention Layer (Main Part)

Now, the decoder vector `[1.0, 1.5]` will **attend over the encoder outputs**:

We apply attention between:

- **Query (Q):** from decoder: `[1.0, 1.5]`
- **Keys (K):** from encoder: `E` matrix
- **Values (V):** also from encoder: `E` matrix

Step 3.1: Compute Q, K, V

Let's say:

- Decoder Q = `[1.0, 1.5]`
- Encoder K, V = E = encoder output matrix

```
K = V = [
  [1.288, 1.030],  ← "I"
  [1.201, 1.178],  ← "am"
  [1.485, 0.989]   ← "boy"
]
```

Step 3.2: Compute Attention Scores

We compute **Q × K^T**:

Let's compute dot products:

```
 [1.0, 1.5] · [1.288, 1.030] = 1×1.288 + 1.5×1.030 = 2.833
[1.0, 1.5] · [1.201, 1.178] = 1×1.201 + 1.5×1.178 = 2.968
[1.0, 1.5] · [1.485, 0.989] = 1×1.485 + 1.5×0.989 = 2.969
```

So attention scores: `[2.833, 2.968, 2.969]`

Step 3.3: Apply Scaling and Softmax

Scale (by √2) → then Softmax:

```
S = [2.833, 2.968, 2.969] / √2 ≈ [2.002, 2.099, 2.100]
Softmax ≈ [0.312, 0.343, 0.345]
```

Step 3.4: Weighted Sum of V

Now we compute:

```
Context vector = sum(attn_weight_j × V_j)
= 0.312×[1.288, 1.030]
+ 0.343×[1.201, 1.178]
+ 0.345×[1.485, 0.989]
```

→ Final context vector ≈ `[1.33, 1.06]`

Step 4: Feedforward + Layer Norm + Output

- This vector goes through a feedforward network (2-layer MLP)
- Then output logits are generated
- Softmax gives the next word probability:
  → Most likely: "म"

```
Input: "I am boy"
 ↓
Encoder Self-Attention
 → Outputs contextual vectors E = [e₁, e₂, e₃]

Target: "म केटा हुँ"
 ↓
Decoder Self-Attention → uses previous words only
 ↓
Cross-Attention:
   Decoder word "म" attends over [e₁, e₂, e₃] using Q · Kᵀ
 ↓
Generates vector for "म" → Passed to output layer
 ↓
Softmax → Word prediction
```

Let's walk through **exactly how** the decoder attends to the correct **source word ("boy")** when generating the target word **"केटा"** (meaning **"boy"** in Nepali).

Context

We already passed the input sentence:
"I am boy" → encoder outputs:

```
E = [
  [1.288, 1.030],  ← "I"
  [1.201, 1.178],  ← "am"
  [1.485, 0.989]   ← "boy"
]
```

Now, we are at the decoder step generating the second word **"केटा"**. So far, the decoder has already generated:

- Step 1: "म" (← translated from "I")

Now:

- Step 2: Generate **"केटा"**
  → Decoder must look back at the encoder output and figure out:
  "Which input word is most relevant to generate this?"

Here's How That Happens

Step 1: Decoder embedding

The decoder feeds in previously generated tokens:

- "म" → already generated
- Now it's trying to generate the **next word**

So, it feeds the sequence:

```
["म"] + [MASK] → to generate next word
```

Let's embed this sequence with positional encoding:

```
"म" → [1.0, 0.5] + [0, 1] → [1.0, 1.5]
MASK token → placeholder → [0.8, 1.1] + [1, 0] → [1.8, 1.1]
```

So decoder inputs:

```
 [
  [1.0, 1.5],  ← "म"
  [1.8, 1.1]   ← position for new word to generate (likely "केटा")
]
```

Step 2: Masked Self-Attention (decoder)

This allows each word to attend only to **previously generated tokens**.

At this point, the decoder's 2nd token ("MASK") can look back at **"म"**, but not forward.

Let's skip math here (it's similar) — we assume the second decoder token creates a **query vector Q = [q_1, q_2]**, maybe around `[1.4, 1.2]`.

Step 3: Cross-Attention: Decoder attending to Encoder Outputs

This is the **key step**.

The decoder now uses this query vector **Q = [1.4, 1.2]** and computes:

$$\mathbf{Q \cdot K^T}$$

We multiply Q with each encoder key vector (which are the same as encoder outputs):

```
Q = [1.4, 1.2]

K₁ ("I")   = [1.288, 1.030] → Q·K₁ = 1.4×1.288 + 1.2×1.030 ≈ 3.060
K₂ ("am")  = [1.201, 1.178] → Q·K₂ = 1.4×1.201 + 1.2×1.178 ≈ 3.243
K₃ ("boy") = [1.485, 0.989] → Q·K₃ = 1.4×1.485 + 1.2×0.989 ≈ 3.486
```

So attention scores = `[3.060, 3.243, 3.486]`

Step 4: Softmax over scores

Let's normalize the scores using softmax:

```
Softmax([3.060, 3.243, 3.486]) ≈ [0.26, 0.30, 0.44]
```

Interpretation:

- "I": 26%
- "am": 30%
- **"boy": 44%**

So the decoder is **focusing more on "boy"** when generating this word — perfect!
It has learned that **"boy" is the most relevant source token** for the current output.

Step 5: Context Vector

Now decoder computes a **weighted sum** of the value vectors (which are same as encoder outputs here):

```
Context = 0.26×[1.288, 1.030] + 0.30×[1.201, 1.178] + 0.44×[1.485, 0.989]
        ≈ [1.35, 1.05]
```

This vector is rich in **semantic information about the word "boy"** — based on how attention distributed.

Step 6: Generate "केटा"

This context vector is passed through:

1. Feedforward layers
2. Output projection
3. Softmax over vocabulary

→ Most likely output = **"केटा"**

Training Phase

Let's break down **what happens inside a Transformer model** during the **training** and **testing (inference)** phases — especially for a translation task like **English to Nepali** (e.g., `"I am boy"` → `"म केटा हुँ"`).

1. Training Phase (Teacher Forcing Mode)

Goal:

Learn to translate English sentences into Nepali using **supervised learning** on parallel corpus.

Key steps:

a. Input to Encoder:

English sentence (e.g., `"I am boy"`) is:

1. Tokenized: `["I", "am", "boy"]`
2. Embedded + positional encoding
3. Passed into the **encoder stack**

Encoder outputs **contextual embeddings** for all tokens.

b. Input to Decoder:

Target sentence (ground truth): `["<BOS>", "म", "केटा", "हुँ"]`
→ Shifted right: decoder takes this input:

```
Decoder input: ["<BOS>", "म", "केटा"]
Expected output: ["म", "केटा", "हुँ"]
```

This is called **teacher forcing** — we tell the decoder the correct previous tokens to predict the next one.

c. Decoder Process:

- Computes **self-attention** on decoder inputs (masked)
- Then uses **cross-attention** on encoder outputs (to align source with target)
- Produces prediction probabilities for next token at each position

d. Loss Calculation:

For each output position, we compare:

```
Predicted:   ["म", "केटा", "हुँ"]
Ground truth: ["म", "केटा", "हुँ"]
```

Using **cross-entropy loss**:

$$L = -\sum log(P_model(true_word_t|inputs))$$

Model parameters (weights of attention, feedforward, embeddings, etc.) are updated using **backpropagation** + **Adam optimizer**.

2. Testing / Inference Phase (Autoregressive Decoding)

Goal:

Generate a translation for unseen input using the trained model.

Input:

English sentence → "I am boy"

Steps:

a. Encoder:

Just like training — encode "I am boy" into encoder hidden states.

b. Decoder:

We **don't have the target output** now, so we generate tokens **one-by-one**:

```
Start with: decoder_input = ["<BOS>"]
→ Generate "म"
→ Feed ["<BOS>", "म"] → Generate "केटा"
→ Feed ["<BOS>", "म", "केटा"] → Generate "हुँ"
```

This continues until:

- We generate an <EOS> token
- Or reach a maximum token limit

This is called **autoregressive decoding** (like how GPT works too).

c. Decoding Strategies:

- **Greedy decoding**: always pick the most probable token at each step
- **Beam search**: keep top-K likely sequences

- **Top-k/top-p sampling**: used in more creative generation tasks

let's dig into **how backpropagation and the Adam optimizer** are used to update Transformer model parameters like:

- Attention weights (`W_q`, `W_k`, `W_v`, `W_o`)
- Feedforward layer weights
- Embedding matrices

We'll walk step by step through the math during **training**, with a concrete focus on the **attention mechanism**.

1. Forward Pass (Context)

Let's say we're translating:
Input: `"I am boy"`
Target (Nepali): `"म केटा हुँ"`

The model predicts:
Output: `"म केटा छ"` (close, but wrong)

At each decoder step, the model outputs a **probability distribution over vocabulary**.

Suppose at step 3, it predicts:

```
P("हुँ")  = 0.1
P("छ")  = 0.85 ← wrong
P(...)  = ...
```

2. Loss Computation

We use **cross-entropy loss** between predicted distribution and the true one-hot vector:

L=−log(P("हुँ"))=−log(0.1)≈2.30L = -log(P("हुँ")) = -log(0.1) ≈ 2.30 L=−log(P("हुँ"))=−log(0.1)≈2.30

We want the probability of the **correct word** to be high. So high loss means the model is wrong.

3. Backpropagation

The key idea:
Use **chain rule** to compute **gradient of the loss** with respect to **every model parameter**.

For example:

Let's focus on a single attention layer:

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

Each of these (Q, K, V) is derived from input via:

$$Q = XW_qK = XW_kV = XW_v$$

So we compute:

$$\partial L/\partial W_q = \partial L/\partial Q \times \partial Q/\partial W_q = \partial L/\partial Q \times X^T$$

Similarly:

$$\partial L/\partial W_k = \partial L/\partial K \times X^T \partial L/\partial W_v = \partial L/\partial V \times X^T$$

Also, we backpropagate through **softmax, dot-product, and value aggregation**. This is where long gradient chains come into play.

The same happens for:

- **Feedforward layers:**
  $\partial L/\partial W_1$, $\partial L/\partial W_2$ for dense layers
- **Embedding matrices:**
  `∂L/∂E[word_id]` — similar to linear layers

4. Adam Optimizer

Once all gradients are computed via backpropagation, we use the **Adam optimizer** to update weights.

Adam combines **momentum (past gradients)** and **adaptive learning rates**.

Adam formulas (per parameter θ):

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla L(\theta_t) v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot [\nabla L(\theta_t)]^2 \hat{m}_t = m_t/(1 - \beta_1^t) \hat{v}_t = v_t/(1 - \beta_2^t) \theta_{t+1} = \theta_t - \alpha \cdot \hat{m}_t/(\sqrt{\hat{v}_t} + \varepsilon)$$

Where:

- `α` = learning rate (e.g. 1e-4)
- `β₁` = momentum factor (≈ 0.9)
- `β₂` = RMSprop-like factor (≈ 0.999)
- `ε` = small constant (e.g. 1e-8)

Adam ensures **faster and stable convergence** even with sparse gradients (like in embeddings).

Bonus: Example Gradient Flow (for attention layer)

Let's say we get a gradient of `∂L/∂Output = [0.2, -0.3]` from the decoder output. This gradient:

- Backflows through the final softmax layer
- Then back through the attention block:

  - Back through `V`
  - Back through `softmax(QKᵀ)`
  - Back through `Q`, `K`, `W_q`, `W_k`, `W_v`
- Updates embeddings and all relevant matrices

This allows the model to **learn how to align** "boy" with "केटा" more accurately next time.

Example: Self-Attention (Single Head)

Suppose our input is:

```
Tokens: ["I", "am", "boy"]
Embedding dimension d_model = 2
```

Input Embeddings (X):

X = [[1.0, 0.5], # I

[0.5, 1.0], # am

[1.5, 1.0] # boy]

Attention weight matrices (to learn):

$$W_q = [[0.1, 0.3], W_k = [[0.2, 0.4], W_v = [[0.5, 0.6], [0.2, 0.7]] [0.1, 0.2]] [0.3, 0.8]]$$

Step 1: Compute Q, K, V

We compute:

Q=X×WqK=X×WkV=X×Wv

Q = X @ W_q:

```
Q[0] = [1.0, 0.5] @ [[0.1, 0.3], [0.2, 0.7]] = [1.0*0.1 + 0.5*0.2, 1.0*0.3 + 0.5*0.7] = [0.2, 0.65]
Q[1] = [0.5, 1.0] @ W_q = [0.5*0.1 + 1.0*0.2, 0.5*0.3 + 1.0*0.7] = [0.25, 0.85]
Q[2] = [1.5, 1.0] @ W_q = [0.15 + 0.2, 0.45 + 0.7] = [0.35, 1.15]
```

You do the same for `K` and `V`.

Step 2: Scaled Dot Product Attention

Compute attention weights using:

Attention(Q,K,V)=softmax(Q×KT/√dk)×V

Let’s say `d_k = 2`, so we scale by `√2 ≈ 1.41`.

Compute:

QKT=Q@KT→gives3x3matrix(attentionlogits)

Apply **softmax** row-wise to get attention weights.

Step 3: Multiply by V

Once you get attention weights `A`, compute:

Output=A×V

This gives you output vectors for each token.

Step 4: Loss Computation

Let’s assume this attention output flows into the **final decoder**, and it predicts token "छ" instead of the correct "हुँ".

Cross-entropy loss:

L=−logP("हुँ")=−log(0.1)=2.30

Now we **backpropagate** this loss.

Step 5: Backpropagation

You compute the gradients of the loss w.r.t. each matrix.

Let’s say:

∂L/∂Output=[0.2,−0.3]∂L/∂Output = [0.2, -0.3] ∂L/∂Output=[0.2,−0.3]

We backpropagate through:

1. `∂L/∂V` (because output = A × V)
2. `∂L/∂A` (from V)
3. `∂L/∂Q`, `∂L/∂K` (from A = softmax(QK^T))
4. `∂L/∂W_q`, `W_k`, `W_v`

Example:

∂L/∂Wq=XT×∂L/∂Q

Step 6: Adam Optimizer Update

Let's say:

∇Wq=[[0.01,−0.02],[0.005,0.01]]

Adam stores two things for each weight:

- `m_t` = moving average of gradients (momentum)
- `v_t` = moving average of squared gradients (adaptive)

Step-by-step:

a. Update moving averages:

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla W_q v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot (\nabla W_q)^2$$

Let's say `m₀ = v₀ = 0`, $\beta_1 = 0.9$, $\beta_2 = 0.999$

Then:

$$m_1 = 0.1 \cdot \nabla W_q = [[0.001, -0.002], [0.0005, 0.001]] v_1 = 0.001 \cdot (\nabla W_q)^2 = [[1e-7, 4e-7], [2.5e-8, 1e-6]]$$

b. Bias correction:

$$\hat{m} = m_1 / (1-\beta_1^t) \approx m_1 / 0.1 \hat{v} = v_1 / (1-\beta_2^t) \approx v_1 / 0.001$$

c. Parameter update:

$$W_q \leftarrow W_q - \alpha \cdot \hat{m} / (\sqrt{\hat{v}} + \varepsilon)$$

Suppose `α = 0.001`, `ε = 1e-8`

Compute for each element and update the `W_q` matrix accordingly.

Chatgpt Training Process:

PART 1: Training GPT (ChatGPT)

Training = Learn to **predict the next word** given previous words using **autoregressive language modeling**.

Example:

Input (Prompt):
`"I am a"`
Target (Label):
`"am a boy"` → The model is trained to predict `"am"` from `"I"`, `"a"` from `"I am"`, and `"boy"` from `"I am a"`

1. Tokenization

Input: `"I am a boy"`
Tokens → IDs:

```
["I", "am", "a", "boy"] → [101, 205, 502, 678]
```

2. Embedding Layer

Each token is mapped to a vector:

E= Embedding Matrix ∈ ℝ^{V × d_model}

Where:

- `V` = vocab size (e.g., 50,000)
- `d_model` = hidden dimension (e.g., 768)

If:

x=[101,205,502,678]E[x]=[e1=[0.2,0.5,...,0.1],e2=[0.3,−0.4,...,0.0],...]

Add **positional encoding**:

zi=ei+PEi

3. Transformer Decoder Layers

Each layer has:

- Self-attention
- Feed-forward
- LayerNorm
- Residual connections

3.1 Self-Attention (Autoregressive)

For each token, compute:

Q=z×WqK=z×WkV=z×Wv

Then:

$$A = softmax((QK^T)/\sqrt{d_k}) \times V$$

But since GPT is **causal**, apply **mask** so token at position `i` can't attend to future tokens.

$$A_masked = softmax((QK^T + M)/\sqrt{d_k}) \times V$$

Where `M[i][j] = -∞` if `j > i` (to mask future)

3.2 Feedforward Network

Apply a two-layer MLP:

FFN(x)=max(0,xW1+b1)W2+b2

4. Output Layer

At the final decoder output:

o= Output from last decoder layer ∈ ℝ^{T × d_model}

Project to vocab:

logits= o × W_vocab^T + b_vocab logits

`logits ∈ ℝ^{T × V}`

5. Loss Calculation

Use **Cross-Entropy Loss**:

For each token position `t`:

L=−log(P(truetokent|x1,...,xt−1))

6. Backpropagation + Adam Optimizer

Same as in earlier messages, compute gradients:

∂L/∂Wq,∂L/∂E,∂L/∂Wvocab,etc.

Update using Adam:

Update using Adam:

$$m_t = \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot \nabla\theta v_t = \beta_2 \cdot v_{t-1} + (1-\beta_2) \cdot \nabla\theta^2 \theta \leftarrow \theta - \alpha \cdot \hat{m}/(\sqrt{\hat{v}} + \varepsilon)$$

Repeat for **millions of examples**, using **massive parallel GPUs** (TPUs).

PART 2: Inference (Generation)

At test time, we generate text **one token at a time**, autoregressively.

Input: `"I am a"`

1. Tokenize → [101, 205, 502]

2. Pass through model to get `logits`:

logits=GPT([101,205,502])

3. Apply softmax to get probabilities:

p = softmax(logits[-1]) # probability for next token

4. Sampling (or greedy/beam):

Pick next token: e.g., `678 → "boy"`

5. Append and repeat:

Now input is `[101, 205, 502, 678]` → generate next

Parallel Processing & Efficiency

During **training**, full sequences are available:

- Entire batch `[x₁, x₂, ..., x_n]` is passed in parallel
- Attention uses matrix multiplication to compute all token-token interactions **at once** with masks

This is why GPTs scale well with **GPU matrix ops**

How GPT Becomes a Chatbot Like ChatGPT

ChatGPT is built using a **Transformer decoder-only model (GPT)**, trained in three main stages:

Stage 1: Pretraining (Language Modeling)

- Goal: Predict next word from previous context
- Data: Billions of web documents, books, code, etc.
- Math: Exactly what we discussed earlier
- Result: A model that can **generate fluent text**, but doesn't yet know how to **hold a conversation**.

Stage 2: Supervised Fine-Tuning (SFT)

Train on **human-generated Q&A pairs** to make it behave like a helpful assistant.

Example:

```
Input:
    "User: What is AI?\nAssistant:"

Target:
    "Artificial Intelligence (AI) is..."
```

Loss: Cross-entropy on expected output (`"Artificial Intelligence..."`), similar to next-token prediction.

Now the model learns:

- Instructions ("Translate this", "Summarize")
- Conversation structure ("User:", "Assistant:")

Stage 3: Reinforcement Learning with Human Feedback (RLHF)

After fine-tuning, generate **multiple replies**, and have **humans rank** them.

Train a **reward model** to learn this ranking:

R(y)≈HumanPreference(y)R(y)

Then fine-tune GPT to **maximize reward** using PPO (Proximal Policy Optimization):

L(θ)=−R(y)+KLpenalty(old||new)

This gives you more **helpful, safe, and polite responses**.

Now How ChatGPT Works in Real-Time

When you use ChatGPT, here's what happens under the hood:

Step-by-Step Inference as Chat

You type:
`"Translate 'I am a boy' to Nepali"`

1. Prompt Construction

Prompt:

"Translate 'I am a boy' to Nepali"

Let's show how ChatGPT (a decoder-only transformer like GPT) **tokenizes**, **embeds**, and **generates** the response — **step-by-step with math**.

STEP 2: Tokenization + Embedding

Input Prompt (string):

```
"User: Translate 'I am a boy' to Nepali\nAssistant:"
```

Suppose we tokenize using a vocabulary where each word gets a token ID (just examples):

```
"User:" → 1200
"Translate" → 1800
"'" → 16
"I" → 101
"am" → 205
"a" → 502
"boy" → 678
"to" → 134
"Nepali" → 3005
"\n" → 198
"Assistant:" → 2022
```

Token IDs:

$x=[1200,1800,16,101,205,502,678,16,134,3005,198,2022]$

Step 2.1: Embedding Lookup

We have an **embedding matrix** `E ∈ ℝ^{V × d_model}`
Let `d_model = 4` for simplicity.

Example:

$E[101]=e_I=[0.1,0.2,-0.1,0.4] E[205]=e_{am}=[-0.2,0.3,0.0,0.1] E[502]=e_a=[0.5,-0.1,0.3,0.0]...$

We now get a matrix:

Xemb= [e_1, e_2, ..., e_n] ∈ ℝ^{n × d_model}

Step 2.2: Add Positional Encoding

Use learned or sinusoidal `PE ∈ ℝ^{n × d_model}`
Example (sinusoidal):

PE0=[0,1,0,1]PE1=[1,0,1,0]...

Final embeddings:

zt=et+PEtZ=[z1,z2,...,zn]

STEP 3: Forward Pass Through Transformer Decoder

Let's process input through one transformer decoder layer:

Step 3.1: Compute Q, K, V

Let:

$$W_Q, W_K, W_V \in \mathbb{R}^{d_model \times d_k}$$

Suppose:

$$W_Q = [[0.1, 0.2], [0.3, 0.4], [-0.1, 0.0], [0.2, -0.3]] W_K = samedims W_V = samedims$$

Then for token `z₁` (shape `[1 × d_model]`):

$$Q_1 = z_1 \times W_Q \in \mathbb{R}^{1 \times d_k} K_1 = z_1 \times W_K V_1 = z_1 \times W_V$$

Do this for all tokens:

$$Q = Z \times W_Q K = Z \times W_K V = Z \times W_V$$

Step 3.2: Scaled Dot-Product Attention

For each token `t`, compute:

$$attention_scores = Q_t \times K^T / \sqrt{d_k}$$

Then apply **mask** so position `t` can't attend to `t+1, ..., n`

Use:

$$A_masked = softmax((QK^T + M)/\sqrt{d_k}) \times V$$

Let's compute for `t = 3`:

$$Q_3 = [0.1, 0.2] K = [[0.2, 0.3], [0.0, 0.1], [-0.1, 0.2]] Q_3 \times K^T = [0.1 \times 0.2 + 0.2 \times 0.3, ..., 0.1 \times -0.1 + 0.2 \times 0.2] = [0.08, 0.02, 0.03]$$

Apply softmax to get attention weights:

$$\alpha = softmax([0.08, 0.02, 0.03]) = [0.34, 0.33, 0.33]$$

3. Pass to Transformer Decoder (Autoregressive)

Each step:

P(y1)=softmax(GPT(x1,...,xn))y1←sample(P(y1))→"म"P(y2)=softmax(GPT(x1,...,xn,y1))→"ए"...

The model generates:

"म एउटा केटा हुँ।"

4. Output Sent Back to User

Browser/app gets back:

```
{
  "response": "म एउटा केटा हुँ।"
}
```

How ChatGPT Maintains Chat History

For multi-turn conversations, it keeps **entire past conversation** in the input prompt:

```
User: Hello
Assistant: Hi! How can I help?

User: Translate "I am a boy" to Nepali
Assistant: म एउटा केटा हुँ।
```

The whole string is re-encoded and sent to the model **every time**.

That's why ChatGPT is memoryless across sessions — but within a session, it uses **long-context transformers** (GPT-4 can do up to 128k tokens).

ChatGPT is a Decoder-Only Transformer

It does **not** have an encoder like the original Transformer used for translation tasks (e.g., English → Nepali with encoder-decoder).

So When Is the Encoder Used?

The **encoder** is used in:

- **Encoder-decoder models** (e.g., original Transformer, T5, BERT2GPT)
- **Translation tasks**, where:
  - Encoder processes **source sentence** (e.g., `"I am a boy"`)
  - Decoder generates **target sentence** (e.g., `"म एउटा केटा हुँ"`), using both:

- Its own previous outputs (via self-attention)
- The encoder’s output (via cross-attention)