How real Transformers — like GPTs, BERTs — work internally:

Attention masks are **essential** for controlling **what a token can "see"** during training and inference.

🎯 What We'll Cover:

- ✅ Why Attention Masks are needed
- ✅ How Attention Masks work
- ✅ Real code example to visualize masking
- ✅ What happens if you *don't* use them

🧠 Why Attention Masks?

In Transformers: - **Each token "attends" to others** — meaning, it can "look at" other tokens to understand context.

But:

Scenario	Masking Needed?	Why?
Encoder (BERT, Translation Encoder)	❌ No	Each token can look at **all** others (bidirectional).
Decoder (GPT, Translation Decoder)	✅ Yes	**Auto-regressive**: token *can't* look at future tokens. Only past and present tokens.

✅ **In GPTs**, the model **MUST NOT peek** into the future when predicting the next token.

Thus, we need a **causal mask** or **attention mask**!

🔥 How Attention Mask Works Visually

Suppose you have this sequence:

`A B C D E`

You want token at position:

Token	Allowed to see
A	A
B	A, B
C	A, B, C
D	A, B, C, D
E	A, B, C, D, E

So you block (mask) future tokens during attention calculations.

✅ Mask is a **matrix** where: - `0` means allowed - `-inf` (very negative number) means forbidden

---

📋 Example: Mask Matrix for 5 tokens

	A	B	C	D	E
A	0	-inf	-inf	-inf	-inf
B	0	0	-inf	-inf	-inf
C	0	0	0	-inf	-inf
D	0	0	0	0	-inf
E	0	0	0	0	0

✅ This is called a **causal mask**.

🛠️ Simple Real PyTorch Code to Create a Causal Mask

```
import torch

def create_causal_mask(seq_len):
    """
    Creates a causal mask (lower triangular matrix) for a sequence of length seq_len
    """
    mask = torch.triu(torch.ones(seq_len, seq_len), diagonal=1)
    mask = mask.masked_fill(mask == 1, float('-inf'))
    mask = mask.masked_fill(mask == 0, float(0.0))
    return mask

# Example usage
seq_len = 5
mask = create_causal_mask(seq_len)
print("\n🛡️ Causal Mask Matrix (5 tokens):\n")
print(mask)
```

✅ Output:

```
tensor([[ 0., -inf, -inf, -inf, -inf],
        [ 0.,   0., -inf, -inf, -inf],
        [ 0.,   0.,   0., -inf, -inf],
        [ 0.,   0.,   0.,   0., -inf],
        [ 0.,   0.,   0.,   0.,   0.]])
```

🚀 How This Mask is Used During Attention Calculation

When the model computes attention scores:

```
attention_scores = (Q @ K.T) / sqrt(dk)
attention_scores = attention_scores + mask
```

- **Q** = Query matrix
- **K** = Key matrix
- `sqrt(dk)` = scaling factor

✅ The mask ensures that future tokens get **-inf added** to their attention scores
→ **Softmax makes their probability 0**
→ **Token can't attend to the future.**

🔥 Mini Example With Random Attention Scores

```
# Dummy attention scores
attn_scores = torch.randn(5, 5)
print("\n🔢 Raw Attention Scores:\n", attn_scores)

# Apply mask
masked_scores = attn_scores + mask
print("\n🛡️ Masked Attention Scores (future blocked):\n", masked_scores)

# Softmax
attn_probs = torch.softmax(masked_scores, dim=-1)
print("\n🎯 Attention Probabilities (after masking):\n", attn_probs)
```

✅ You will see that probabilities for future tokens become ~0.

🎨 Simple Diagram of Masking (Visualization)

Imagine attention scores without masking:

```
Token A: looks at [A, B, C, D, E]
Token B: looks at [A, B, C, D, E]
Token C: looks at [A, B, C, D, E]
...
```

With Causal Masking:

```
Token A: looks at [A]
Token B: looks at [A, B]
Token C: looks at [A, B, C]
Token D: looks at [A, B, C, D]
Token E: looks at [A, B, C, D, E]
```

✅ **Only past and present are visible!**

📚 Quick Recap Table

Concept	Description
Attention Mask	Matrix controlling which tokens can attend to which others
Causal Mask	Ensures that tokens can only see their past (not future)
Effect on Softmax	Prevents leaking future info during training and generation
Used in	GPT, GPT-2, GPT-3, ChatGPT, Codex, etc.