Transformers: The Revolution in Deep Learning

#What Is a Transformer?

- The Transformer architecture excels at handling text data which is inherently sequential. They take a text sequence as input and produce another text sequence as output. eg. to translate an input English sentence to Spanish.
- Replaced RNNs and LSTMs for sequence modelling → Allowed parallel training and handled long-range dependencies much better.
- Key Idea: Use **Self-Attention** mechanisms to capture relationships between words in a sentence, regardless of their distance.
- In the sentence: "XYZ went to France in 2019 when there were no cases of COVID and there, he met the president of that country" the word "that country" refers to "France".
  However, RNN would struggle to link "that country" to "France" since it processes each word in sequence leading to losing context over long sentences. This limitation prevents RNNs from understanding the full meaning of the sentence.

#Transformer Architecture Overview:

Component	Purpose
Input Embedding	Converts input tokens into dense numerical vectors (size = embedding_dim).
Positional Encoding	Adds sequence order information since transformers process all tokens simultaneously.
Encoder Block (Multiple Layers)	Processes input embeddings + applies self-attention + feed-forward layers.
Decoder Block (Multiple Layers)	Generates output sequence one token at a time, attends to encoder output.
Output Layer	Converts hidden representations to prediction (e.g., next word, classification label).

#Workflow Summary:

1. Input text → Tokenized → Embedded (plus positional encodings).
2. Self-Attention calculates how much each word should attend to every other word.
3. Passes through feed-forward layers → Intermediate representation.
4. Final prediction is generated (next word, classification label, etc.).

#Attention Mechanism:

1. **Intuition:** Measures “importance” of other words in the sequence when encoding a particular word.

2. **Scaled Dot-Product Attention Formula:**
   Given: Query (Q), Key (K), and Value (V) matrices.
   1. Calculate Attention Scores:
   2. $\text{Attention}(Q, K, V) = \text{softmax}((Q \cdot K^T) / \sqrt{d\_k}) \cdot V$

Where:

- $Q \cdot K^T$ → Measures similarity between query and key vectors.
- $\sqrt{d\_k}$ → Scale factor (dimension of key) → Prevents large values leading to small gradients.

3. Softmax Normalization → Produces attention weights.
4. Multiply by V (Values) → Weighted sum.

Example Intuition:
Sentence: “The cat sat on the mat.”

- To understand “sat”, the model calculates how much attention to give “cat” vs “mat” vs “The”.
- Self-attention enables capturing long-range dependencies → “sat” ↔ “cat” relationship important.

#Multi-Head Attention

Why Multiple Heads?

- Single attention head can only focus on one aspect of relationship.
- Multi-head allows the model to jointly attend to information from different representation subspaces.

#Process:
Split Q, K, V into multiple heads → Apply attention separately → Concatenate results → Linear layer → Richer representation.

#**Positional Encoding**

Why Needed?

- Transformers process tokens in parallel → No inherent sequence order.

#Positional Encoding Formula:

For position pos and dimension i:

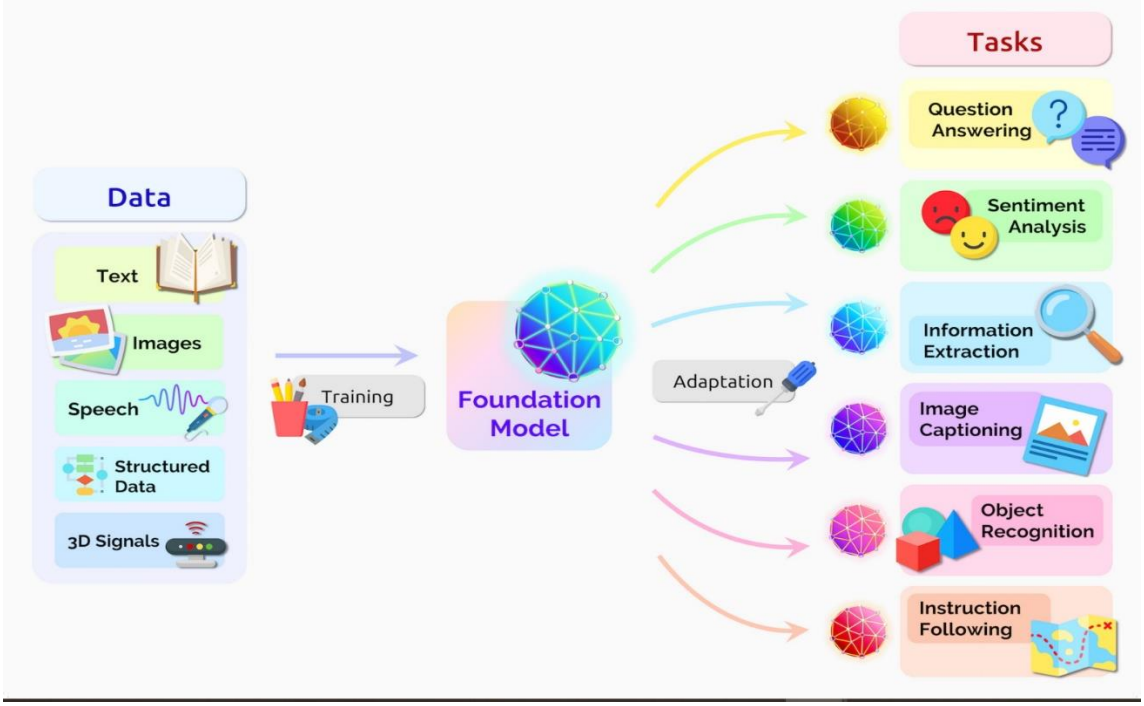
PE(pos, 2i) = sin(pos / 10000^(2i / embedding_dim))

PE(pos, 2i+1) = cos(pos / 10000^(2i / embedding_dim))

→ Allows model to distinguish word order.

#Transformer Example (Using Hugging Face):

```
from transformers import AutoTokenizer, AutoModel

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModel.from_pretrained('bert-base-uncased')

sentence = "Transformers are amazing for NLP."

# Tokenize + Convert to Tensor
inputs = tokenizer(sentence, return_tensors='pt')

# Pass through model
outputs = model(**inputs)

print(outputs.last_hidden_state.shape)  # (batch_size, sequence_length, hidden_dim)
```

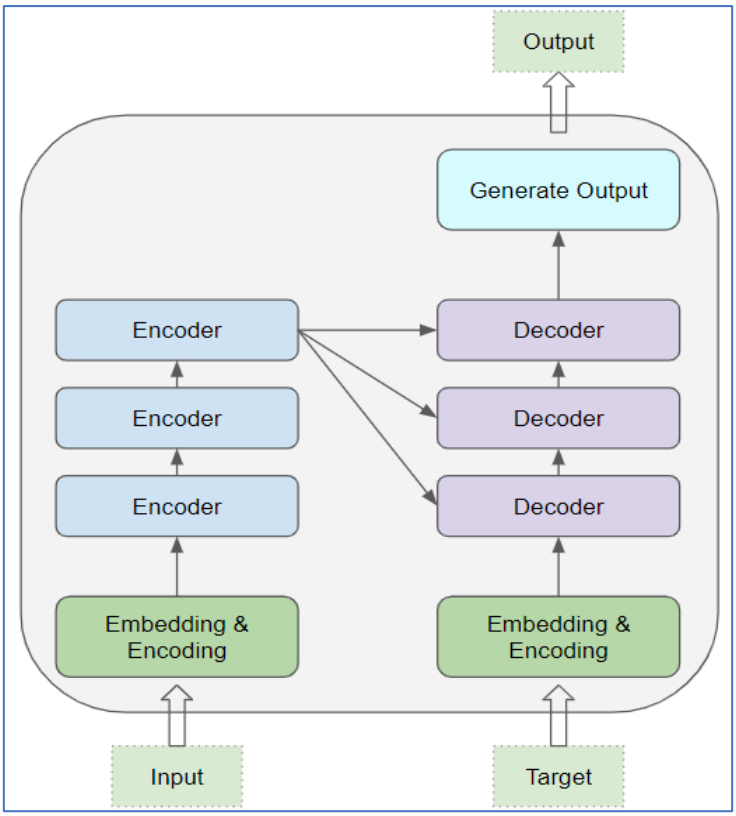


#Why Transformers Are Foundation of Modern Gen AI & RAG:

Concept	**Importance**
Self-Attention	Learns semantic relationships → Better than fixed BoW vectors.
Multi-Head Attention	Allows richer representations from multiple "views" of the sequence.
Pre-training + Fine-tuning	Powerful transfer learning → Trained once on massive corpus, fine-tuned for tasks (classification, QA, summarization).
Embeddings	Contextual embeddings capture word meaning in context → Dynamic vs. Static embeddings.
Parallel Processing	Efficient training → GPUs, large datasets.

#Real-World Use Cases:

Application	**Example**
Text Classification	Spam detection, sentiment analysis.
Question Answering (RAG)	Given documents → Find best answer span + Generate response.
Text Summarization	News article summarization.
Chatbots	Generative conversational agents.
Machine Translation	Google Translate (Sentence-level translation).



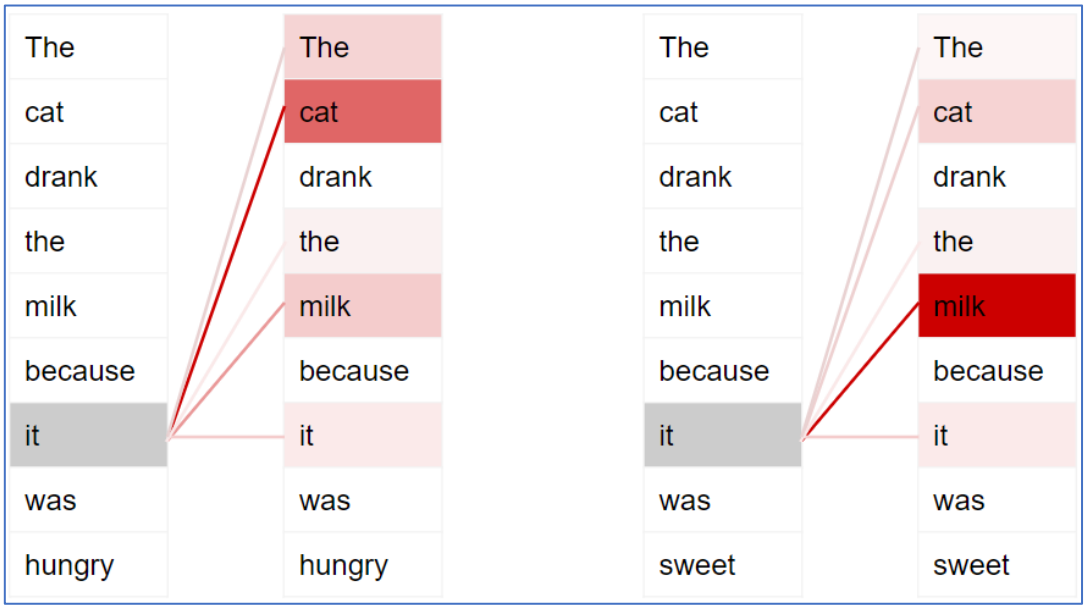
Example:

The Transformer architecture uses self-attention by relating every word in the input sequence to every other word.

eg. Consider two sentences:

- The *cat* drank the milk because **it** was hungry.
- The cat drank the *milk* because **it** was sweet.

In the first sentence, the word 'it' refers to 'cat', while in the second it refers to 'milk. When the model processes the word 'it', self-attention gives the model more information about its meaning so that it can associate 'it' with the correct word.



To enable it to handle more nuances about the intent and semantics of the sentence, Transformers include multiple attention scores for each word.

ex. While processing the word ‘it’, the first score highlights ‘cat’, while the second score highlights ‘hungry’. So, when it decodes the word ‘it’, by translating it into a different language, for instance, it will incorporate some aspect of both ‘cat’ and ‘hungry’ into the translated word.

#Encoder-Decoder Structure

Encoder:

- Purpose:
  Processes the input sequence (source language) → Outputs a contextualized representation of each word (embedding + attention).
- Structure:
  - Multiple identical layers (e.g., 6 layers).
  - Each layer consists of:
    1. **Self-Attention Mechanism** → Allows each word to attend to all other words in the input sentence.
    2. **Feed-Forward Neural Network** → Non-linear transformation of attention output.
    3. **Add & Norm Layers** → Residual connections + Layer normalization for stability.
- Output:
  - Contextualized embeddings of input tokens → Capture relationships between words.

Decoder

- Purpose:
  Takes encoder output + previously generated target tokens → Predicts next word in target sequence (output language).
- Structure:
  - Also composed of multiple identical layers (e.g., 6 layers).
  - Each layer consists of:
    1. **Masked Self-Attention** → Ensures that each token only attends to previous tokens (prevents “cheating” during training).

2. **Encoder-Decoder Attention** → Attention over encoder outputs → Injects source context.
3. **Feed-Forward Neural Network**
4. **Add & Norm Layers**

- Final Layer:
  - Linear + Softmax layer → Predicts probability distribution over vocabulary.

#Step-by-Step Flow:

Step 1 – Input Sentence (Source Language)
Example: "ich bin piyush"

Step 2 – Tokenization + Embedding

- Convert each word → Token IDs → Word Embeddings + Positional Encoding.

Step 3 – Encoder Process

- Each encoder layer applies self-attention → Outputs context-rich embedding for each word.

Step 4 – Decoder Process

- Initially feeds start-of-sequence token → Generates first word ("I").
- Masked Self-Attention → Ensures model doesn't look at future words.
- Uses encoder-decoder attention to get context from source sentence representation.

Step 5 – Iterative Prediction

- Next word generated based on previous words + encoder context → ("I" → "am" → "Piyush").
- Process continues until end-of-sequence token is predicted.

#Why Multiple Layers?

- Each encoder and decoder block refines representation →
  Deeper layers capture more abstract patterns (syntax, semantics).
- Stacking (e.g., 6 encoder + 6 decoder layers) → Model captures high-level context.

#Parallelism Benefit

- Unlike RNNs, transformers process all words in parallel during training →
  Speeds up training significantly and enables learning long-range dependencies without vanishing gradients.

#Single Encoder-Decoder Structure:

- In a standard Transformer model for sequence-to-sequence tasks (like translation), there is usually **one large Encoder stack** and **one large Decoder stack** (e.g., 6 layers each in vanilla Transformer).
- These are not multiple independent encoder-decoder models voting — they work as a **single end-to-end model**.

How the Output Is Generated:

Example Task: Language Translation (German → English)

Input Sentence (Source Language):
"Ich bin Piyush"
Expected Output Sentence (Target Language):
"I am Piyush"

Step 1 – Tokenization + Input Representation

- The input sentence is first tokenized → Tokens:
- ["Ich", "bin", "Piyush"]
- Each token is converted into a fixed-size dense vector (embedding), plus positional encoding is added to capture word order.

Step 2 – Encoder Process

- The **encoder** consists of multiple identical layers (e.g., 6 layers).
- At each encoder layer:
  - Applies **self-attention**: Each word attends to every other word in the input sequence.
  - Generates contextualized embeddings → Each embedding now represents the word in context.

Example:

- "Ich" embedding captures the meaning of "Ich" in the context of "Ich bin Piyush" (not just the word "Ich" alone).

The final encoder output is a matrix of shape (sequence_length, embedding_dim):

[
Encoder_output_1 ("Ich"),
Encoder_output_2 ("bin"),
Encoder_output_3 ("Piyush")
]

Step 3 – Decoder Process

- Begins with special start token (<SOS>).
- Step by step generation:

First Output Token (t=1):

- Input: <SOS>
- Decoder performs:
  1. Masked self-attention → Prevents seeing future tokens.
  2. Encoder-decoder attention → Uses encoder outputs as keys/values.
  3. Outputs probabilities over target vocabulary →
     Example Output:
  4. {'I': 0.85, 'You': 0.1, 'We': 0.05}
  5. The word "I" is chosen as the most probable.

Second Output Token (t=2):

- Input: <SOS>, "I"
- Repeat process:
  - Encoder-decoder attention → Context: "Ich bin Pioche"
  - Predict next word probabilities:
  - {'am': 0.9, 'is': 0.05, 'are': 0.05}
  - "am" is selected.

Third Output Token (t=3):

- Input: <SOS>, "I", "am"
- Predict next word probabilities:
- {'Piyush': 0.95, 'happy': 0.02, 'student': 0.03}
- "Piyush" is selected.

Final Step (t=4):

- Predict end-of-sequence (<EOS>) token → Stop generation.

#Fine-Tuning Transformers: Step-by-Step Process

#What Is Fine-Tuning?

- Fine-tuning means taking a pre-trained transformer (e.g., BERT, GPT) and adjusting its weights on a specific dataset/task.
- Pre-trained models are trained on huge generic datasets → Capture general language understanding.
- Fine-tuning specializes the model for tasks like sentiment analysis, classification, or question answering.

#Step-by-Step Workflow:

Step 1 – Load Pre-Trained Model + Tokenizer

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained('bert-base-uncased')
model = AutoModelForSequenceClassification.from_pretrained('bert-base-uncased', num_labels=2)  # For binary classification
```

Step 2 – Prepare Dataset

```
#Input data → Tokenized into input IDs and attention masks.

encoded_inputs = tokenizer(["I love this movie", "This movie is bad"], padding=True, truncation=True, return_tensors='pt')
labels = torch.tensor([1, 0])  # 1 = Positive, 0 = Negative
```

Step 3 Fine-Tuning Process

```
from transformers import AdamW

optimizer = AdamW(model.parameters(), lr=2e-5)

# Forward Pass
outputs = model(**encoded_inputs, labels=labels)
loss = outputs.loss

# Backpropagation
loss.backward()
optimizer.step()
```

- Typically done for multiple epochs over batches of data.

Step 4 – Evaluation

```
with torch.no_grad():
   predictions = model(**encoded_inputs).logits.argmax(dim=-1)

accuracy = (predictions == labels).float().mean()
print(f"Fine-Tuned Model Accuracy: {accuracy}")
```

#Notes:

- Fine-tuning adjusts the entire model or just the classification head (depending on use case).
- Requires GPU for efficiency.
- Pre-trained transformers already learned language semantics → Fine-tuning needs much less data than training from scratch.