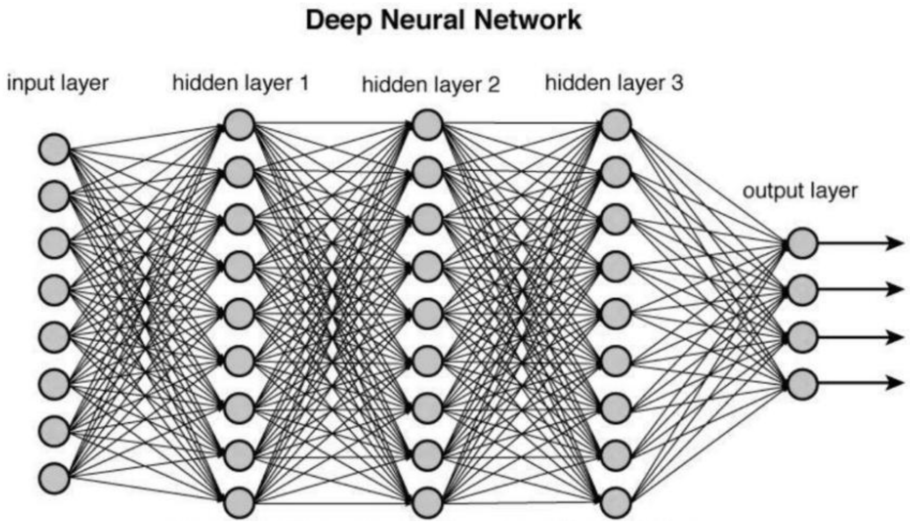
307307
Part 3 – Introduction to Deep Learning and Large Language Models

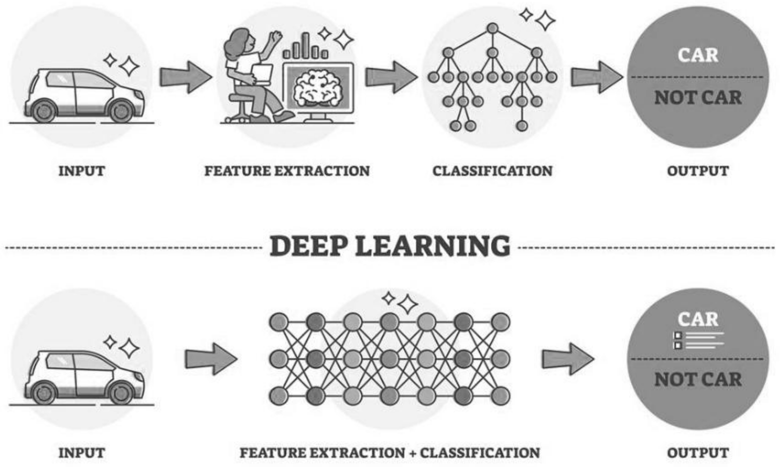
Content

- Introduction to Deep Neural Networks
    - CNNs
    - RNNs
    - The Transformer
- Contextual Word Embeddings
    - Introduction BERT
    - Introduction to HuggingFace and the Transformers Library

Introduction to Deep Learning

- Neural Networks have revolutionized artificial intelligence by enabling machines to learn from data in ways that mimic human neural processes.
- Deep neural networks (DNNs) are Neural Networks that are composed of multiple processing layers that can learn representations of data with multiple levels of abstraction.
- The power of deep learning comes from its ability to automatically discover intricate patterns in raw data through the learning process, without requiring human engineers to manually specify all the knowledge needed by the computer system.





Introduction to Deep Learning

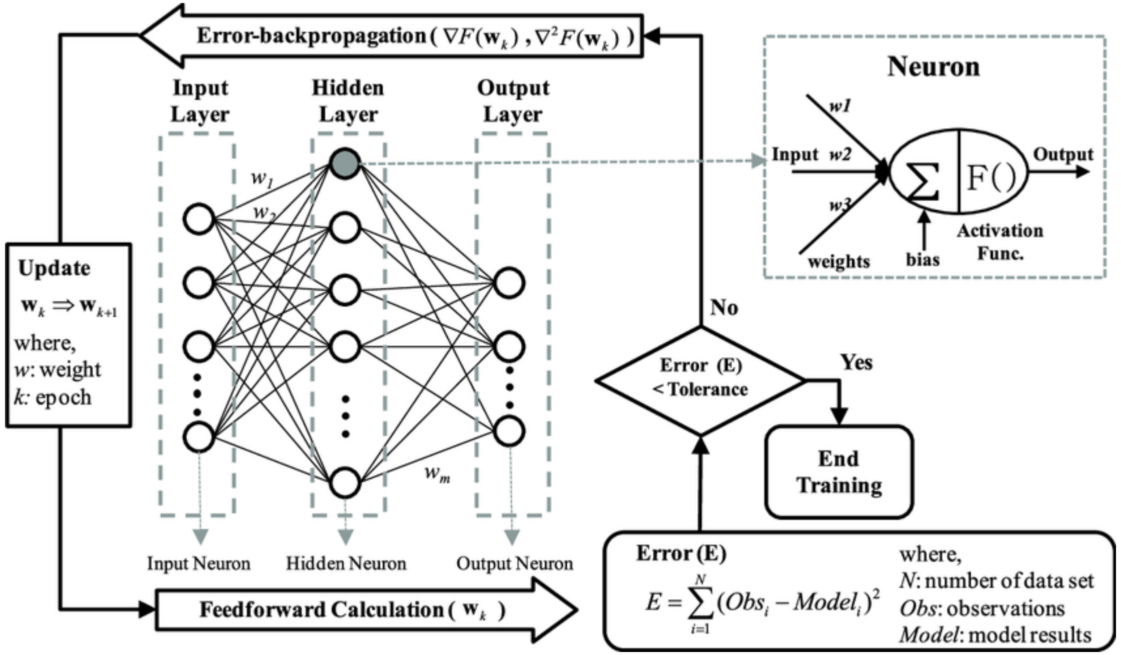
Fundamentals of Neural Networks

At their core, neural networks consist of:

1. **Neurons**: Mathematical functions that take inputs, apply weights, add a bias, and produce an output
2. **Layers**: Collections of neurons that process information in stages
3. **Activation Functions**: Non-linear functions that introduce complexity into the network
4. **Weights and Biases**: Parameters that are adjusted during training

The basic workflow involves:

- Forward propagation: Data flows through the network
- Loss calculation: The network's prediction is compared to the actual value
- Backpropagation: Errors are propagated backward to update weights
- Optimization: Weights are adjusted to minimize errors



Convolutional Neural Networks

CNNs revolutionized image processing by introducing specialized layers that mimic how the visual cortex processes information.

Key components include:

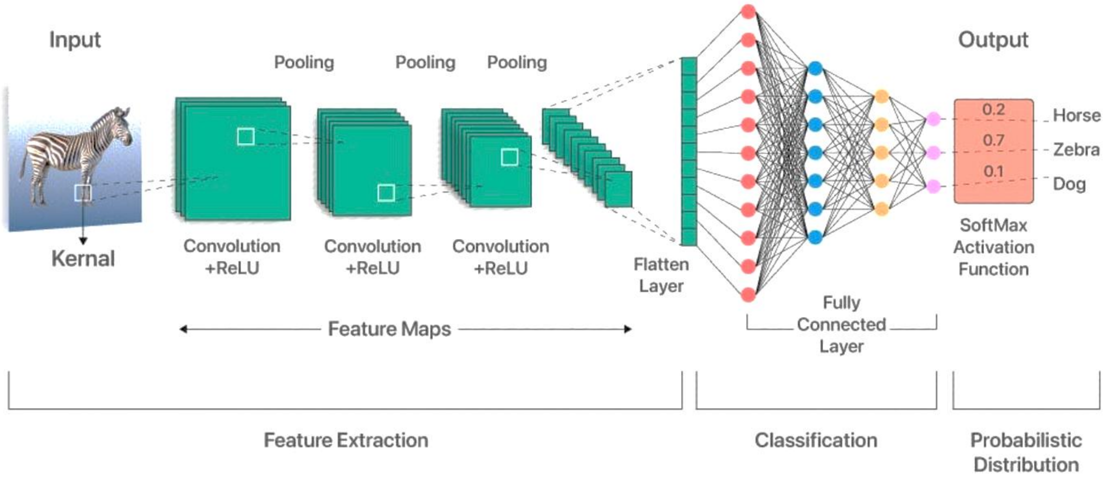
1. **Convolutional Layers**: Apply filters that scan across the input data to detect patterns
2. **Pooling Layers**: Reduce dimensions while preserving important features
3. **Fully Connected Layers**: Connect every neuron to every neuron in adjacent layers

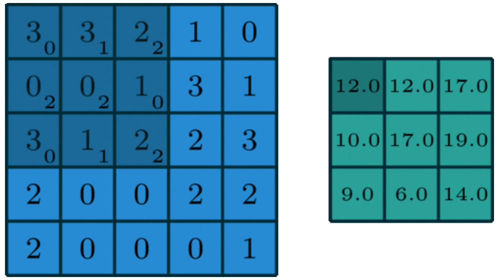
Instead of each neuron connecting to every pixel in an image (which would be computationally expensive), CNNs use:

- **Local connectivity**: Neurons connect only to nearby pixels
- **Parameter sharing**: The same filter is applied across the entire image

Business applications include:

- Product image recognition
- Visual quality control in manufacturing
- Document processing
- Customer behavior analysis in retail





3_0	3_1	2_2	1	0
0_2	0_2	1_0	3	1
3_0	1_1	2_2	2	3
2	0	0	2	2
2	0	0	0	1

12.0	12.0	17.0
10.0	17.0	19.0
9.0	6.0	14.0

Recurrent Neural Networks (RNNs)

Unlike traditional neural networks, RNNs process sequences by maintaining a form of memory of previous inputs.

Key characteristics:

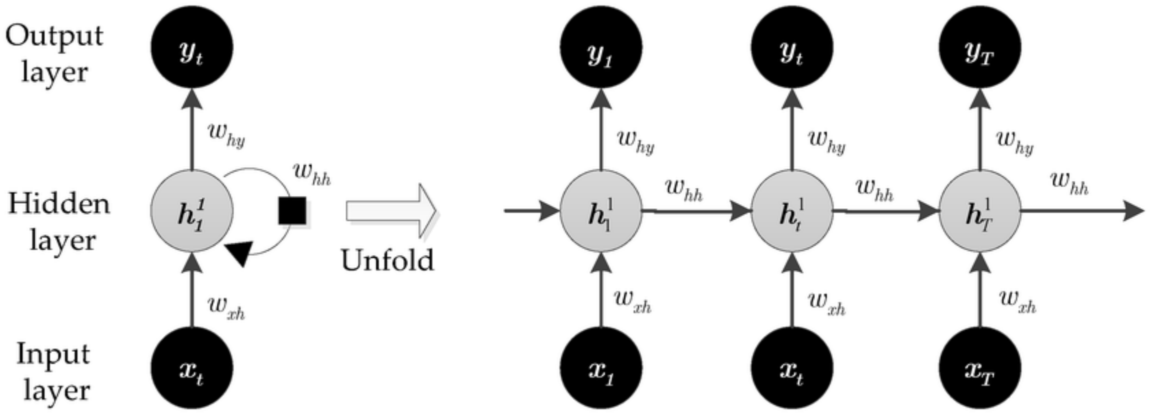
- **Time-dependent processing**: Output depends on both current and previous inputs
- **Shared parameters**: The same weights are applied at each time step
- **Memory**: Internal state acts as a form of short-term memory

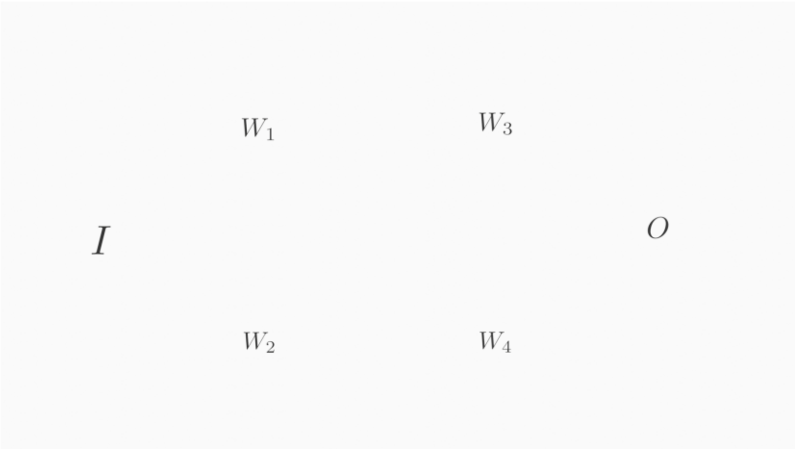
However, basic RNNs struggle with long-term dependencies due to:

- **Vanishing gradient problem**: The influence of early inputs fades over time
- **Exploding gradient problem**: Gradients grow uncontrollably during training

Business applications include:

- Language Modeling & Text Generation – Predicting the next word in a sequence (e.g., autocomplete, chatbots).
- Machine Translation – Translating text from one language to another.
- Speech Recognition – Converting spoken language into written text.
- Stock Price Prediction – Predicting future stock or financial data.
- Weather Forecasting – Modeling temporal patterns in weather data.
- Patient Monitoring – Analyzing sequences of medical data (e.g., ECG signals).
- Music Generation – Creating sequences of musical notes.
- Fraud Detection – Detecting unusual sequences in financial transactions.
- Network Intrusion Detection – Monitoring patterns of activity over time.





Transformers

Transformers – Architecture and Principles

What is a Transformer?

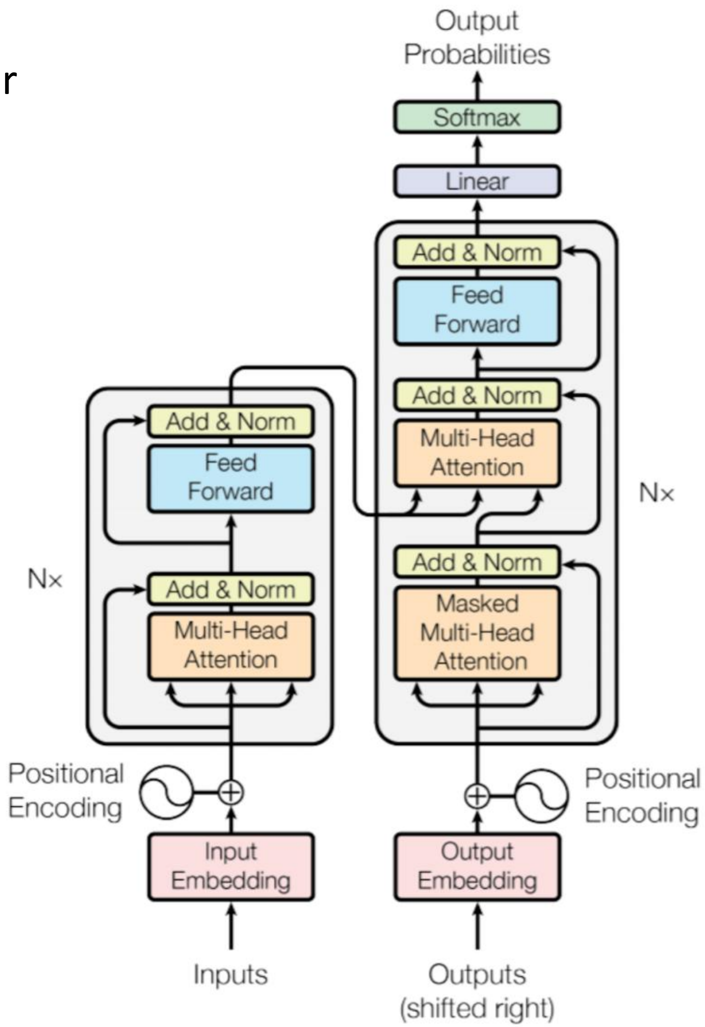
- A deep learning model based entirely on **self-attention**, with no recurrence or convolutions
- Introduced in the paper *"Attention Is All You Need"* (Vaswani et al., 2017)

Transformer Components:

1. **Embeddings**: Convert tokens to vector representations
2. **Positional Encoding**: Adds position information
3. **Multi-Head Attention**: Processes relationships from multiple perspectives
4. **Feed-Forward Networks**: Process each position independently
5. **Layer Normalization**: Stabilizes training
6. **Residual Connections**: Helps with gradient flow

Architecture Variations:

- **Encoder-only** (BERT): Good for understanding (classification, NER)
- **Decoder-only** (GPT): Good for generation
- **Encoder-decoder** (T5): Good for transformation tasks (translation, summarization)



The Transformer in Detail

The transformer architecture is designed to process sequential data, such as natural language, in a highly efficient and effective manner. Unlike traditional models that rely on sequential processing (like RNNs), transformers utilize a mechanism called self-attention, allowing them to analyze the entire input sequence at once. This capability enables them to capture complex relationships and dependencies between tokens in the sequence.

The transformer model consists of two main parts:

1. **Encoder:** The encoder processes the input sequence and generates a continuous representation of it. This representation captures the contextual information of the input tokens.
2. **Decoder:** The decoder takes the encoder's output and generates the final output sequence. It does this by predicting one token at a time, using the encoded representations and previously generated tokens.

- Both the encoder and decoder are composed of multiple identical layers—typically six layers in the original transformer architecture—allowing for deep learning and complex feature extraction.

Key Components of Transformers

1. **Multi-Head Attention:**

Function: Multi-head attention allows the model to focus on different parts of the input sequence simultaneously. It computes attention scores for each token in relation to all other tokens, enabling the model to weigh the importance of each token when making predictions.

Mechanism: The attention mechanism uses three vectors: Query (Q), Key (K), and Value (V). The attention scores are calculated as the dot product of the query and key vectors, scaled by the square root of the dimension of the key vectors. These scores are then used to weight the value vectors, producing a context-aware representation of the input.

2. Feed-Forward Networks:

Function: Each layer of the encoder and decoder contains a feed-forward neural network that processes the output from the attention mechanism. This network enhances the model's ability to learn complex representations.

Structure: The feed-forward network consists of two linear transformations with a non-linear activation function (usually ReLU) applied between them. This allows the model to capture intricate patterns in the data.

3. Positional Encoding:

Purpose: Since transformers do not inherently understand the order of tokens, positional encodings are added to the input embeddings. This encoding provides information about the position of each token within the sequence, allowing the model to consider the order of words.

Implementation: Positional encodings are typically generated using sine and cosine functions, which create unique encodings for each position that can be added to the input embeddings.

Key Components of Transformers

4. Layer Normalization:

Function: Layer normalization stabilizes the training process by normalizing the inputs to each layer. This helps mitigate issues related to internal covariate shift and improves convergence during training.

Application: It is applied after the attention and feed-forward layers, ensuring that the outputs are centered and scaled appropriately.

5. Residual Connections:

Purpose: Residual connections help facilitate the flow of gradients during training, addressing the vanishing gradient problem. They allow the model to learn more effectively by providing a direct path for gradients to flow through the network.

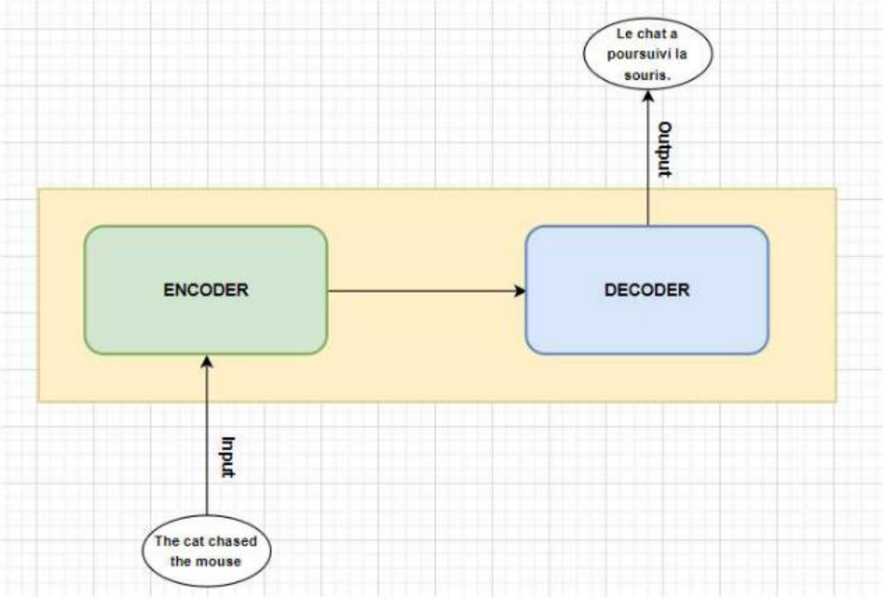
Implementation: The output of each sub-layer (attention and feed-forward) is added back to the original input, creating a shortcut that enhances learning.

The transformer architecture is a powerful and flexible model that has transformed the landscape of natural language processing and other fields. Its ability to process entire sequences simultaneously, leverage self-attention mechanisms, and utilize deep learning through stacked layers makes it a robust choice for a wide range of tasks.

How Transformer Works

Let's understand how the transformer takes input, processes and gives output.
We will consider a simple example of translating an English sentence to French using a transformer model.
Suppose we have,

- **Input sentence:** "Your cat is a lovely cat”
  We want to translate this to French:
- **Output sentence:** "Ton chat est un chat adorable.“
  The transformer takes the input, translates, and gives the output.

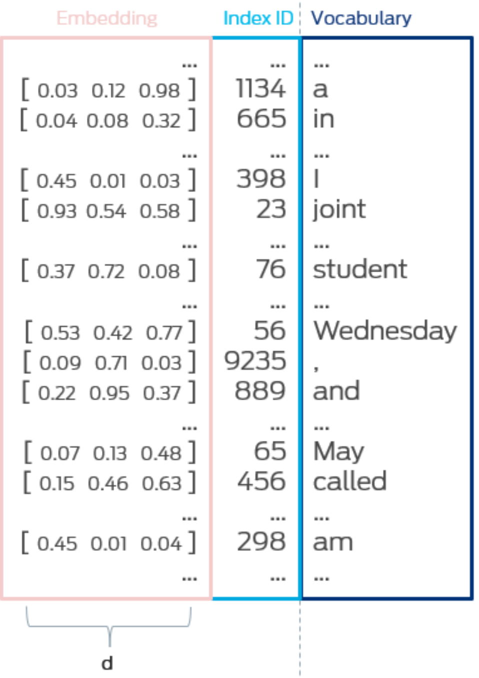


Input Preparation

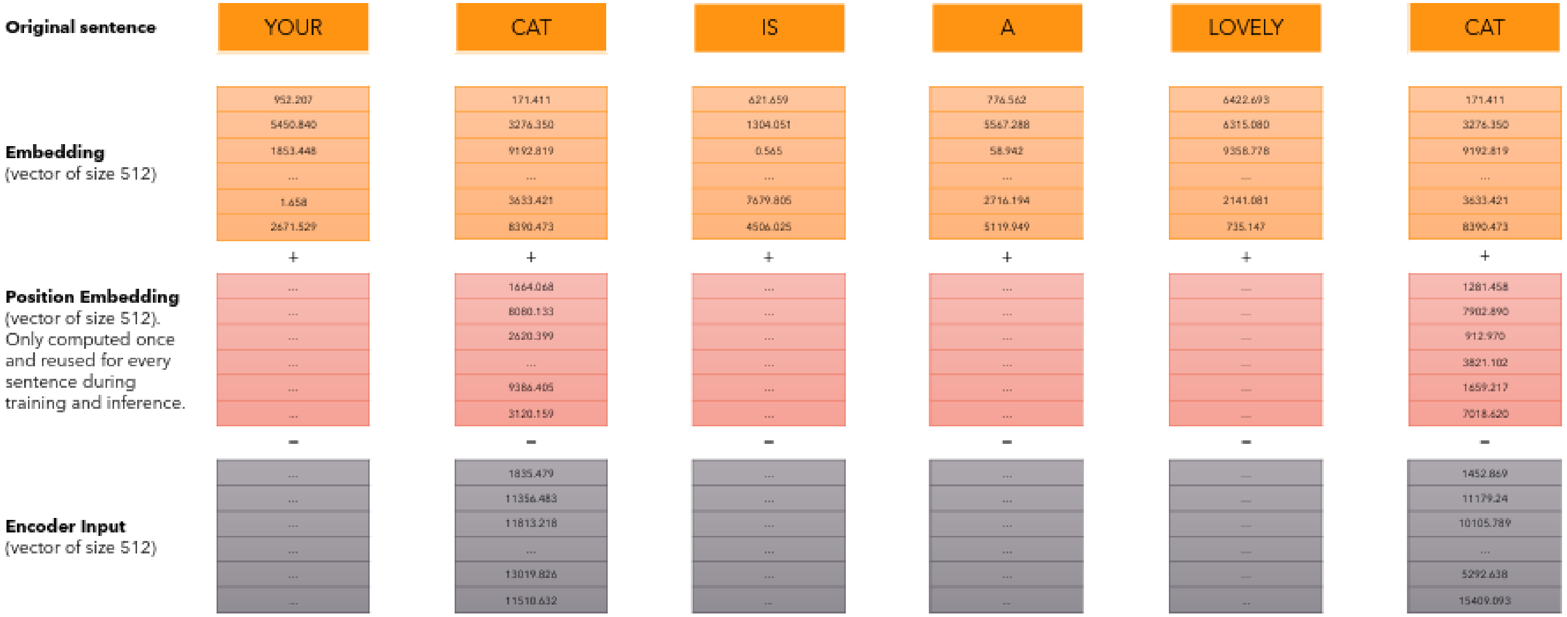
English sentence (source):

“Your cat is a lovely cat“

- Tokenization: Break the sentence into tokens: ["<s>", "Your", "cat", "is", "a", "lovely", "cat", "</s>"]
- Embedding: Convert each token into a vector using a learned embedding matrix. These embeddings capture the semantic meaning of each word.
- Positional Encoding: Add position-based information to each token embedding to provide information about the position of each token in the sequence. This is necessary because transformers process the entire sequence simultaneously, unlike RNNs that process one word at a time.
- Without positional encoding, the self-attention mechanism would treat the sentence as a bag of words.



Embedding	Index ID	Vocabulary
...	...	...
[0.03 0.12 0.98]	1134	a
[0.04 0.08 0.32]	665	in
...	...	...
[0.45 0.01 0.03]	398	I
[0.93 0.54 0.58]	23	joint
...	...	...
[0.37 0.72 0.08]	76	student
...	...	...
[0.53 0.42 0.77]	56	Wednesday
[0.09 0.71 0.03]	9235	,
[0.22 0.95 0.37]	889	and
...	...	...
[0.07 0.13 0.48]	65	May
[0.15 0.46 0.63]	456	called
...	...	...
[0.45 0.01 0.04]	298	am
...	...	...



Original sentence	YOUR	CAT	IS	A	LOVELY	CAT
Embedding (vector of size 512)	952.207 5450.840 1853.448 ... 1.658 2671.529	171.411 3276.350 9192.819 ... 3633.421 8390.473	621.659 1304.051 0.565 ... 7679.805 4506.025	776.562 5567.288 58.942 ... 2716.194 5119.949	6422.693 6315.080 9358.778 ... 2141.081 735.147	171.411 3276.350 9192.819 ... 3633.421 8390.473
	+	+	+	+	+	+
Position Embedding (vector of size 512). Only computed once and reused for every sentence during training and inference.	... ...	1664.068 8080.133 2620.399 ... 9386.405 3120.159	... ...	... ...	... ...	1281.458 7902.890 912.970 3821.102 1659.217 7018.620
	=	=	=	=	=	=
Encoder Input (vector of size 512)	... ...	1835.479 11356.483 11813.218 ... 13019.826 11510.632	... ...	... ...	... ...	1452.869 11179.24 10105.789 ... 5292.638 15409.093

Encoder

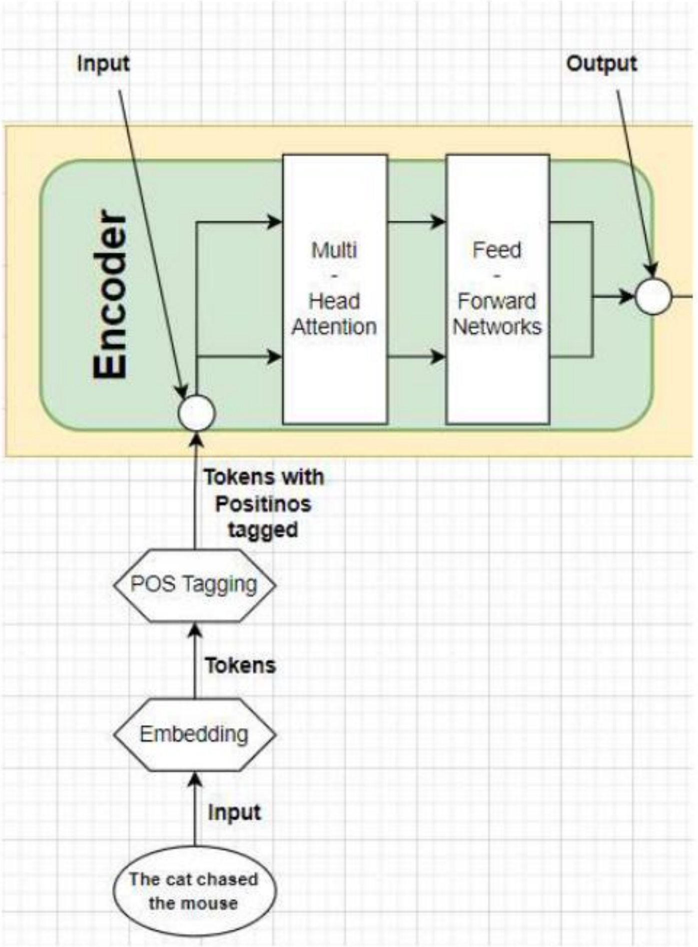
The encoder processes the full input sequence in parallel through a stack of layers.

The encoder takes the input embeddings with positional encodings and passes them through multiple layers of multi-head attention and feed-forward networks. Each encoder layer processes the input, allowing the model to learn complex representations of the sentence.

For example, in a sentence like “The cat chased the mouse”, the attention mechanism in the encoder might learn that "cat" is related to "chased" and "mouse", capturing the semantic relationships between the tokens.

Each encoder layer includes:

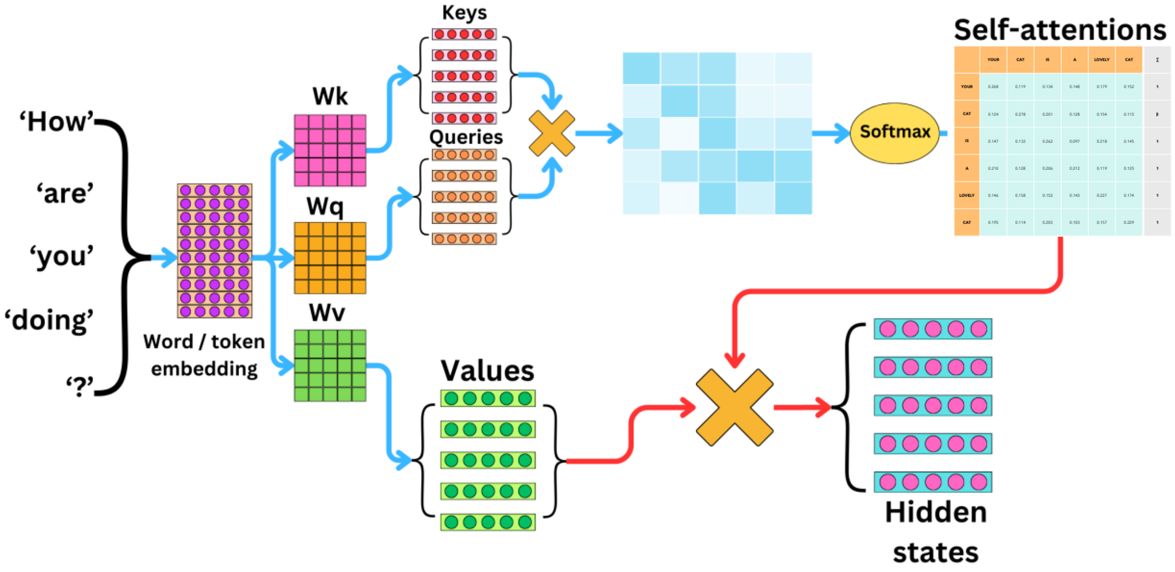
- **Multi-head Self-Attention:**
  Each word learns which other words to focus on.
  Example: The second occurrence of "cat" may attend to the first "cat" to recognize repetition or coreference.
- **Add & Norm:**
  A residual connection followed by layer normalization.
- **Feedforward Network:**
  A two-layer neural network processes each position independently.
- **Add & Norm:**
  Another residual connection and normalization.
- This stack is repeated several times (e.g., 6 layers), producing a set of **contextualized vectors**, one for each token.



The Attention Mechanism

1. **Create Q, K, V matrices**: Each word embedding is multiplied by three learned weight matrices (WQ, WK, WV) to create Query, Key, and Value representations:
   - Q = X · WQ
   - K = X · WK
   - V = X · WV
2. **Compute attention scores**: Each query vector is dot-producted with all key vectors to get raw attention scores:
   - Scores = Q · K^T
3. **Scale the scores**: Divide by √dk (where dk is the dimension of the key vectors) to prevent the values from getting too large:
   - Scaled scores = (Q · K^T) / √dk
4. **Apply softmax**: Convert the scaled scores to probabilities:
   - Attention weights = softmax(Scaled scores)
5. **Compute weighted values**: Multiply the attention weights by the value vectors:
   - Output = Attention weights · V

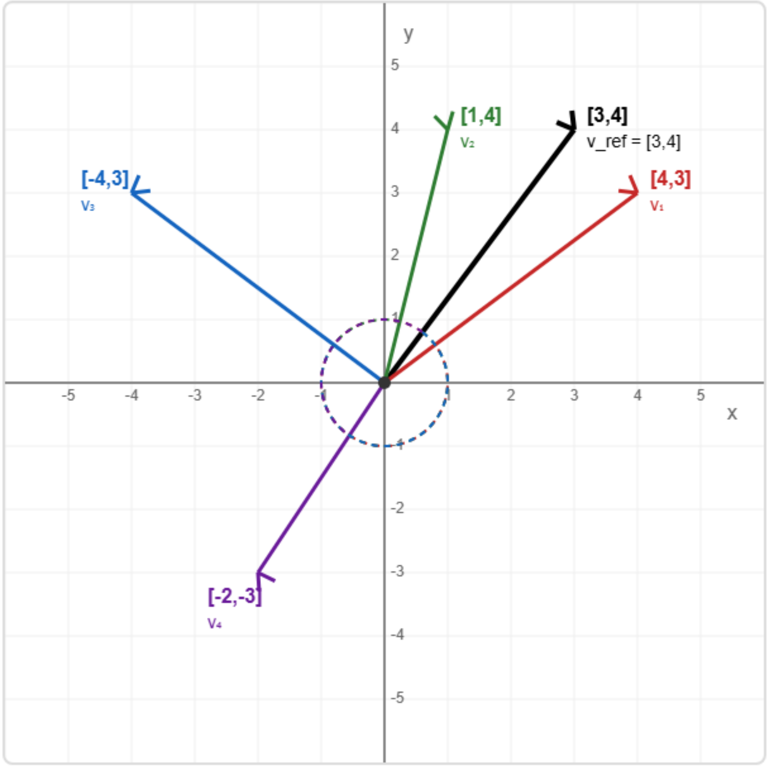
So the formula is: Attention(Q,K,V) = softmax((Q·K^T)/√dk) · V



	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

Vector Dot Product as Similarity Measure

$$v_1 \cdot v_2 = x_1x_2 + y_1y_2$$



1. Very Similar (High Positive)

`Red Vector v₁ = [4, 3]`

$v_1 \cdot v\_ref = (4\times3) + (3\times4) = 24$

Almost same direction as reference

2. Similar (Positive)

`Green Vector v₂ = [1, 4]`

$v_2 \cdot v\_ref = (1\times3) + (4\times4) = 19$

Generally same direction

3. Neutral (Zero)

`Blue Vector v₃ = [-4, 3]`

$v_3 \cdot v\_ref = (-4\times3) + (3\times4) = 0$

Perpendicular (90° angle)

4. Dissimilar (Negative)

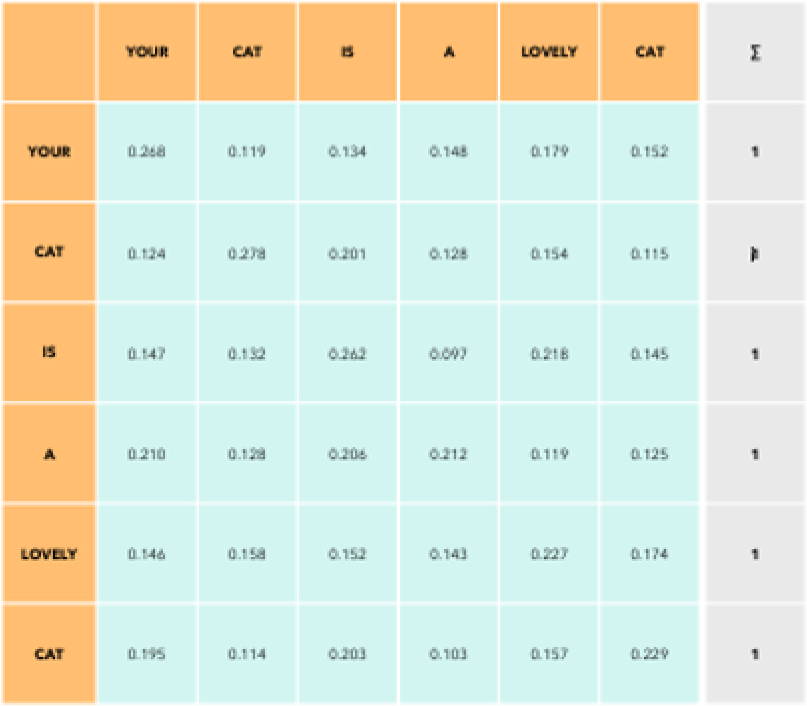
`Purple Vector v₄ = [-2, -3]`

$v_4 \cdot v\_ref = (-2\times3) + (-3\times4) = -18$

Opposite direction

Key Innovation: Self-Attention Mechanism

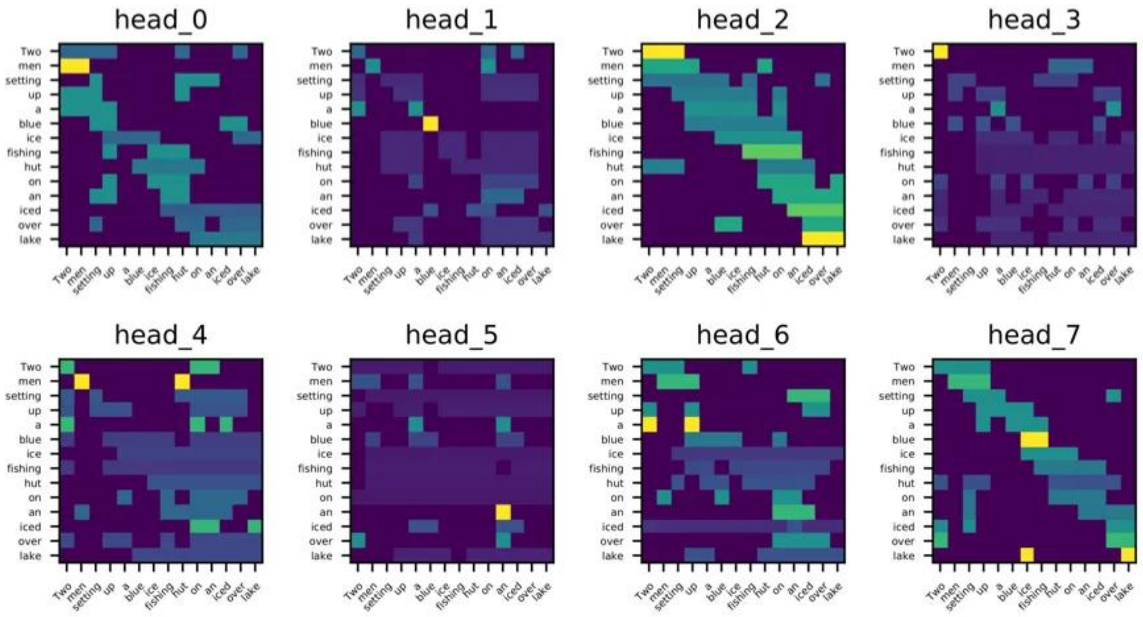
- Self-attention allows each word to compute its embedding by gathering information from all other words in the sequence.
- The "attention weights" determine how much each word should focus on other words.
- Words that are semantically related tend to have higher attention scores between them.
- This mechanism helps capture long-range dependencies and relationships regardless of word distance.
- Multiple attention heads in parallel (Multi-head Attention) allow the model to focus on different aspects of relationships.



	YOUR	CAT	IS	A	LOVELY	CAT	Σ
YOUR	0.268	0.119	0.134	0.148	0.179	0.152	1
CAT	0.124	0.278	0.201	0.128	0.154	0.115	1
IS	0.147	0.132	0.262	0.097	0.218	0.145	1
A	0.210	0.128	0.206	0.212	0.119	0.125	1
LOVELY	0.146	0.158	0.152	0.143	0.227	0.174	1
CAT	0.195	0.114	0.203	0.103	0.157	0.229	1

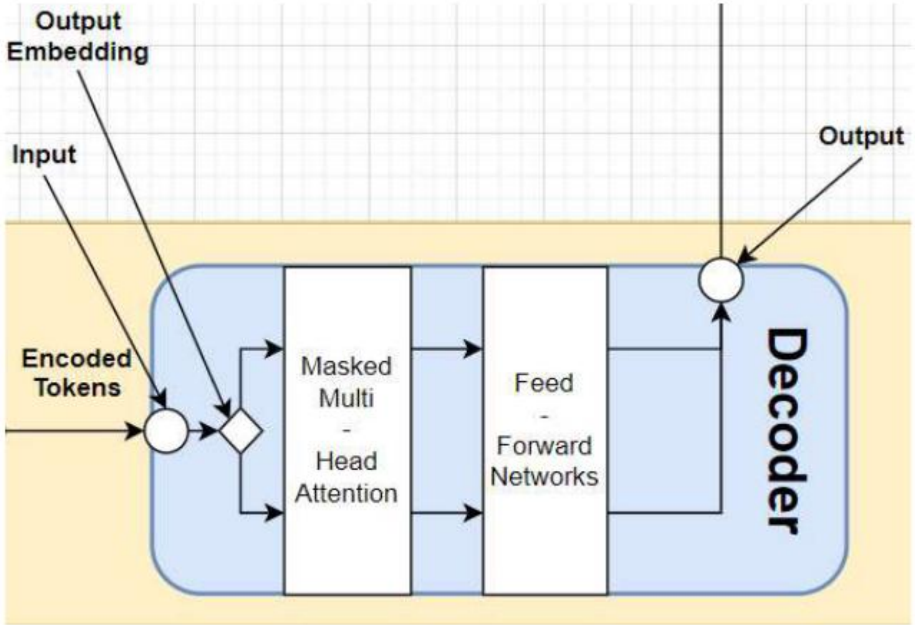
Multiple Attention Heads

- Multiple attention heads in parallel (Multi-head Attention) allow the model to focus on different aspects of relationships.
- For example, one head might learn which words are related by grammar, while another might focus on semantic meaning.
- This allows the model to capture a richer and more comprehensive understanding of the input.



Decoding Process

- The decoder takes the encoder's output and generates the output sequence in French.
- It uses masked multi-head attention to ensure that predictions for a given token do not depend on future tokens.
- This allows the decoder to generate the output one token at a time.
- The decoder also attends to the encoder's output, enabling it to incorporate context from the input sentence while generating the translation.
- For instance, the attention mechanism in the decoder might focus on the representation of "cat" when generating "Le chat", ensuring that the translation is consistent with the input.



Decoder Components

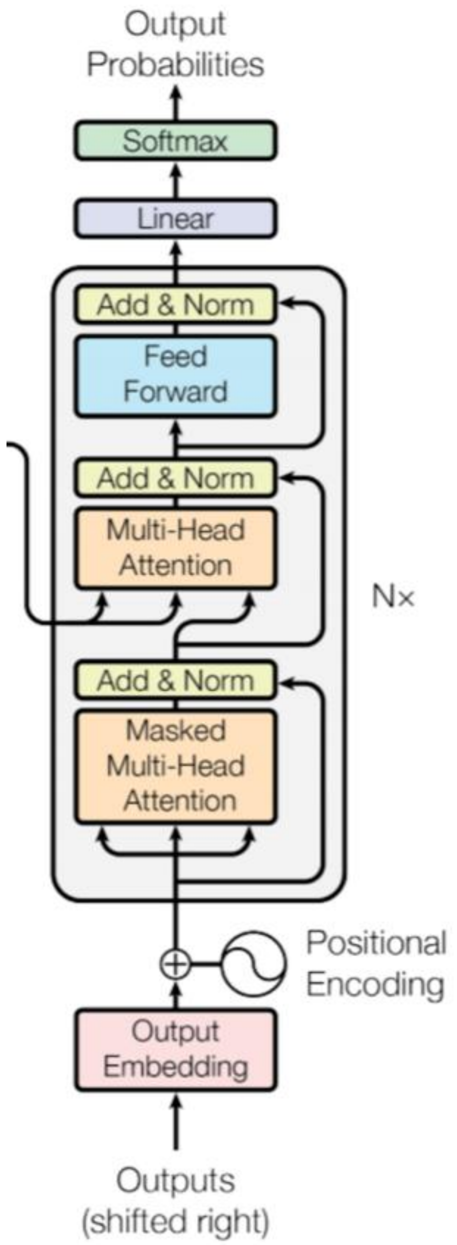
1. **Masked Multi-head Self-Attention:**
   Looks at the previously generated French words. Future words are masked to prevent cheating.

$$\overset{\text{Scaled Scores}}{\begin{bmatrix} 0.7 & 0.1 & 0.1 & 0.1 \\ 0.1 & 0.6 & 0.2 & 0.1 \\ 0.1 & 0.3 & 0.6 & 0.1 \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix}} + \overset{\text{Look-Ahead Mask}}{\begin{bmatrix} 0 & \text{-inf} & \text{-inf} & \text{-inf} \\ 0 & 0 & \text{-inf} & \text{-inf} \\ 0 & 0 & 0 & \text{-inf} \\ 0 & 0 & 0 & 0 \end{bmatrix}} = \overset{\text{Masked Scores}}{\begin{bmatrix} 0.7 & \text{-inf} & \text{-inf} & \text{-inf} \\ 0.1 & 0.6 & \text{-inf} & \text{-inf} \\ 0.1 & 0.3 & 0.6 & \text{-inf} \\ 0.1 & 0.3 & 0.3 & 0.3 \end{bmatrix}}$$

2. Encoder-Decoder Attention:
Each decoder token can attend to all encoder outputs.
Example: The decoder token "chat" may attend to the English "cat" to align the translation.

3. Feedforward Network:
Processes each token vector separately.

Also, each sub-layer includes residual connections and layer normalization.



Decoder Training and Inference

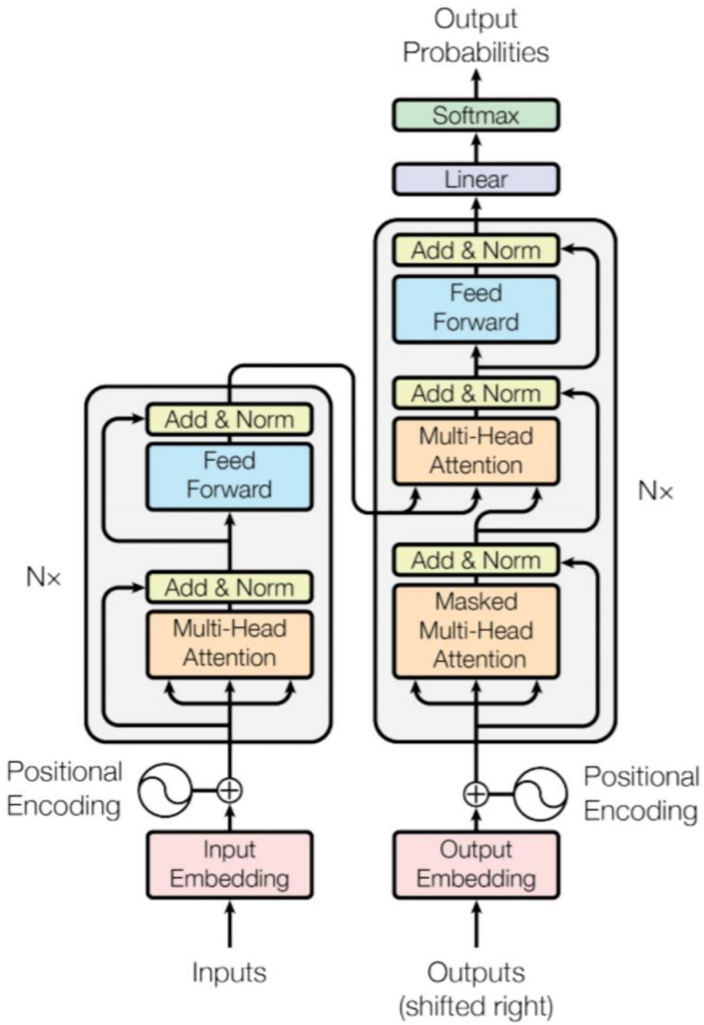
Training Phase:

- The decoder begins with a special start-of-sequence token <s>
- At each time step, it receives the actual previous target words.
- Example: Step 1: <s>, Step 2: <s> Ton, Step 3: <s> Ton chat, etc.

Inference Phase:

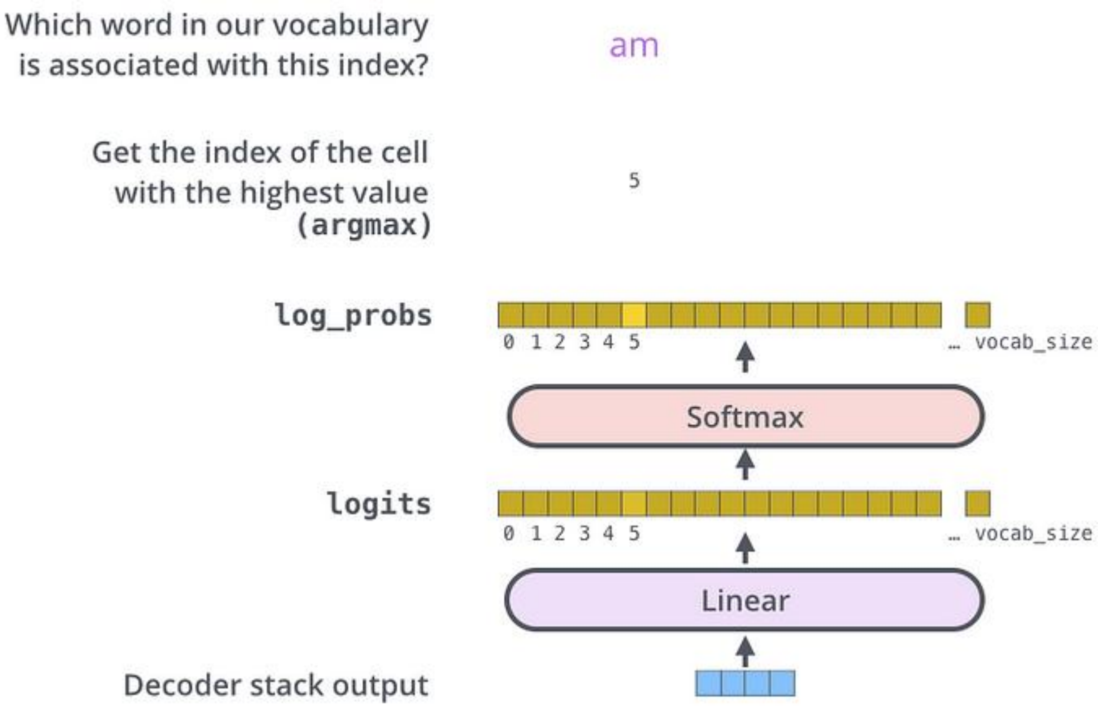
- Starts with <s> and generates one word at a time.
- Each new word is used as input for the next step.
- Example: <s> → Ton → chat → est → ...

Summary:
By combining attention to past outputs and the encoded input, the decoder generates coherent, context-aware text—essential for tasks like translation and summarization.



Output Generation

- The decoder outputs a vector at each time step.
- A linear layer followed by softmax turns this vector into a probability distribution over the French vocabulary.
- The model selects the most probable next word ("Ton", then "chat", then "est", etc.).
- This continues until an end-of-sentence token </s> is generated or a length limit is reached.



Which word in our vocabulary is associated with this index?
am
Get the index of the cell with the highest value (argmax)
5
log_probs
0 1 2 3 4 5
… vocab_size
Softmax
logits
0 1 2 3 4 5
… vocab_size
Linear
Decoder stack output

A Summary of how the Transformer Works

Input Sentence: "The cat chased the mouse."

Input Encoding:

- Break down the sentence into tokens (words).
- Convert each token into a numerical representation called an embedding.
- Add positional encodings to the embeddings to provide information about the position of each token.

Encoder Processing:

- The encoder takes the input embeddings with positional encodings.
- Pass the input through multiple layers of multi-head attention and feed-forward networks.
- Each encoder layer processes the input, allowing the model to learn complex representations of the sentence.
- The attention mechanism in the encoder learns relationships between tokens (e.g., "cat" is related to "chased" and "mouse").

Decoder Processing:

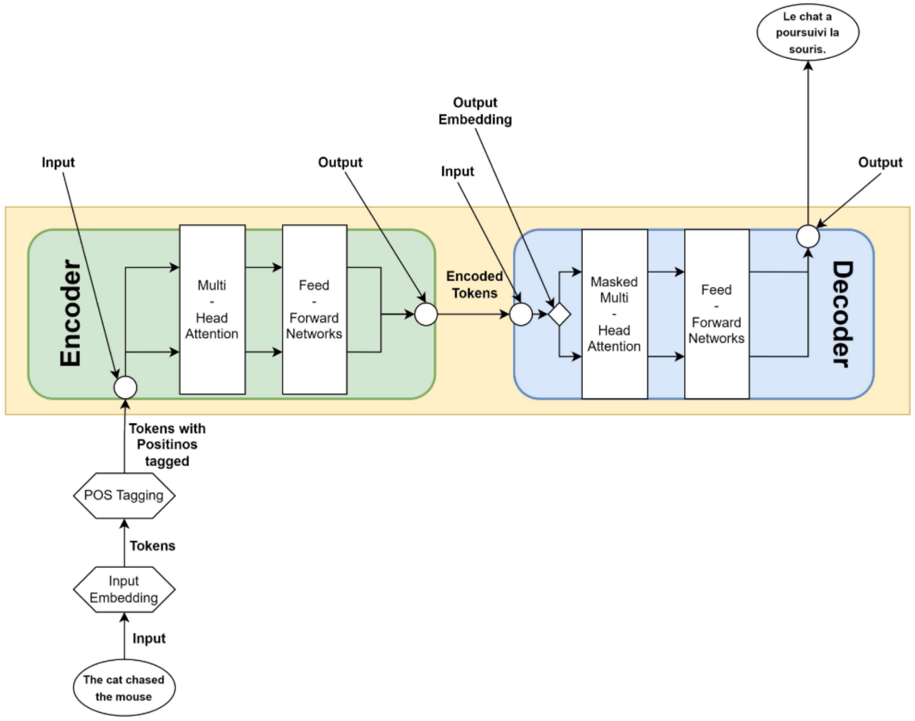
- The decoder takes the encoder’s output.
- Use masked multi-head attention to generate the output one token at a time.
- Attend to the encoder’s output to incorporate context from the input sentence while generating the translation.
- The attention mechanism in the decoder focuses on relevant parts of the input (e.g., the representation of "cat" when generating "Le chat").

Output Generation:

- The decoder generates the output sequence token by token.
- For the example, it generates: “Le”, “chat”, “a”, “poursuivi”, “la”, and “souris”.
- The complete French translation is: “Le chat a poursuivi la souris.”

Key Advantages:

- Process the entire input sequence simultaneously.
- Use attention mechanisms to capture relationships between tokens.
- Efficiently translates sentences, even with long-range dependencies.



Number of Parameters - Original Transformer Model

Component	Parameter	Formula / Size	Total Parameters
Input	Token embedding	Vocab_Size × d_model = 37000 × 512	≈ 18.94M
	Positional encoding (fixed)	n × d_model	Not learned (original paper used fixed)
Attention (per layer)	Q/K/V weights per head	3 × d_model × d_k = 3 × 512 × 64	98,304
	Output projection	d_model × d_model = 512 × 512	262,144
	Total per Multi-Head block	–	≈ 360K
Feed-Forward (per layer)	Linear 1: 512 × 2048	–	1,048,576
	Linear 2: 2048 × 512	–	1,048,576
	Total FFN per layer	–	≈ 2.10M
LayerNorm	2 × γ, β per layer	2 × d_model = 2 × 512	1,024
Encoder Block Total	–	Attention + FFN + LayerNorm	≈ 2.46M
Encoder Total (6 layers)	–	6 × 2.46M	≈ 14.76M
Decoder Total (6 layers)	Similar structure + cross attention	≈ 2.6M per layer	≈ 15.6M
Output Layer	d_model × Vocab_Size = 512 × 37000	tied/shared with embedding	≈ 18.94M
Total Model Parameters	–	Encoder + Decoder + Embedding + Output	**≈ 65M**

References

- https://jalammar.github.io/illustrated-transformer/
- https://tamoghnasaha-22.medium.com/transformers-illustrated-5c9205a6c70f

Contextual Word Embeddings

Contextual Word Embeddings (BERT and GPT)

Key Innovation

Unlike static embeddings (Word2Vec, GloVe), contextual models generate **different vectors for the same word** depending on its context.

Approach

Uses deep, pre-trained neural networks (often transformer-based)
Embeddings are derived from entire sentences, capturing syntax and semantics dynamically

Examples

- **BERT (2018)**: Transformer-based neural networks trained with masked language modeling and next sentence prediction
- GPT (2018): Transformer-based unidirectional language model focused on generation.

Characteristics

- Embeddings are **context-sensitive** (e.g., “bank” in “river bank” vs. “savings bank”)
- Each word is embedded based on its role in the sentence.
- Embeddings vary for the same word depending on its position and meaning.
- Significantly improve performance on downstream NLP tasks.

BERT – Overview and Architecture

What is BERT?

- A **pre-trained language model** based on the **Transformer encoder**
- Developed by Google in 2018
- Reads text **bidirectionally**, enabling deep contextual understanding

Key Ideas

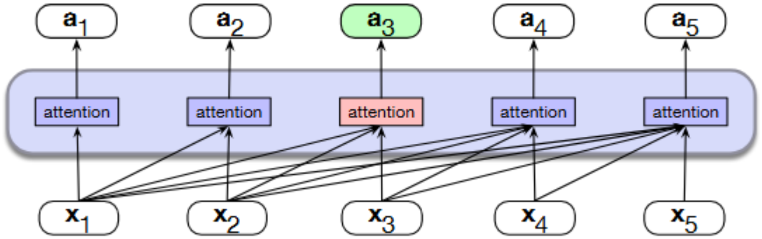
- Uses only the **encoder** stack of the Transformer
- Pre-trained on large text corpora, then fine-tuned on specific tasks

Pretraining Objectives

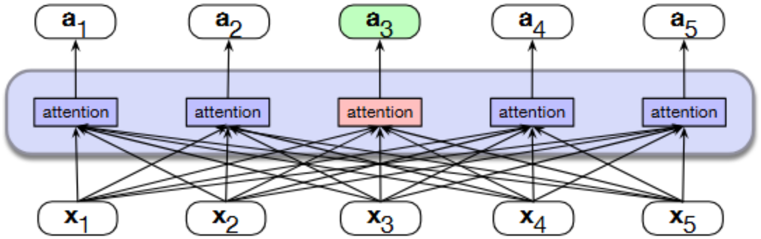
- **Masked Language Modeling (MLM)**: Predict randomly masked words in a sentence
- **Next Sentence Prediction (NSP)**: Predict if one sentence follows another

Applications

- Sentiment Analysis
- Question Answering
- Named Entity Recognition
- Text Classification
- Semantic Search



a) A causal self-attention layer



b) A bidirectional self-attention layer

Number of Parameters – BERT

Component	Parameter	**BERT-Base** (L=12, H=768, A=12)	**BERT-Large** (L=24, H=1024, A=16)
Embedding Layer	Token + Positional + Segment	30522 × 768 + 512 × 768 + 2 × 768	30522 × 1024 + 512 × 1024 + 2 × 1024
		≈ 23.8M	≈ 31.4M
Self-Attention	Q/K/V + Output per layer	4 × H^2 = 4 × 768^2 = 2.36M	4 × 1024^2 = 4.19M
	Total across all layers	12 × 2.36M = 28.3M	24 × 4.19M = 100.6M
Feedforward	Two linear layers per layer	768 × 3072 + 3072 × 768 = 4.71M	1024 × 4096 × 2 = 8.39M
	Total across all layers	12 × 4.71M = 56.5M	24 × 8.39M = 201.4M
LayerNorms	Two per layer	2 × 768 = 1.5K × 12 = 18K	2 × 1024 × 24 = 49K
Pooler	Final CLS output to 768 or 1024	768 × 768 = 0.59M	1024 × 1024 = 1.05M
Total Parameters	–	**≈ 110M**	**≈ 340M**

- L: number of layers (Transformer blocks)
- H: hidden size
- A: number of attention heads (H / A = size per head)
- Vocabulary size: 30,522
- FFN hidden size: 4×H (3072 for base, 4096 for large)

GPT – Overview and Architecture

What is GPT?

- A family of **Transformer-based language models** developed by OpenAI
- Uses only the **decoder stack** of the original Transformer architecture
- Trained with **causal (autoregressive) language modeling** to predict the next token

Training Objective

- Predict the next token in a sequence

GPT Variants

- **GPT-1**: Introduced the pretrain-then-finetune paradigm
- **GPT-2**: Scaled up model size, trained on web-scale data
- **GPT-3**: 175B parameters, enabled in-context learning
- **GPT-4**: Multimodal, stronger reasoning and generalization

Applications

- Text generation (e.g., chat, storytelling, code)
- Summarization
- Translation
- Question answering
- Semantic search and reasoning tasks

Number of Parameters – GPT 3

Component	Parameter	Formula / Size	Total Parameters (Approx.)
Embedding Layer	Token Embeddings	Vocab × d_model = 50K × 12288	614.4M
	Positional Embeddings	n_ctx × d_model = 2048 × 12288	25.2M
Self-Attention (per layer)	Q, K, V, Output	4 × d_model × d_model = 4 × 12288^2	604.6M per layer
Feedforward (per layer)	2 layers (gelu)	d_model × d_ff + d_ff × d_model	1.2B per layer
LayerNorms (per layer)	Two per layer	2 × d_model	24.6K per layer
Total per layer	–	Self-attn + FFN + norms	≈ 1.8B per layer
Transformer Block Total	96 × 1.8B	–	≈ 172.8B
Final LayerNorm	–	d_model	12.3K
Output Layer (tied)	Shared with token embedding	–	— (tied with input embedding)
Total Parameters	–	Sum of above	**≈ 175B**

- d_model = 12288
- n_layers = 96
- n_heads = 96 → each head has d_k = d_v = 128
- d_ff = 4 × d_model = 49152
- Vocabulary size ≈ 50,000
- Sequence length (n_ctx) = 2048

BERT vs. GPT

- BERT: Bidirectional, great for **understanding**
- GPT: Autoregressive, great for **generation**

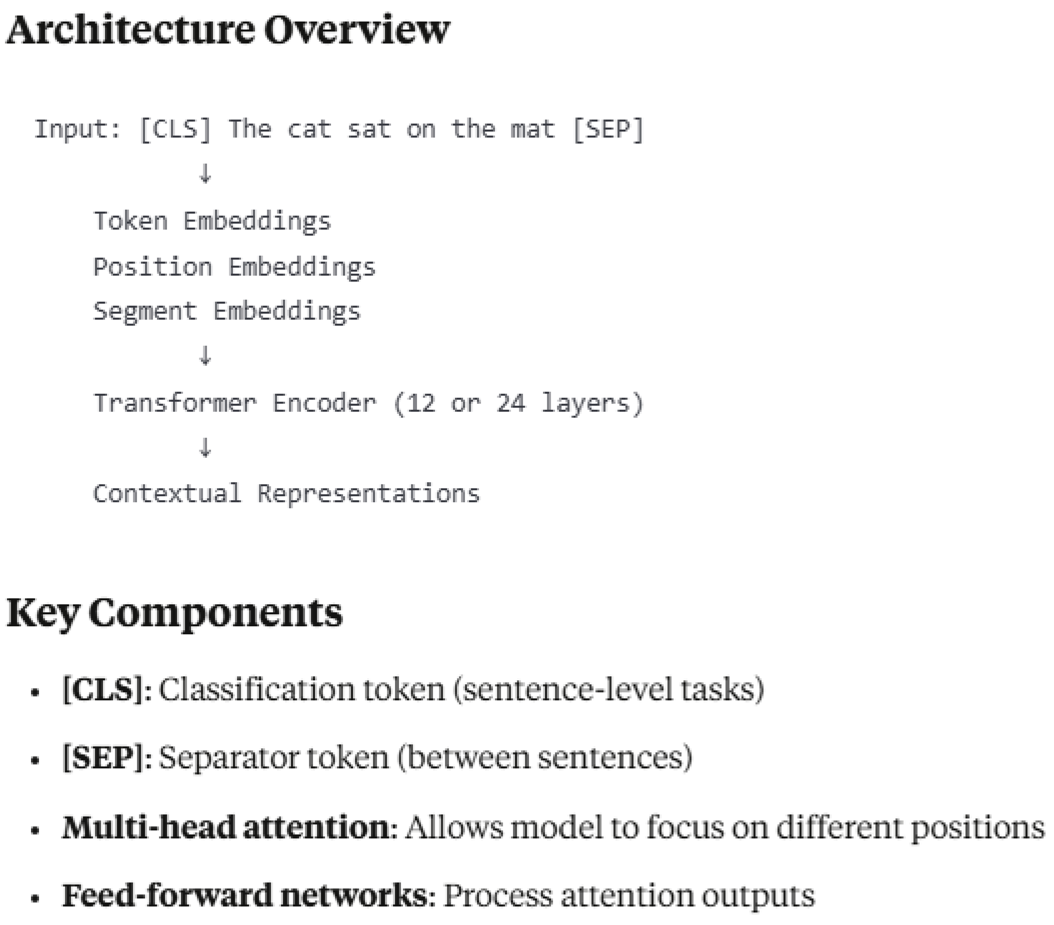
More About BERT

What is BERT?

- **B**idirectional **E**ncoder **R**epresentations from **T**ransformers
- Developed by Google AI Language in 2018
- Pre-trained language model that revolutionized NLP
- Based on Transformer architecture (Attention mechanism)

Key Advantages

- Contextual embeddings: Word meanings change based on context
- Captures long-range dependencies
- Pre-trained on massive datasets → Transfer learning
- State-of-the-art performance on 11 NLP tasks when released



Architecture Overview

```
Input: [CLS] The cat sat on the mat [SEP]
           ↓
    Token Embeddings
    Position Embeddings
    Segment Embeddings
           ↓
    Transformer Encoder (12 or 24 layers)
           ↓
    Contextual Representations
```

Key Components

- **[CLS]**: Classification token (sentence-level tasks)
- **[SEP]**: Separator token (between sentences)
- **Multi-head attention**: Allows model to focus on different positions
- **Feed-forward networks**: Process attention outputs

BERT Pre-training

Two Pre-training Tasks

1. Masked Language Model (MLM)

- Randomly mask 15% of tokens
- Predict masked tokens using context
- Example: "The [MASK] is very cute" → "cat"

2. Next Sentence Prediction (NSP)

- Given two sentences, predict if B follows A
- Helps understand sentence relationships
- Example:
  - A: "It is raining heavily."
  - B: "I need an umbrella." → True

BERT Variants

Common BERT Models

Model	Parameters	Layers	Hidden Size	Attention Heads
BERT-Base	110M	12	768	12
BERT-Large	340M	24	1024	16
DistilBERT	66M	6	768	12
RoBERTa	355M	24	1024	16
ALBERT	12M-235M	12-24	768-4096	12-64

Specialized Variants

- **BioBERT**: Biomedical text
- **SciBERT**: Scientific text
- **FinBERT**: Financial text
- **ClinicalBERT**: Clinical notes

Brief Introduction to Hugging Face 🤗

The Problem They Solved

Before Hugging Face (2018)

- Models locked in research papers
- Complex implementations
- Different APIs for each model
- Difficult to compare/use models
- Limited accessibility

After Hugging Face

- Unified API for all models
- Easy model sharing
- Standardized interfaces
- Community-driven development
- Democratized AI access

2016: Founded as a Chatbot Company

- Started with conversational AI
- Built tools for developers
- Recognized broader need

2018: The Pivot

- Released transformers library
- Open-sourced BERT implementation
- Explosive community growth

Today: The GitHub of AI

- 350,000+ models
- 50,000+ datasets
- Millions of users
- Industry standard

The Hugging Face Ecosystem

Core Components

1. Transformers Library

```
from transformers import pipeline
# One line to use any model
classifier = pipeline("sentiment-analysis")
result = classifier("I love Hugging Face!")
```

2. Model Hub

- Centralized model repository
- Version control for models
- Model cards (documentation)
- Direct integration with code

3. Dataset Library

```
from datasets import load_dataset
dataset = load_dataset("imdb")
# Automatic downloading, caching, processing
```

4. Tokenizers

- Fast tokenization (Rust-based)
- Supports all tokenization schemes
- Handles special tokens
- Production-ready speed

5. Spaces

- Host ML demos for free
- Gradio/Streamlit integration
- Share with one click
- GPU support available

Why This Matters

Impact on AI Development

1. Accessibility

- PhD not required ✓
- Free models and tools ✓
- Great documentation ✓
- Active community ✓

2. Reproducibility

- Standardized implementations
- Version control
- Model cards with metrics
- Easy sharing

3. Innovation Speed

- Build on others' work
- Rapid prototyping
- Easy experimentation
- Quick deployment

4. Democratization

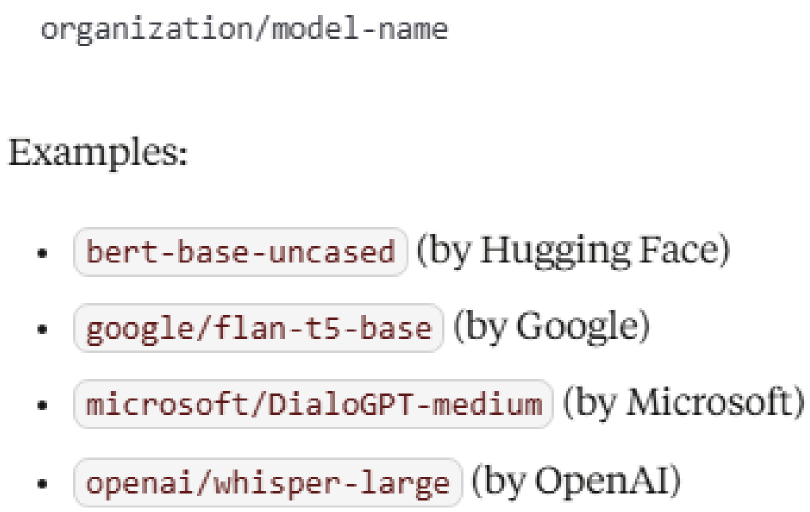
- Small teams can compete
- Global collaboration
- Open science
- Reduced barriers

Model Hub Deep Dive

Understanding Model Cards

Every model on Hugging Face has:

- **Model Card**: Documentation about the model
- **Files**: Model weights, config, tokenizer
- **Metrics**: Performance benchmarks
- **License**: Usage restrictions
- **Tags**: Task type, language, framework



```
organization/model-name
```

Examples:

- `bert-base-uncased` (by Hugging Face)
- `google/flan-t5-base` (by Google)
- `microsoft/DialoGPT-medium` (by Microsoft)
- `openai/whisper-large` (by OpenAI)

From Concept to Code - The Pipeline Abstraction

- **Traditional Approach**

1. Load tokenizer

2. Preprocess text

3. Load model

4. Run inference

5. Post-process results

... 50+ lines of code

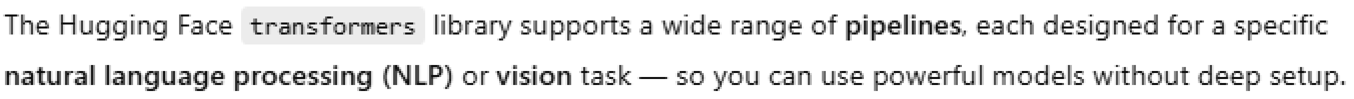
Hugging face Approach

```
from transformers import pipeline
classifier = pipeline("task-name")
result = classifier("your text")
# 2 lines!
```

Under the Hood

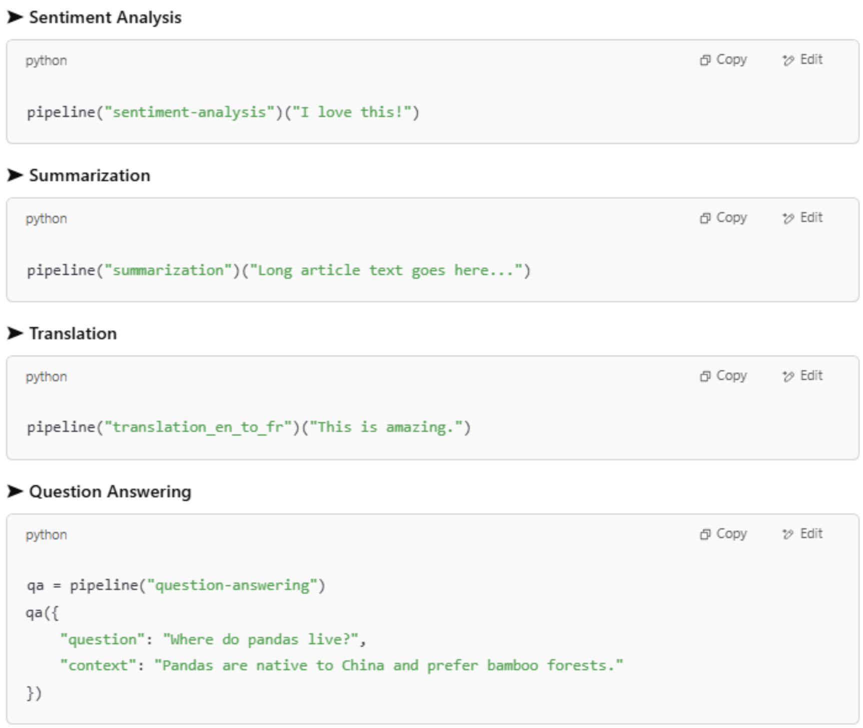
1. **Automatic model selection**
2. **Tokenization handled**
3. **Inference optimization**
4. **Result formatting**
5. **Device management**

Other Hugging face Piplines



The Hugging Face `transformers` library supports a wide range of **pipelines**, each designed for a specific **natural language processing (NLP)** or **vision** task — so you can use powerful models without deep setup.

Pipeline Name	Task Description
"sentiment-analysis"	Classify sentiment (positive/negative)
"text-classification"	General text classification (multi-label or multi-class)
"zero-shot-classification"	Classify into labels **without training** on them
"text-generation"	Generate text (e.g., GPT models)
"text2text-generation"	Text-to-text tasks (e.g., summarization, translation)
"translation"	Translate between languages
"summarization"	Generate a summary of input text
"question-answering"	Extract answer from context
"fill-mask"	Predict missing word in a sentence (BERT-style)
"ner" (Named Entity Recognition)	Extract entities (like names, places, etc.)
"conversational"	Chatbot-style conversation
"sentence-similarity"	Measure similarity between two sentences
"token-classification"	Classify each token (used for NER, POS tagging, etc.)
"feature-extraction"	Extract embeddings/features from a model
"table-question-answering"	QA over structured data (tables)



➤ **Sentiment Analysis**

```python
pipeline("sentiment-analysis")("I love this!")
```

➤ **Summarization**

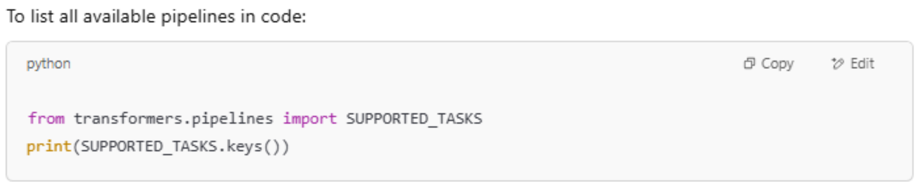
```python
pipeline("summarization")("Long article text goes here...")
```

➤ **Translation**

```python
pipeline("translation_en_to_fr")("This is amazing.")
```

➤ **Question Answering**

```python
qa = pipeline("question-answering")
qa({
    "question": "Where do pandas live?",
    "context": "Pandas are native to China and prefer bamboo forests."
})
```



To list all available pipelines in code:

```python
from transformers.pipelines import SUPPORTED_TASKS
print(SUPPORTED_TASKS.keys())
```

Example: Print out BERT Embeddings

```
from transformers import BertTokenizer, BertModel
import torch

# Load pretrained BERT
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased’)

# Sentence
sentence = "He went to the bank to deposit money.“

# Tokenize
inputs = tokenizer(sentence, return_tensors='pt’)

# Get outputs
with torch.no_grad():  # No training, just inference
    outputs = model(**inputs)

# Get the hidden states (embeddings)
embeddings = outputs.last_hidden_state  # Shape: (batch_size, sequence_length,

# (hidden_size)
print(embeddings.shape)  # Example output: torch.Size([1, 11, 768])
```

Open in Colab

Example: Question Answering using BERT

QA Task Overview

- **Input**: Context paragraph + Question
- **Output**: Answer span from the context
- BERT identifies start and end positions of answer

QA Pipeline Components

1. **Tokenization**: Convert text to tokens
2. **Encoding**: Create input embeddings
3. **Model Inference**: Get start/end logits
4. **Post-processing**: Extract answer text

```python
# Import required libraries
from transformers import AutoTokenizer, AutoModelForQuestionAnswering
from transformers import pipeline
import torch

# Using pipeline (High-level API)
qa_pipeline = pipeline( "question-answering",
model="bert-large-uncased-whole-word-masking-finetuned-squad",
tokenizer="bert-large-uncased-whole-word-masking-finetuned-squad" )

# Example usage
context = """ BERT is a method of pre-training language representations,
meaning that it trains a general-purpose language understanding
model on a large text corpus (like Wikipedia),
and then uses that model for downstream NLP tasks like question answering. """

question = "What is BERT?"
result = qa_pipeline(question=question, context=context)
print(f"Answer: {result['answer']}")
print(f"Confidence: {result['score']:.4f}")
```

Model Choice Explanation

- bert-large-uncased-whole-word-masking-finetuned-squad
  - Based on BERT-Large architecture
  - Uncased: converts text to lowercase
  - Whole word masking: improved pre-training
  - Fine-tuned on SQuAD dataset (Stanford Question Answering Dataset)

Open in Colab

Fine-Tuning Large Language Models (LLMs)

What is Fine-Tuning?

- **Pre-training Phase**
  - Trained on large corpus using Masked Language Modeling (MLM) and Next Sentence Prediction (NSP)
- **Fine-Tuning Phase**
  - Fine-tuning is the process of continuing the training of a pretrained LLM on a smaller, task-specific dataset.
  - The objective is to specialize the model for a particular use case or domain.
- **Why Fine Tune LLMs?**
  - Improve performance on specific tasks.
  - Inject domain-specific knowledge (legal, medical, financial, etc.).
  - Adapt to company-specific language or tone.
  - Reduce inference cost by limiting model size and scope.
- **Use Cases of BERT Fine-Tuning**
  - Customer Support Chatbots (trained on company FAQs).
  - Legal Document Analysis.
  - Scientific Paper Summarization.
  - Code Assistants for specific frameworks.
  - Sentiment classification for product reviews.

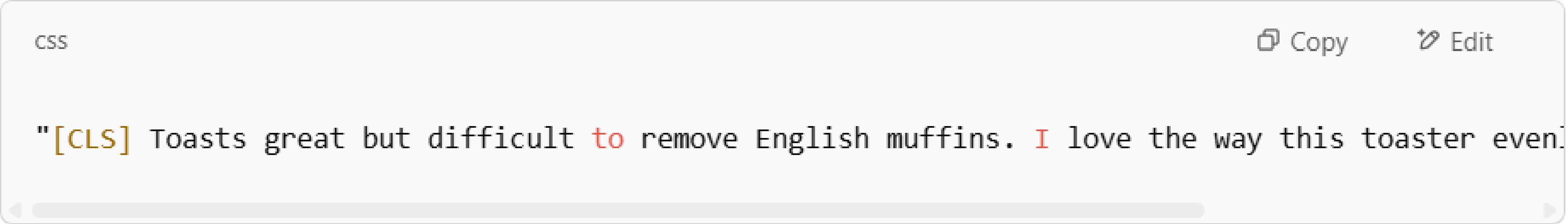
Fine-Tuning Workflow

1. Load pre-trained BERT model
2. Add task-specific head (e.g., linear layer)
3. Tokenize and preprocess input text
4. Define loss function and optimizer
5. Train on labeled dataset
6. Evaluate and deploy

CO Open in Colab

Data Preparation

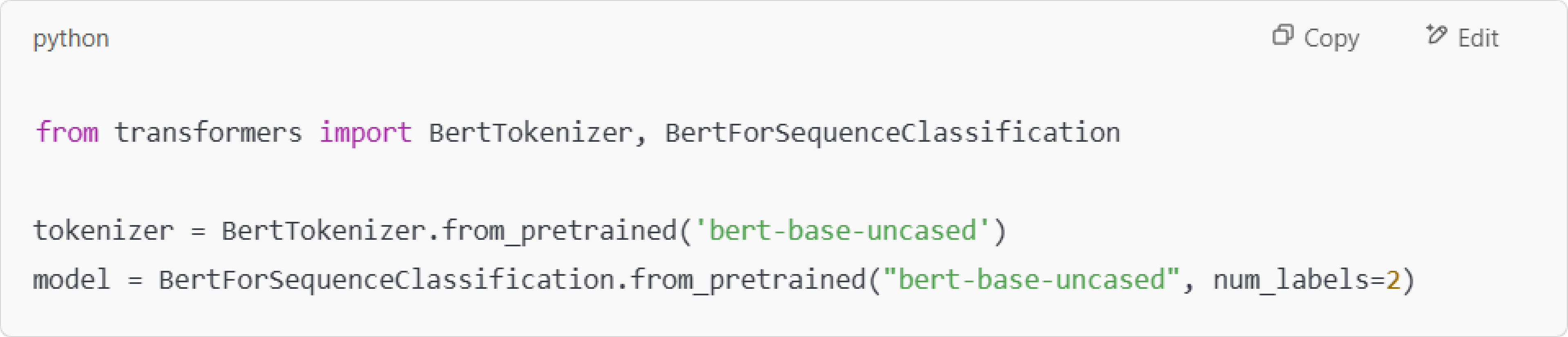
- Concatenate `title` and `content` into a single input string.
- Example input:



```css
"[CLS] Toasts great but difficult to remove English muffins. I love the way this toaster evenl
```

- Preprocess:
  - Lowercase (if using `bert-base-uncased`)
  - Tokenize with `BertTokenizer`
  - Pad/truncate to max length (e.g., 256 tokens)
  - Encode labels (`0`, `1`)

Model Setup

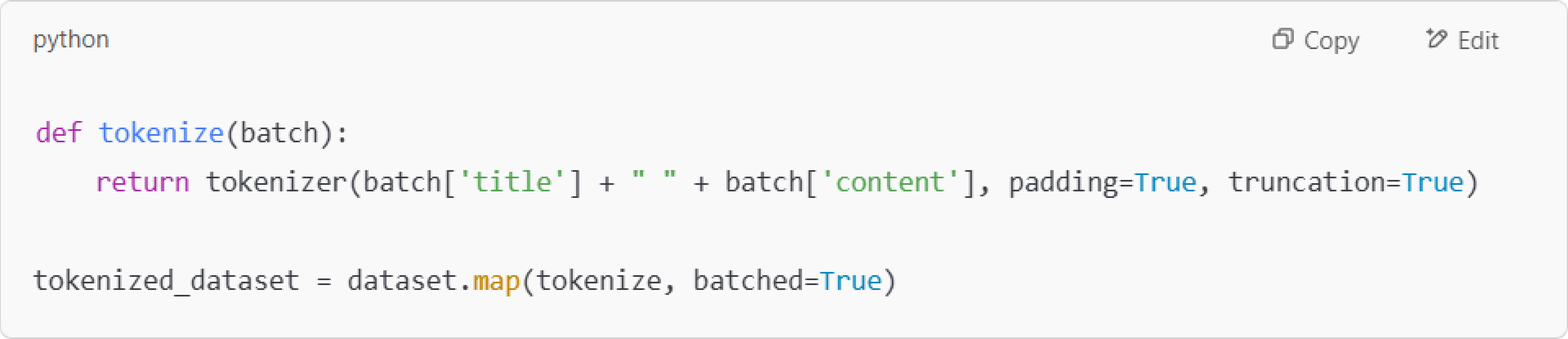


```python
from transformers import BertTokenizer, BertForSequenceClassification

tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
model = BertForSequenceClassification.from_pretrained("bert-base-uncased", num_labels=2)
```

- Use `BertForSequenceClassification`
- `num_labels=2` for binary classification

Tokenizing the Dataset

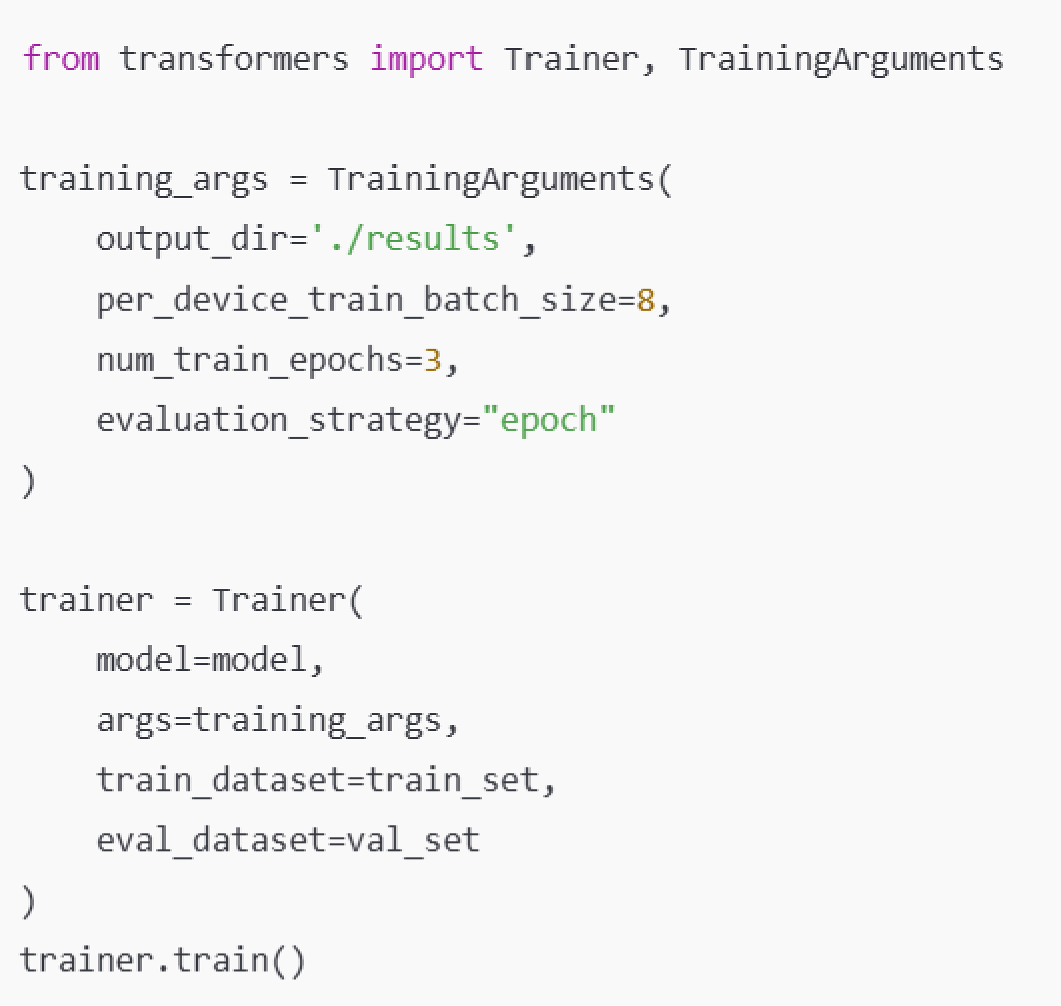


```python
def tokenize(batch):
    return tokenizer(batch['title'] + " " + batch['content'], padding=True, truncation=True)

tokenized_dataset = dataset.map(tokenize, batched=True)
```

- Concatenate fields
- Apply BERT tokenization

Training Step



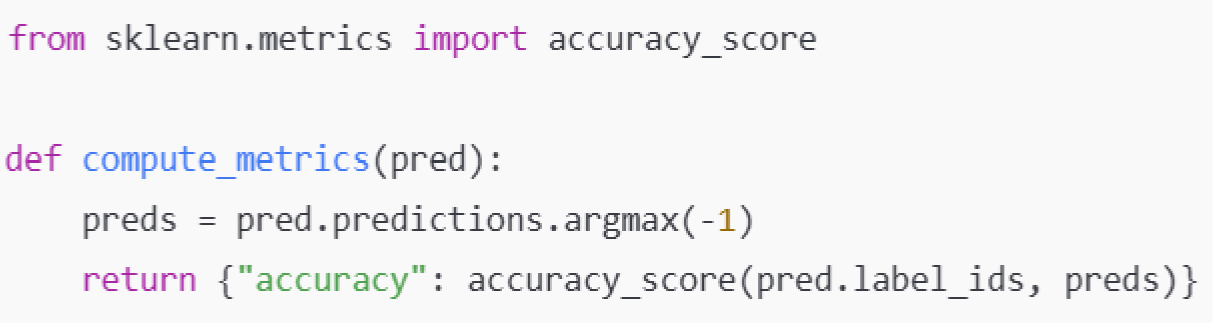
```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',
    per_device_train_batch_size=8,
    num_train_epochs=3,
    evaluation_strategy="epoch"
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_set,
    eval_dataset=val_set
)
trainer.train()
```

- a

Evaluation



```
from sklearn.metrics import accuracy_score

def compute_metrics(pred):
    preds = pred.predictions.argmax(-1)
    return {"accuracy": accuracy_score(pred.label_ids, preds)}
```

- Use metrics like:
  - Accuracy
  - F1 Score
  - Precision/Recall