AI & Machine Learning for Genomic Data Science

Master in Genomic Data Science – Università di Pavia

Barbara Tarantino

AA 2025-2026

Course Outline

- Day 1 → Python Foundations & AI in Medicine
- Day 2 → Core Machine Learning for Genomics
- Day 3 → Deep Learning Foundations (PyTorch)
- Day 4 → Computer Vision for Medicine
- Day 5 → Large Language Models & Clinical Text

Objective: to acquire theoretical and practical bases to apply AI to genomics, clinical images, medical text.

Program - Day 5

Large Language Models & Clinical Text

- Introduction to clinical text
- NLP Fundamentals: Tokenization, Vocabulary, Embedding
- Dalle RNN ai Transformer: Encoder, Decoder, Attention
- Large Language Models (LLM): pre-training, fine-tuning, inference
- Hugging Face Ecosystem: tokenizer, modelli, pipeline
- Biomedical LLMs: BioGPT, PubMedBERT

Practical organization – Day 5

Morning (9:30 – 12:30)

- Introduction to clinical text
- Representing language: from tokens to embeddings
- From vectors to models: RNN, Transformer and LLM

Afternoon (14:30 – 17:00)

- Python ecosystem for LLMs
- Notebook pratico: `Biomedical_LLMs_Intro.ipynb`

Section 1

Introduction to Clinical Text

Medical language as a source of knowledge

- In medicine, **much of the clinical information is textual**: reports, folders, medical notes, guidelines, scientific articles.

- These texts contain **crucial data** on diagnoses, symptoms, treatments and outcomes.

- But they are **written in natural language**, not in numerical form.

The goal of AI is to transform this textual information into analyzable data, without losing its meaning.

Why text is difficult to analyze automatically

- Words are not numbers: machines **do not understand language**, they process it only if it is formalized.
- Each doctor writes differently → high **linguistic variability**.
- **Acronyms**, abbreviations, or mixed languages ("MI", "BP", "ECG") are often used.
- The texts are **long, messy and unstructured**.

We need a way to "teach" machines to read and interpret medical language.

Examples of biomedical text

Source	Content example	Language
Clinical reports	"Paz. with chest pain, positive ECG, beta-blocker therapy."	abbreviated, technical
Medical records (EHRs)	Notes, medical history, symptoms	natural, subjective
Scientific literature	“BRCA1 mutation associated with breast cancer risk.”	formal, precise
Forums or patient notes	"For days I have been feeling sluggish and nauseous."	informal, emotional

A model must adapt to different registers of language: technical, formal, colloquial.

The challenges of the biomedical text

- **Ambiguity:** the same word can have multiple meanings ("MI" → *heart attack* or *mitral regurgitation*).
- **Variability:** Each hospital, doctor, or specialty has its own way of writing.
- **Multilingual:** mixed texts between English, Latin and Italian.
- **Data sensitivity:** anonymization is needed.
- **Few annotated data:** techniques that learn from raw texts are needed.

Medical language is complex, but very rich in information: we must learn to represent it mathematically.

What is Natural Language Processing (NLP)

- **NLP** (Natural Language Processing) is the field of AI that studies **how to make machines understand human language**.

- In medicine it is used for:
  - identify clinical concepts (diagnosis, drugs, symptoms);
  - analyze reports or scientific articles;
  - integrate text with other data (genomic, clinical, images).

In short: NLP allows you to switch from natural language to machine language

The text as data: from information to representation

- In order to be analyzed by a model, **the text must be represented in numerical form.**

- Words become **mathematically manipulable units of information.**

- Only in this way can models *recognize linguistic patterns* and *learn relationships between concepts*.

This is the fundamental challenge of AI applied to language: transforming text into numbers that have meaning.

From text to vectors

Clinical text (unstructured)

↓

Language pre-processing (cleaning, tokenization)

↓

Numerical representation (vectors)

↓

AI model (learning patterns and meanings)

The next section shows exactly this step:
how words become numbers and what they represent in machine space.

Section 2

Representing language: from tokens to embeddings

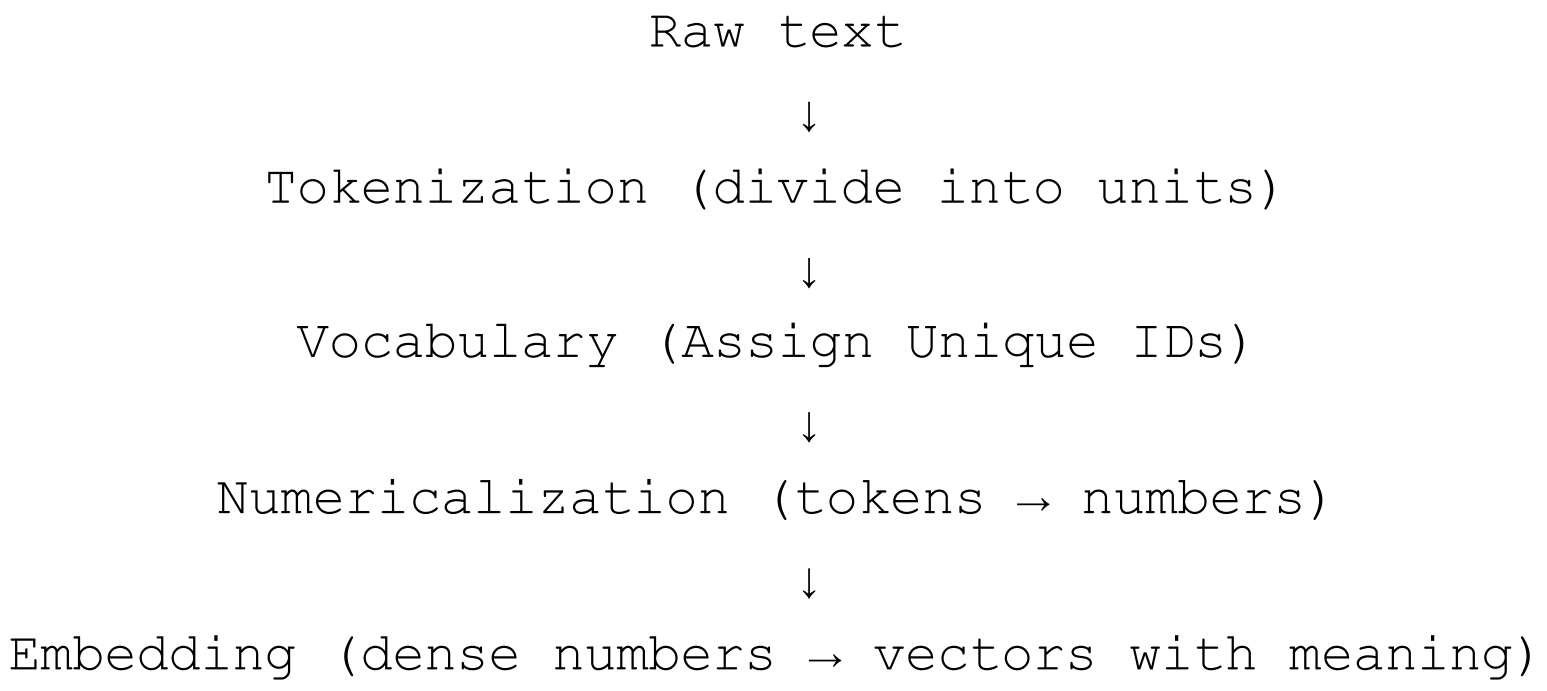
From natural language to numerical representation

- Machines don't "understand" words: **they have to work on numbers**.
- The goal of AI is **to transform language into mathematical vectors**.
- Each word → a **numerical representation (embedding)** that captures its meaning.

```
"patient with high fever" → [0.24, -0.53, 0.72, ..., 0.15]
```

- Thus the machine can **measure similarities and semantic relationships**:
  - "heart" near "cardiac", away from "computer".

Text representation pipelines



- Each step reduces the distance between **human language** and **machine language**.
- All NLP modeling (RNN, Transformer, LLM) starts from here.

Tokenization: Split text into units

- **Tokenization** breaks text into **minimal units (tokens).**

- A token can be:
  - one **word** ("aspirin")
  - a **subword** ("cardio" + "logia")
  - one **character** ("a", "b", "c")

- Example:
  "Acute Myocardial Infarction" → [Infarction, Myocardial, Acute]

- For modern models**, tokenizer subwords** (*WordPiece*, *BPE)* are used:
  - manage abbreviations and rare words
  - "myocardial" → "my", "cardic"

Vocabulary: building the alphabet of the model

- After tokenization, we build a **vocabulary**, i.e., the **list of all unique tokens**.
- Simplified example:

Token	ID
"patient"	2
"fever"	3
"high"	4

- Vocabulary is used to translate words numbers.

"high fever patient" → [2,3,4]

- In LLMs, the vocabulary contains tens of thousands of tokens, including symbols and punctuation marks.

One-Hot Encoding: the first numerical representation

- Each token becomes a vector of zeros and ones:

```
"Patient with high fever"
"patient" → [1,0,0,0]
"fever" → [0,0,1,0]
"high" → [0,0,0,1]
```

- Problems:
  - **very long vectors and almost all zero**;
  - no relationship between words.

We need a **dense** representation where proximity reflects meaning.

What are Word Embeddings

- An **embedding** is a **vector of real numbers** (e.g. 300 dimensions) that represents a word.

- Example (3 sizes, simplified):

```
"patient" → [0.12, 0.85, 0.33]
"fever" → [0.15, 0.88, 0.30]
"computer"  → [-0.80, 0.10, 0.60]
```

- "Patient" and "fever" are **close in space**, "computer" far away.
- Each direction of the vector captures a latent characteristic (clinical ↔, technological, positive ↔, negative…).

- Basic principle:
- *The meaning of a word is given by the context in which it appears.* (J.R. Firth, 1957)

How static embeddings are learned

1. Word2Vec – predictive model

- Learn to predict words that appear close together:
  - *Skip-Gram*: "heart" → predicts "pain", "thoracic", "ECG"
  - *CBOW:* "ECG chest pain" → predicts "heart"
- After training, words that share similar contexts will have **similar vectors**.

2. GloVe – co-occurrence model

- It creates a large "word × context" matrix (how many times they appear together) and learns its structure.

Word	x_1	x_2
heart	0.6	0.8
cardiac	0.5	0.7
lung	0.9	0.1

→ distance (heart, cardiac) < distance (heart, lung)

What static embeddings capture

- Each word is a **point in space**: similar words appear **close together**, different words **far away**.
- Relationships between words become **geometric relationships**:
  - `v(re) − v(uomo) ≈ v(regina) - v(donna)`
- The model "learns" that words that co-occur often share meaning.
- They are therefore **global and stable representations** of the average meaning of a word, regardless of the context.

Why static embeddings are not enough

- **Same vector, different contexts** → the machine does not distinguish shades of meaning.
  - “MI diagnosticato all’ECG” → *myocardial infarction*
  - "Mitral MI prolapse" → *mitral insufficiency*
    → For the static model, "MI" is identical.

- **Ignored context:**
  - "chronic patient" ≈ "happy patient" → same embedding for "patient".

- **Logical conclusion:**
  Static models fix a unique meaning per word → need **dynamic representations** that fit the sentence.

This is how contextual **embeddings were born**, the basis of modern models (ELMo, BERT, BioBERT…).

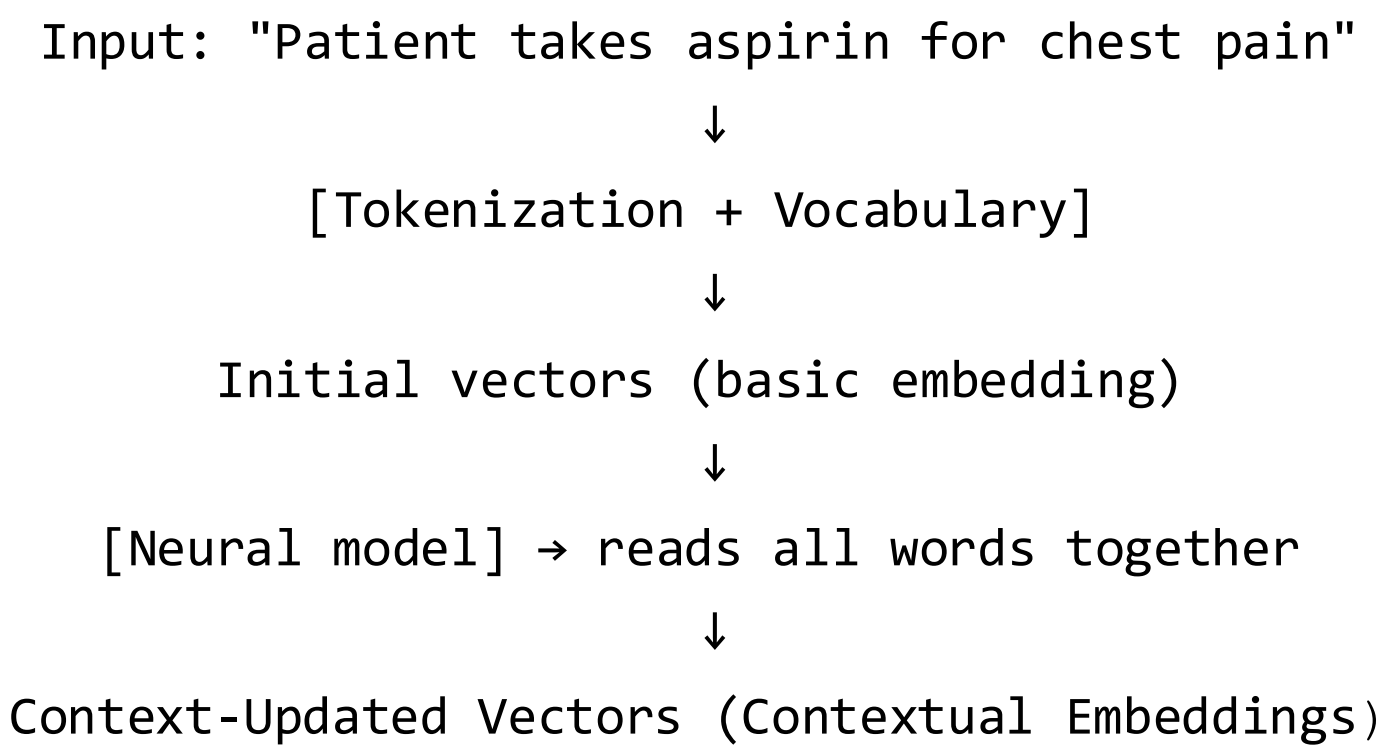
From statistical embeddings to contextual embeddings

- In static embeddings, one word = only one vector → fixed meaning.
- In contextual embeddings, a word = a vector that changes with the sentence.
- The model does not assign an "average" meaning, but infers it from the surrounding context.

Sentence	Meaning of "MI"	Vector (different)
"I was diagnosed on ECG"	Myocardial infarction	[0.82, 0.10, 0.33, ...]
“Prolasso MI mitralico”	Mitral insufficiency	[0.14, 0.87, 0.25, ...]

- The model **understands the whole sentence** before constructing the vector of each word.

How a contextual embedding works



How to update embeddings in the neural model

1. Embedding di base

- Each word is initially represented by a **numeric vector** (embedding).
- These vectors are **learned parameters**: initially random.
- They represent the *semantic position* of a word in space (e.g., "aspirin" next to "drug").

2. Transition into the neural model

- A sequential neural network) processes all vectors **together**.
- Thanks to the **weights** and **attention mechanisms**, the model "understands" the relationships between words.
- Each word "looks" at the surrounding context (e.g. *"chest pain"* changes the meaning of *"aspirin")*.

3. Embeddings updated from context

- After passing through the template, each vector is **updated** to reflect the meaning *in the context of the sentence*.
- These are the **contextual embeddings**:
  - "aspirin" → specific drug in this context
  - "Pain" → symptom, not a generic concept
- Each layer in the model produces a more abstract and rich representation.

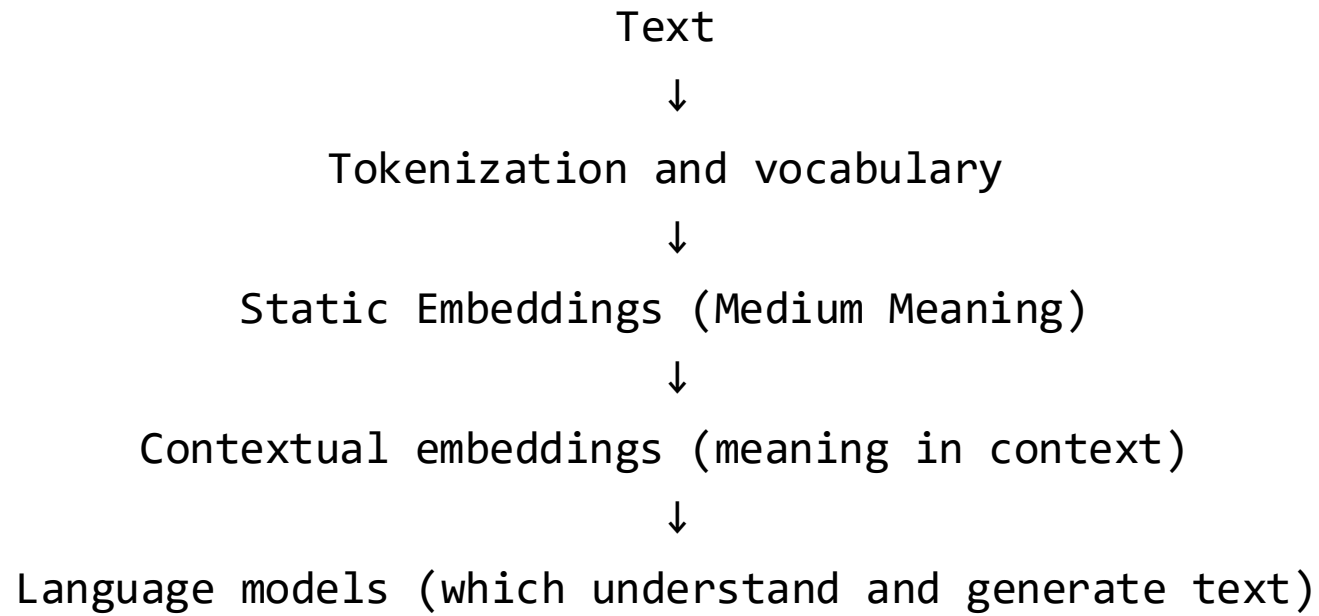
Because they are a revolution

- Before: the meaning was **static and local** → was not enough to understand long or ambiguous text.

- Now: the meaning is **dynamic and global** → the model understands the context and role of the word.

- Contextual embeddings are therefore:
  - the **basis for neural language models** (RNN and Transformer);
  - the **bridge** between representation and understanding.

From language representation to language models



- So far we have learned to **represent language** (from text → numbers → meaning).
- Now let's see **how neural models** use these representations to **understand, reason, and generate text.**

Section 3

From vectors to models: RNN, Transformer and LLM

From word vectors to templates

- **Embeddings** represent words as **numbers with meaning**.
- Now we need to understand **how a neural network** uses these vectors to **understand and generate text**.
- A **language model** is a system that **predicts the next word** based on previous ones.

Example:

"The patient assumes..." → the model predicts "aspirin" with high probability.

From words as numbers → to models that learn the rules of language.

Why sequential models are needed

- Textual data are **ordered sequences**.
- To predict the next element, the **context of the previous ones** is needed.
- **Feed-Forward Networks (FNNs)** treat inputs independently → **no memory**.
- We need a model that **remembers over time → Recurrent Neural Networks (RNNs)** were born.

Read a sentence word for word: To understand the last word, you need to remember the previous ones.

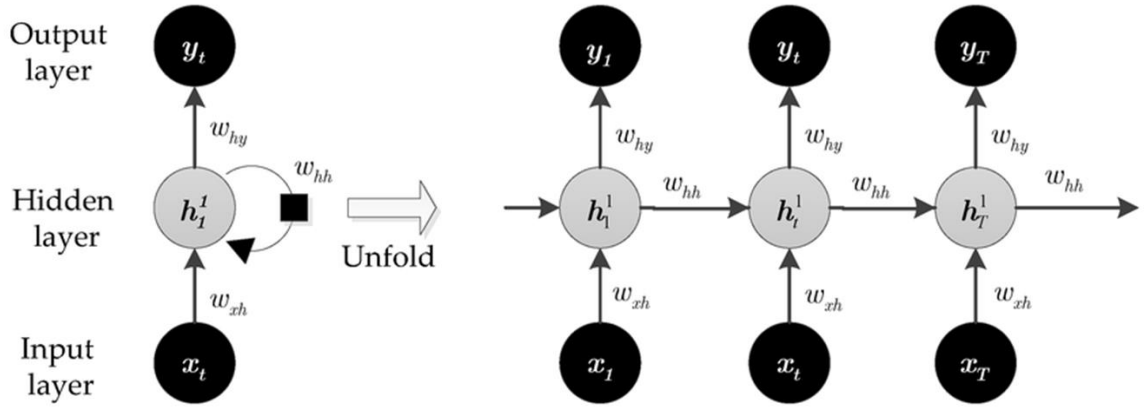
How an RNN works

- A **Recurrent Neural Network** processes a **sequence word by word (or token by token).**

- At every step of time :t

  - **Reads the current input** → the current word x_t
  - **Combine** with **previous** statush_{t-1}
  - **Produces:**
    - A new **updated** statush_t
    - an **output** , which can be the y_t**prediction of the next word**

$$h_t = f(W_{xh}x_t + W_{hh}h_{t-1})$$
$$y_t = g(W_{hy}h_t)$$



Limits of RNNs

- RNNs read one word at a time and remember over time,
  but **this memory is fragile and slow**.

- As the sequence increases:

  - **Vanishing gradient** → the network forgets what is far away,
    because the gradients are reduced too much propagating backwards.
  - **Short-term memory** → can only remember the last words,
    losing the overall context of the sentence.
  - **Sequential process** → each word depends on the previous one → no parallelization.

Solution: Networks with "smart" memory

- **LSTMs (Long Short-Term Memory)** and **GRUs (Gated Recurrent Units)** add a small "control system" called **gates**.

- Each gate decides **what information to keep or discard** from the time stream:
  - **Forget gate** → decides *which parts of the past memory to delete*
  - **Input gate** → decides *what new information to add*
  - **Output gate** → decides *which part of the memory to use as output*

Problem in RNNs	LSTM / CRANE solution
Vanishing gradient → memory "vanishes" after a few words	The gates control the flow of the gradient → the error signal does not cancel out
Loss of context → don't remember long sentences	They maintain a cell state that retains important information over time
Slow and unstable sequential processing	More stable learning and less sensitive to sequence length

From local memory to global attention

Final limit of RNNs, LSTMs and GRUs

- Even though they remember better, **they still read word for word**
- The context is constructed **in a linear way** (one word after another)
- They can't "see" all the words together to understand more complex relationships

Transition to Transformers

- Transformers eliminate sequential reading
- They look at **all the words in parallel**
- They understand **which words influence each other** thanks to the **Self-Attention mechanism**

Key idea:

LSTM and GRU remember over time.

Transformers **understand the whole context together**.

The paradigm shift: from local memory to global context

- RNNs "read" word for word → lens, losing the distant context.
- Transformers read **all the words together**, and learn **which words influence which**.
- Sequential memory is no longer needed: the context is understood in a single step.
- Idea *chiave: ogni parola presta attenzione alle altre*.

What does "attention" mean

- "Attention" ≈ **how important one word is to understand another**.
- When the model processes the current word, it looks at all the others and weighs the most relevant ones.
- Example:
  "Patient takes aspirin for chest pain."
  → to understand *"aspirin,"* the model pays attention to *"chest pain."*
- Attenzione = *focus selettivo sul contesto rilevante*.

General architecture of the Transformer

Each Transformer is composed of **repeating blocks** with two key subcomponents:

1. **Self-Attention Layer** → allows every word to see all the others

2. **Feed-Forward Layer** → processes the collected information

Each block includes:

$$\text{Output} = \mathit{LayerNorm}\Big(x + \mathit{FeedForward}\big(\mathit{SelfAttention}(x)\big)\Big)$$

Residual connection = direct input → output connection → maintains information and stabilizes learning.

Input del Transformer

- For each word in the text:

1. You get the **embedding vector**
2. **Positional encoding** is added (position in the sentence)

$$x_i = Embedding(token_i) + PositionalEncoding(i)$$

each contains x_i**meaning + position** of the word.

Self-Attention: l'idea di base

- For each word, the model constructs three representations: x_i

$$Q_i = x_i W_Q, K_i = x_i W_K, V_i = x_i W_V$$

- **Query (Q):** What am I looking for?
- **Key (K):** What information do I offer?
- **Value (V):** What content do I stream?

Origin of weights W_Q, W_K, W_V

- The weights belong to W_Q, W_K, W_V **three small linear layers (feed-forward)** that transform the vector of each word into the three spaces Query, Key, and Value. x_i

- For each word, the model applies three linear transformations:

$$x_i \xrightarrow{\textit{Linear Layer}} Q_i,\ K_i,\ V_i$$

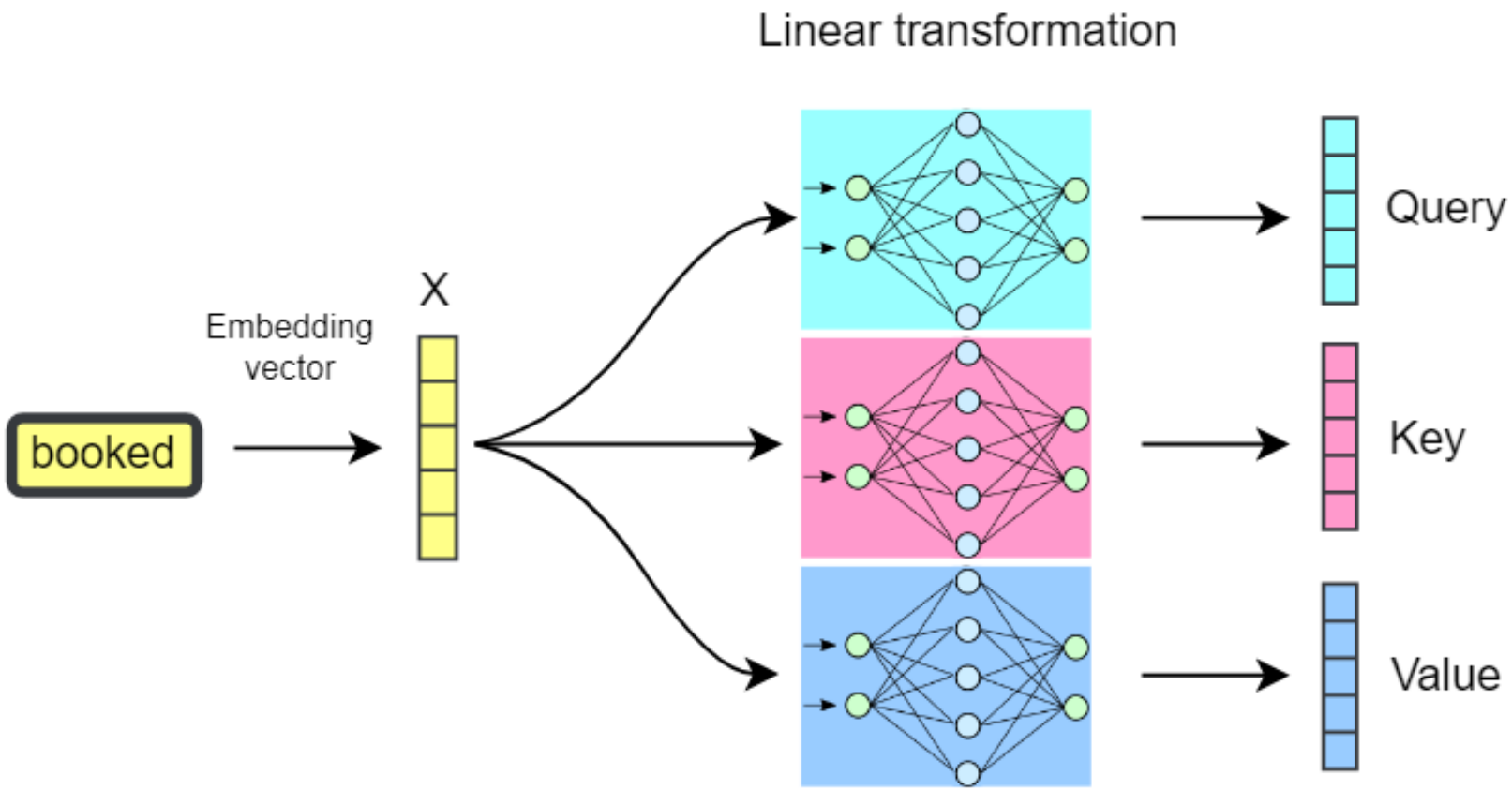
- These "linear layers" are **dense neural networks** (matrices of weights + bias) without activation functions, i.e. simple projections:

$$Q_i = x_i W_Q + b_Q, K_i = x_i W_K + b_K, V_i = x_i W_V + b_V$$

- During training, gradients update exactly like the weights of an MLP. W_Q, W_K, W_V

- In this sense, each *attention head* is built on top of **three linear feed-forward networks** shared among all tokens.

From embedding to Q, K, V projections



Calculating the "similarity" between words

- Each word "questions" all the others:

$$\text{score}(i, j) = Q_i \cdot K_j^T$$

- → measure how relevant the word *j* is to the word *i*.
  The taller the product, the more *i* pays attention to *j*.

- It normalizes with a softmax to obtain the weights **of attention**:

$$\alpha_{ij} = softmax\left(\frac{Q_i K_j^T}{\sqrt{d_k}}\right)$$

the term (key vector dimension K) stabilizes gradients $\sqrt{d_k}$

Information aggregation

- Once the attention weights have been calculated , α_{ij} the model combines the information of the most relevant words.

$$\text{Attention}(Q_i) = \sum_j \alpha_{ij} V_j$$

- Each word i **updates its vector** by combining the **Values** V_j of the other words, weighted according to **how important it considers them** (i.e. the weights). α_{ij}

- The new representation of the word contains pieces of information i from all the words in the context, in proportion to their relevance.

User-friendly example

- Sentence: "The patient takes **aspirin** for **chest pain**."
- "aspirin" → : he asks Q_i *"with which words am I semantically linked?"*
- Compare with all other words.$Q_i K_j$
- The weights will be high for "pain" and "thoracic".α_{ij}
- Combine the **values** V_j of those words → get the new vector that now represents "aspirin in the context of chest pain."**Attention**$(\boldsymbol{Q_i})$

In few words

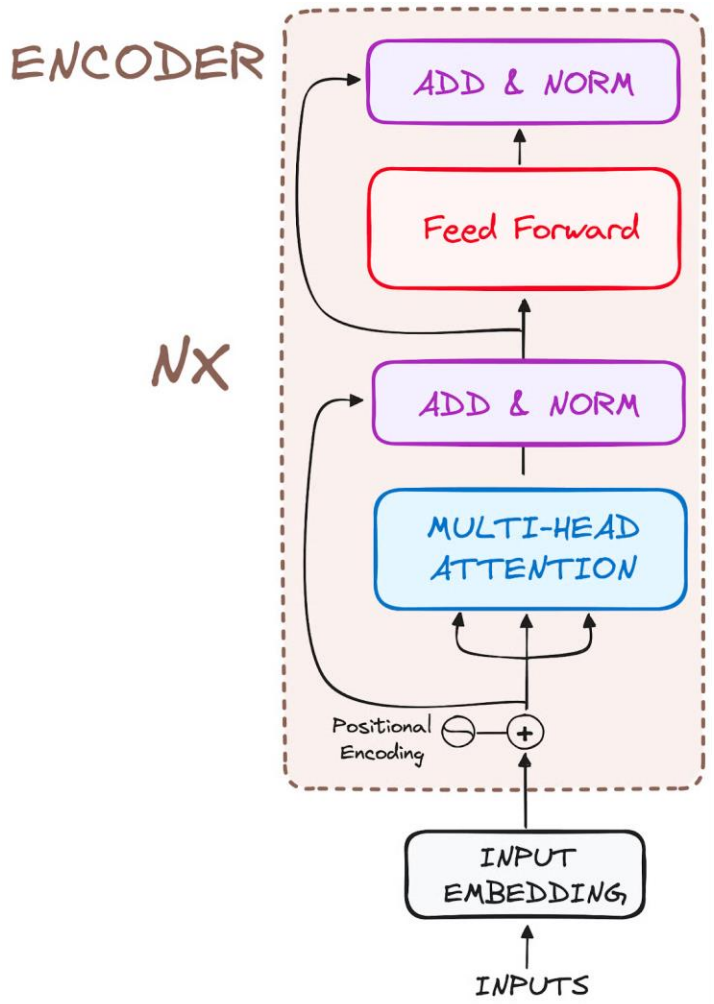
Symbol	Intuitive meaning
Q_i	The "question" posed by the word i
K_j	The "key" offered by the word j
V_j	The informative content of the word j
Attention(Q_i)	**the new embedding** of , calculated as a weighted average of the most relevantiV_j

After self-attention

- After calculating the new attention vectors, each token also passes into another **nonlinear feed-forward network** (two layers with ReLU), which processes and combines the information learned from attention.

- It is the block called **the Feed-Forward Network (FFN)** of the Transformer:

```
ogni token → [Self-Attention] → [Feed-Forward Network] → [Residual + LayerNorm]
```



Multi-Head Attention

In practice, the model does not use a single attention, but **several heads in parallel**:

$$\text{MultiHead}(Q, K, V) = Concat(head_1, \ldots head_h)W_O$$

Each head captures a different type of relationship:

- one can look at syntactic relationships (e.g. subject-verb)
- another semantics (e.g. drug-disease)

This gives **a multidimensional view** of language.

Encoder — Understanding the meaning of text

- It analyzes the entire input sequence and builds **semantic representations** of the text.

- Process **all tokens in parallel** thanks to *self-attention*, modeling the relationships between words.

- It produces **contextual embeddings**, i.e., vectors that capture the meaning of each word in context.

In summary: the Encoder **understands the input message**, creating the numerical base on which the Decoder will generate the output.

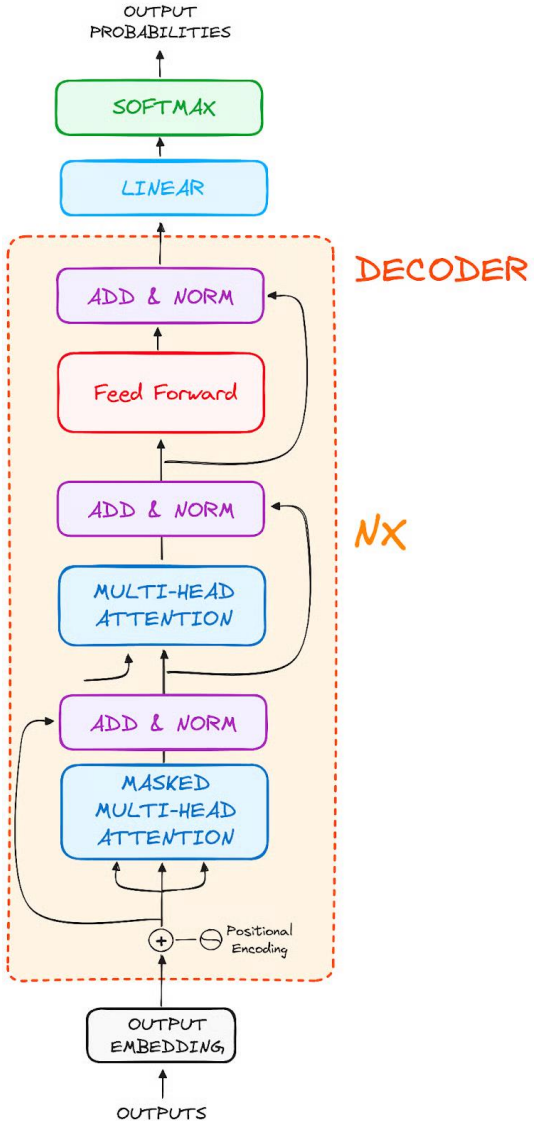
Decoder — Generate Coherent Sequences

- It uses the Encoder representations to **generate one word at a time** (autoregressive mode).

- Employ two attentions:
  - **Masked Self-Attention:** only looks at words that have already been generated.
  - **Cross-Attention:** links each generated word to the relevant parts of the input (Encoder).

- Each new token depends on the **previous context** and the **content understood** by the encoder.

In summary: the Decoder **translates comprehension into language**, producing coherent and fluid text.



Cross-Attention — The Bridge Between Understanding and Generation

- The **Decoder** uses **Cross-Attention** to connect with the **Encoder** and retrieve information from the input text.

- At this stage,
  - the **Queries (Q)** come from the *Decoder* → represent **what he wants to know** to generate the next word;
  - the **Keys (K)** and **Value (V)** come from the *Encoder* → represent **what the input contains** and **its meaning**.

- The Decoder compares each Query with all the Encoder Keys to calculate *how relevant each input word is* and combines the corresponding Values into a **context vector**.

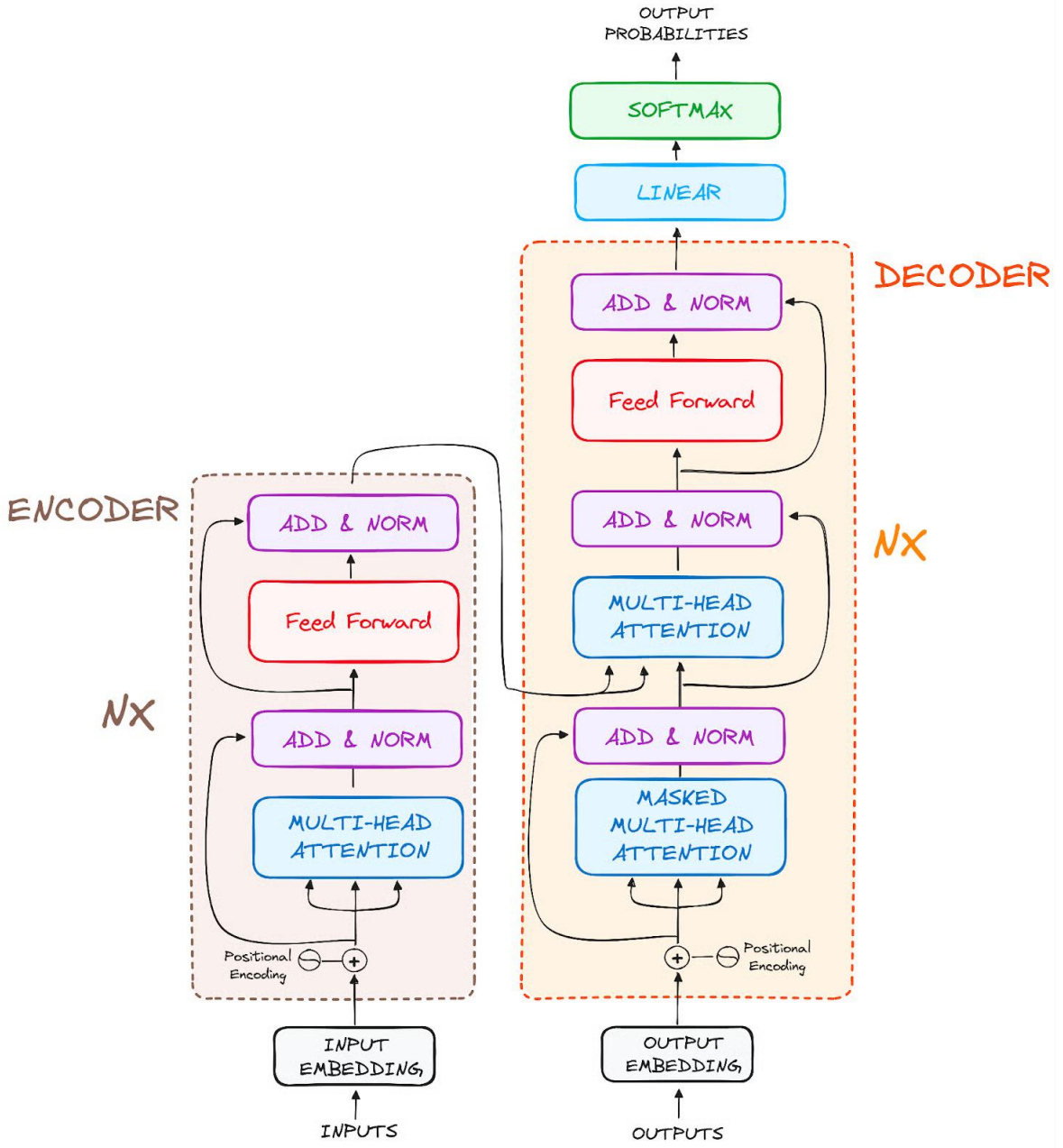
- In summary: Cross-Attention is the **heart of the Transformer encoder-decoder**, because it allows the model to dynamically "interrogate" the input and integrate the understanding of the Encoder into the text generation.

Transformer: Overall Architecture

The **Transformer** consists of two main blocks:

Encoder: Processes the input sequence, capturing the relationships between tokens.

Decoder: generates the output sequence, one word at a time, using the information learned from the Encoder.



General Transformer Flow

- **Input embedding + positional encoding** → numerical representation of tokens.
- **Encoder** → learns meaning and global semantic relationships.
- **Decoder** → generates text progressively, consulting the Encoder via cross-attention.
- **Layer Output** → converts the last state into a probability for the next word.

In summary:

- **Encoder = language understanding**
- **Decoder = language generation**
- **Cross-Attention = integration between the two processes**

Dai Transformer ai Large Language Models (LLM)

- **Large Language Models (LLMs)** use **the same Transformer architecture**, but on a much larger scale.

- Trained on **billions of words**, they automatically learn the **structure, meaning, and relationships** of language.

- They can **understand and generate text** in a general way, without specific re-training.

In short: **an LLM is a "giant" Transformer trained on huge amounts of data.**

How an LLM works

- **Input → Tokenization**
  Text is divided into units (tokens) and converted into numeric vectors.

- **Transformer →Block Processing**
  Each token interacts with the others via *self-attention* to construct context.

- **Output → Autoregressive prediction**
  The model predicts one word at a time based on the previous ones.

The principle remains that of the Transformer, but on millions of parameters and texts.

From training to using an LLM

- LLMs are built and used through **three fundamental phases**, which correspond to three levels of language knowledge.

1. Pre-training

- It is the initial phase, the most expensive and the longest.
- The model is trained on **huge amounts of unlabeled text** (books, articles, web, PubMed…).
- The goal is very simple but powerful: **to predict the next word** (*next-token prediction*).
- This allows the model to automatically learn:
  - grammar rules,
  - semantic relationships between words,
  - general knowledge of the world and texts.

After pre-training, the LLM can "understand" and "write" text, but still in a generic way.

From training to using an LLM

2. Fine-tuning

- In this phase, the model is **specialized** on a task or domain.
- It retrains (usually on smaller but targeted datasets) to adapt to a technical language or a specific goal:
  - *e.g.* PubMedBERT → biomedical texts,
  - *e.g.* BioGPT → clinical questions and scientific papers.
- Several strategies can be applied:
  - **Supervised fine-tuning:** with labeled examples,
  - **Instruction tuning / RLHF:** to learn how to follow human instructions and respond more useful or safely.

Fine-tuning transforms a "general" model into a "specialized" model.

From training to using an LLM

3. Inference (practical use)

- Once trained, the LLM no longer needs to be re-trained.
- After training, he has learned the rules of language and can be **queried directly** with a *text prompt* (a question or instruction in natural language).
- The model responds by generating the most likely sequence of words based on context.

Example:

Prompt: "Summarize in plain language an article on BRCA1 mutations."

Output: "The BRCA1 gene is involved in DNA repair; Some mutations increase the risk of cancer."

LLM for Biomedical Language

- Biomedical **language models** are **LLMs based on Transformer architecture**, pre-trained on **scientific and clinical texts** instead of generic data.

- Their aim is to understand medical terminology, the relationships between genes, diseases, drugs and experimental results.

Why you need dedicated templates:

- medical language is **rich in specialized terms** and acronyms;
- many concepts have **meanings other than** common language;
- models trained on **reliable sources (PubMed, clinical trials, EHR) are needed**.

Main biomedical LLM families

Model	Base	Domain / Data	Main applications
BioBERT	BERT	PubMed and PMC articles	NER, classification of relationships, extraction of concepts
PubMedBERT	BERT (from scratch)	PubMed Text Only	Understanding pure clinical language
ClinicalBERT	BERT	EHR and clinical notes	Analysis of health records and symptoms
BioGPT	GPT-2	Abstract e paper biomedici	Text generation and scientific Q&A
MedPaLM / MedPaLM-2	PaLM (Google)	Supervised clinical QA	Questions and answers in natural medical language
Galactica	Transformer decoder	Scientific articles, formulas, quotes	Abstract, quote, scientific reasoning

Section 4

LLM in Python

Hugging Face: The Language Model Ecosystem

Hugging Face is the go-to platform for working with language models (LLMs) and **Transformer architecture**.

Offers:

- **Hugging Face Hub** → repositories of already trained models (BioBERT, GPT, T5, etc.)
- **Transformers library** → Python interface to use these templates
- **Datasets, Tokenizers, Evaluate** → complementary libraries for data, encoding and metrics

Objective
To make the use and reuse of language models accessible **without having to train them from scratch**.

Hugging Face provides the infrastructure to move from Transformer theory to practice with just a few lines of code.

The General Flow of Hugging Face

Each language model in Hugging Face always follows the **same processing flow**:

```
Text → Tokenizer → Model → Interpreted Output
```

1. **Text (input)**
   → Sentence or document that we want to analyze or generate.
2. **Tokenizer**
   → Converts text to numbers (token ID).
3. **Transformer Model**
   → Calculates internal representations and produces predictions.
4. **Decode**
   → Translates model output into readable results (classes, text, responses).

Everything we studied (embedding, attention, decoder) happens within this flow — Hugging Face provides a simplified interface.

The `transformers library`

- The **`transformers library`** is the operational heart: it provides classes and functions to manage each phase of the NLP flow.

Component	Purpose	Example
AutoTokenizer	Convert text ↔ to numeric tokens	`tokenizer = AutoTokenizer.from_pretrained("BioBERT")`
AutoModel	Load the base model (representations only)	`model = AutoModel.from_pretrained("BioBERT")`
AutoModelForSequenceClassification	Upload model + head for classification	`model = AutoModelForSequenceClassification.from_pretrained("BioBERT")`
AutoModelForQuestionAnswering	Specific to Q&A tasks	`model = AutoModelForQuestionAnswering.from_pretrained("BioBERT")`
pipeline	NLP task-ready interface (tokenizer+tensors+decoding)	`qa = pipeline("question-answering", model="BioBERT")`

Each Car Class is a … automatically recognizes the correct configuration based on the chosen model.

The key elements of the flow

Tokenizer

```
tokenizer = AutoTokenizer.from_pretrained("dmis-
 lab/biobert-base-cased-v1.1")
```

- Load pre-trained **vocabulary and segmentation rules.**
- It divides the text into tokens and converts them into IDs.
- Returns PyTorch tensors (`return_tensors="pt"`) ready for the model.

The key elements of the flow

Model

```
model = AutoModelForSequenceClassification.from_pretrained("dmis-
  lab/biobert-base-cased-v1.1")
```

- Charge **Transformer architecture + pre-trained weights**.
- Performs the *forward pass*: calculates the output (e.g. logits, embeddings, probability).
- `The ForSequenceClassification` suffix indicates the *type of task* (e.g. classification, Q&A, generation…).

Each "Auto…" automatically understands the correct format of the chosen task.

Practical example: Text classification

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

tokenizer = AutoTokenizer.from_pretrained("dmis-lab/biobert-base-cased-v1.1")
model = AutoModelForSequenceClassification.from_pretrained("dmis-lab/biobert-base-
  cased-v1.1")

text = "Patient presents with chest pain"
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)
```

- The tokenizer → transforms text into numerical tensors
- The → model produces the output (logits or probability)
- We can then reverse engineer or use the pipeline to automate

This is the "manual" flow: perfect for understanding what goes on behind the scenes.

Practical example with `pipeline()`

```
from transformers import pipeline

classifier = pipeline("text-classification", model="dmis-lab/biobert-
  base-cased-v1.1")
classifier("Patient presents with chest pain")
```

- `pipeline()` incapsula **tokenizer + modello + output decoder**
- Automatically detects input and task types (QA, NER, classification, etc.)
- Returns **direct result**:

```
[{'label': 'disease_related', 'score': 0.94}]
```

It's the "out-of-the-box" mode, which is useful for quickly experimenting with biomedical LLMs.

What tasks can we perform

Task	Description	Example
text-classification	Label phrases or documents with a category	Input: “The report shows high glucose levels.” → Output: “Diabetic” (probabilità 0.92)
question-answering	Answering questions about a provided text	Q: “What gene is mutated in BRCA1 cancer?” Context: “BRCA1 mutations increase breast cancer risk.” → A: “BRCA1”
ner (Named Entity Recognition)	Recognize clinical entities (genes, diseases, drugs)	Input: “Aspirin reduces inflammation in arthritis.” → Output: {“Aspirin”: Drug, “arthritis”: Disease}
summarization	Summarizing scientific or clinical texts	Input: “Abstract di PubMed di 250 parole” → Output: “Aspirin reduces inflammation by inhibiting COX enzymes.”
text-generation	Generate coherent or explanatory text	Prompt: “Explain how aspirin works:” → Output: “Aspirin blocks prostaglandin synthesis, reducing pain and fever.”

All tasks use the same Transformer architecture: only the type of model *head* changes (AutoModelFor…).

The "head" of the model in the Transformers

- All Hugging Face models **use the same Transformer base** (encoder/decoder and attention mechanism).

- What **changes** from one task to another is the **head:**
  a **final layer** added on top of the base to adapt the model to the type of problem.

- **General architecture**

```
Input → Embedding → Encoder/Decoder (Transformer) → Testa specifica → Output
```

Examples of heads (`AutoModelFor...`)

Head	Function	Example Output
`AutoModelForSequenceClassification`	Classification of texts or sentences	Etiquette and probability
`AutoModelForQuestionAnswering`	Extracting responses from a context	Response phrase
`AutoModelForSeq2SeqLM`	Text generation (e.g. translation, summary)	Generated phrase
`AutoModelForCausalLM`	Generative autoregressive language	Continuous text, GPT type

The "body" of the model (Transformer) **understands the language**,
the "head" **adapts the output** to the specific task.
It's like adding an end module for each type of NLP application.

Biomedical models to use

Model	Basic architecture	Domain	Typical use
BioBERT	BERT	PubMed / PMC	classification, NER
PubMedBERT	BERT (from scratch)	PubMed puro	Semantic understanding
ClinicalBERT	BERT	clinical notes, EHR	Report analysis
BioGPT	GPT-2	Scientific texts	Q&A, generation
MedPaLM	Palm	Supervised QA	Natural language responses

These models are already optimized for biomedical data and load with `.from_pretrained()`.

From theory to practice

Now that we understand:

- how a **Transformer** works (embedding → attention → decoder),
- how Hugging Face makes it easy to use via **AutoTokenizer, AutoModel, and pipeline**,
- and like every model is specialized through its own **head (head),**

We can finally **put everything into practice** in the notebook:
`Biomedical_LLMs.ipynb`