# Profili differenti per un Assistente AI

Prof. Ing. Loris Penserini, PhD

https://orcid.org/0009-0008-6157-0396

# Overview

Parleremo di:

- Introduzione ai modelli LLM in pratica con TensorFlow
- Assistente AI con «Ollama»
- IoT con smart plugs: Shelly & Tapo
- Comandi vocali con «pyttsx3» e Ollama
- UI per Python con QT Designer

**Creare un assistente AI con capacità di giocare ruoli diversi**

# LLM cosa è…

Quasi tutti i moderni LLM (come GPT, LLaMA, BERT, Claude, Gemini…) si basano sull'architettura Transformer introdotta da Google nel 2017.

**Daremo in seguito una rappresentazione strutturata e precisa del funzionamento interno di un LLM come BERT e LLaMA.**

# Fasi principali in un LLM

| N | Fase | Descrizione |
|---|---|---|
| 1 | Creazione del vocabolario | Lista di token unici usati dal modello (parole, sottoparti, simboli, subword) |
| 2 | Tokenizzazione | Frase → sequenza di token usando il vocabolario e una tokenizer (es. WordPiece, BPE) |
| 3 | Embedding | Ogni token viene convertito in un vettore numerico (es: 768 dimensioni in BERT-base) |
| 4 | Aggiunta di positional embedding | Aggiunge al vettore di ogni token un'informazione sulla sua posizione nella frase |
| 5 | Costruzione matrici Q, K, V | Ogni token genera 3 vettori: Q (query), K (key), V (value), via layer lineari appresi |
| 6 | Calcolo dell'attenzione | Per ogni token: $\text{softmax}(Q \cdot K^T / \sqrt{d})$ → pesi di attenzione → combinati con V per creare nuova rappresentazione |
| 7 | Multi-head Attention | 12 (o più) teste indipendenti calcolano attenzioni diverse → concatenate e fuse con layer lineare |
| | | |

# Fasi principali in un LLM

| N | Fase | Descrizione |
|---|---|---|
| 8 | Feed-Forward Layer | Dopo l'attenzione, ogni token passa in una rete neurale FFN condivisa |
| 9 | Layer Norm + Residual | Normalizzazione e somma con input originale per stabilizzare la rete |
| | **Ripetizione dei layer** | **I passaggi 5–9 si ripetono per ogni layer (es. 12 volte per BERT-base)** |
| 10 | Output finale (embedding) | L'ultimo layer produce una rappresentazione semantica ricca per ogni token |
| 11 | Task-specific head | In base al task (es. classificazione, generazione, QA): aggiunta di output layer specifico |

# Il «Transformer»

Il **Transformer** definisce **come le informazioni scorrono nel modello**, e include principalmente:

- **Self-Attention**: ogni token nella sequenza può "guardare" tutti gli altri token e pesare la loro importanza per capire il contesto.
- **Multi-Head Attention**: più meccanismi di self-attention paralleli per catturare diversi tipi di relazione tra i token.
- **Feed-Forward Layer (FFL)**: una rete neurale densa che elabora ogni token in maniera indipendente.
- **Layer Norm e Residual Connections**: aiutano il modello a convergere più velocemente e a stabilizzare l'apprendimento.
- **Stacking dei layer**: il blocco di cui sopra viene ripetuto N volte (dipende dalla dimensione del modello).

Quindi quando diciamo **"Transformer"**, parliamo principalmente di questa **architettura interna**, cioè dei blocchi di attenzione e feed-forward che vengono ripetuti nei layer.

# Struttura generale

Struttura generale del blocco Transformer (Encoder o Decoder):

- Ogni Transformer layer (es. in BERT o GPT) è composto da due sotto-moduli principali:
  - Self-Attention / Multi-Head
  - AttentionFeed-Forward Layer (FFL)
- Entrambi sono avvolti da:
  - Residual Connection (connessione residua)
  - Layer Normalization

# Feed-Forward Layer (FFL)

È una **rete neurale completamente connessa (dense)** applicata **a ogni token in modo indipendente**.

Formula di base, se indichiamo con $x_i$ il vettore di un token (dimensione $d_{model}$), la FFL calcola:

$$\mathbf{FFL(xi) = W_2 \cdot f(W_1 \cdot x_i + b_1) + b2}$$ dove:

- $\mathbf{W_1}$ è una matrice di pesi di dimensione $d_{model} \times d_{ff}$
- $\mathbf{W_2}$ è una matrice di pesi di dimensione $d_{ff} \times d_{model}$
- $\mathbf{f(\cdot)}$ è una funzione di attivazione non lineare (ReLU o GELU)
- $\mathbf{d_{ff}}$ è la **dimensione interna**, di solito **più grande** di $d_{model}$ (ad esempio: 768 → 3072)

# Funzionamento intuitivo

- Dopo la **self-attention**, ogni token ha già "visto" gli altri e incorporato il contesto.
- Ora la **FFL** trasforma quel vettore per raffinarlo, come se "riformattasse" l'informazione dentro ogni token.
- Questa trasformazione è identica per tutti i token, ma applicata separatamente (cioè in parallelo su tutti i token).

**Si può pensare come a un piccolo MLP (Multi-Layer Perceptron) che lavora token per token, migliorando la rappresentazione di ciascuno.**

# Applicare la FFL significa...

La **Self-Attention** da sola è lineare: mescola i token, ma non introduce non-linearità forti.

La **FFL** serve quindi a:

- Introdurre **capacità non lineare** nel modello.
- Permettere al modello di **trasformare** o **rimappare** le informazioni già integrate dal meccanismo di attenzione.
- Dare **profondità e potere espressivo** al layer — in effetti, quasi tutta la "potenza di rappresentazione" deriva proprio da questi FFL ripetuti nei vari strati.

# Funzione di attivazione non lineare

In una rete neurale (come la Feed-Forward Layer del Transformer), ogni neurone esegue due passi:

- Combina linearmente i valori in ingresso

$$W \cdot x + b$$

(una semplice moltiplicazione per pesi + somma)

- Applica una funzione di attivazione che trasforma quel valore in modo **non lineare**.

# Funzioni: lineari vs non lineari

**Una funzione lineare** come: $y = ax + b$ può solo: scalare, traslare, ruotare lo spazio dei dati.

Ma se tutti gli strati della rete fossero lineari, allora tutta la rete sarebbe equivalente a una sola matrice. Cioè, inutile! Non imparerebbe nulla di complesso.

**Le funzioni non lineari**: piegano, curvano, distorcono lo spazio dei dati e permettono alla rete di imparare concetti come: astrazioni, categorie, interazioni complesse, pattern linguistici.

# Attivazioni usate nei Transformer...

## ReLU (Rectified Linear Unit)

$f(x) = \max(0, x)$

Esempio:

se $x = -2$ $\rightarrow$ output = 0

se $x = 3$ $\rightarrow$ output = 3

È semplice, veloce, efficace.

# Attivazioni usate nei Transformer...

## GELU (Gaussian Error Linear Unit)

È l'attivazione usata nei modelli più avanzati (BERT, GPT, Vision Transformer, ecc.).

La formula approssimata: $\boldsymbol{f(x) = x \cdot \Phi(x)}$ dove $\Phi(x)$ è la CDF (Cumulative Distribution Function) di una normale (Gaussiana standard).

Non serve memorizzarla: intuitivamente, non taglia i valori negativi come ReLU, ma li smorza con una curva morbida.

# Attivazioni usate nei Transformer...

GELU (Gaussian Error Linear Unit) è definita (in forma semplificata):

$$\mathbf{GELU}(x) = x \cdot \Phi(x)$$

Quindi:

- Se $x$ è negativo → Φ(x) è vicino a 0 → output ≈ 0
- Se $x$ è moderato → Φ(x) è fra 0 e 1 → output è un valore "attutito"
- Se $x$ è grande → Φ(x) ≈ 1 → output ≈ x

Questo rende la GELU una ReLU "morbida e probabilistica": invece di "tagliare" di colpo, decide quanto far passare in base a una probabilità.

# Processo di «embedding»

**E' la trasformazione di una parola in un vettore numerico**.

Esistono dizionari (Word2Vect, GloVe, FastText, BERT, ecc.) pre-addestrati dove ogni parola ha già assegnato un vettore:

**'ciao' → [0.14, -0.33, ..., 0.92] (dim=300)**

Vediamo un esempio di processo di embedding utilizzando il framework **Keras** che è una potente libreria opensource di deep learning ad alto livello, progettata per essere semplice, modulare e facile da usare, soprattutto per chi lavora con TensorFlow.

Dal 2017, Keras è parte ufficiale di TensorFlow, e la versione più usata oggi è **tf.keras** (Keras come frontend di TensorFlow). Keras fornisce un'interfaccia user-friendly che permette di definire/addestrare modelli di reti neurali di deep learning in modo dichiarativo, leggibile e veloce, anche per chi non è un esperto.

# Esempio di «embedding»

```
import tensorflow as tf

vocab = {'ciao': 0, 'come': 1, 'stai': 2}
indice_ciao = tf.constant([[vocab['ciao']]])  # [[0]]

embedding = tf.keras.layers.Embedding(input_dim=3, output_dim=8)  # 3 parole, 8 dimensioni
vet_ciao = embedding(indice_ciao)

print(vet_ciao)
```

**Output**:

**vet_ciao** → [[[ 0.12 -0.34  0.58 ... ]]] ← vettore di 8 numeri, inizializzati random e aggiornati durante l'addestramento

# Esempio di «embedding»

```
embedding_matrix = [
    [ 0.12, -0.34, ..., 0.21],   # ciao (id 0)
    [-0.09,  0.57, ..., 0.38],   # come (id 1)
    [ 0.44, -0.19, ..., 0.76],   # stai (id 2)
]
```

Per cui, **embedding(0)** → corrisponde alla prima riga della matrice, cioè è la rappresentazione della parola «ciao».

Durante l'allenamento, la rete modifica i vettori di embedding per rappresentare meglio le **relazioni** tra parole (**semantiche**, **sintattiche**, ecc.). Vedremo che lo fa attraverso i layer delle HEADS…

# Esempio: in sintesi…

| Parola | Fase | Risultato |
|---|---|---|
| `"ciao"` | → indice | `0` |
| `0` | → embedding | `[0.12, -0.34, ..., 0.21]` |
| embedding | → input a rete | Usato per calcolare Q, K, V… |

Per cui, in questo esempio abbiamo visto queste tre fasi, necessarie all'apprendimento di una rete neurale di tipo deep learning.

**Qual è l'obiettivo che si vuole ottenere?**

# L'Attenzione Neurale

La trasformazione delle parole/token in tensori numerici, facilita la logica della rete neurale…

| Ruolo | Significato «umano» |
|---|---|
| **Query** | Il token che sta cercando informazioni |
| **Key** | Il token che potrebbe offrire informazioni |
| **Value** | I dati veri che vengono combinati |

**Si crea un meccanismo in cui ogni token di una frase "guarda" gli altri token per capire da chi prendere informazione.**

# Complichiamo l'esempio…

Esempio completo su GitHub, testato su Colab.

```
# Frase di input
frase = "Il gatto mangia il topo."

# Tokenizzazione corretta (senza offset_mapping)
inputs = tokenizer(frase, return_tensors="tf", add_special_tokens=True)
input_ids = inputs["input_ids"][0].numpy()
tokens = tokenizer.convert_ids_to_tokens(input_ids)
outputs = model(**inputs)
attention = outputs.attentions[-1][0][0].numpy()  # ultimo layer, head 0
```

Analizziamo il codice:

**input_ids = inputs["input_ids"][0].numpy()**

è un passaggio cruciale per ottenere la sequenza di ID numerici (token ID) corrispondente alla frase di input.

# Complichiamo l'esempio…

Dettagli:

```
input_ids = inputs["input_ids"][0].numpy()
```

**inputs["input_ids"]**: estrae solo il tensore con gli ID dei token

**[0]**: prende la prima (e unica) sequenza, perché anche una singola frase viene codificata come batch di 1 riga

**.numpy()**: converte il tensore TensorFlow in un array NumPy (necessario se si vuole usare con librerie come matplotlib, numpy, ecc.)

# Complichiamo l'esempio…

```
inputs = tokenizer(frase, return_tensors="tf", add_special_tokens=True)
```

Questa istruzione tokenizza la frase (es. "Il gatto mangia il topo.") e restituisce un dizionario TensorFlow:

```
inputs = {
    'input_ids': tf.Tensor([[101, 1110, 16021, 17020, 18747, 12078, 1110,
 18456, 12898, 119, 102]]),
    'token_type_ids': ...,
    'attention_mask': ...
}
```

Questi numeri sono gli ID dei token nel **vocabolario di BERT**.

**Ricordate, qual è l'obiettivo di queste trasformazioni di frasi in tensori composti da parole/token numerici?**

# Obiettivo: «La self-attention»

È un meccanismo in cui ogni token di una frase «guarda» gli altri token per capire da chi prendere informazione. Per farlo, ogni token genera 3 vettori:

**Query** (chi osserva)

**Key** (chi viene osservato)

**Value** (il contenuto da prendere)

# La self-attention

Esempio pratico: «**Il gatto mangia il topo**»

Supponiamo che **Query** = **«mangia»**

Cosa fa la rete neurale? Il token «mangia» genera un **Query vector**

Questo «Query» viene confrontato (dot product) con i Key di tutti gli altri token: 'Il', 'gatto', 'mangia', 'il', 'topo'.

Più è alto il punteggio Query·Key, più **«mangia»** dà attenzione a quel token.

Se 'mangia' somiglia molto a 'gatto' nel contesto:

**punteggio alto → più attenzione → 'gatto' avrà più peso nel calcolo finale**.

# Matrice di «ATTENZIONE»

«**attn_weights**» è la matrice di attenzione [5×5] che andremo a visualizzare.

```
attn_weights = tf.nn.softmax(scores, axis=1)  [1]
```

Nella [1] si applica «softmax» per normalizzare i punteggi e ottenere i pesi di attenzione. **Questi dicono quanta attenzione ogni token dà agli altri**.

```
output = tf.matmul(attn_weights, V)    [2]
```

Calcola l'output come combinazione pesata di V

# Matrice di «ATTENZIONE»

Il tokenizer di BERT (wordPiece) spezza le parole in sotto-componenti chiamati **subtoken**. Questo accade quando una parola non è presente esattamente nel vocabolario del modello.

Il simbolo **##** indica che quel token è la continuazione di un token precedente.

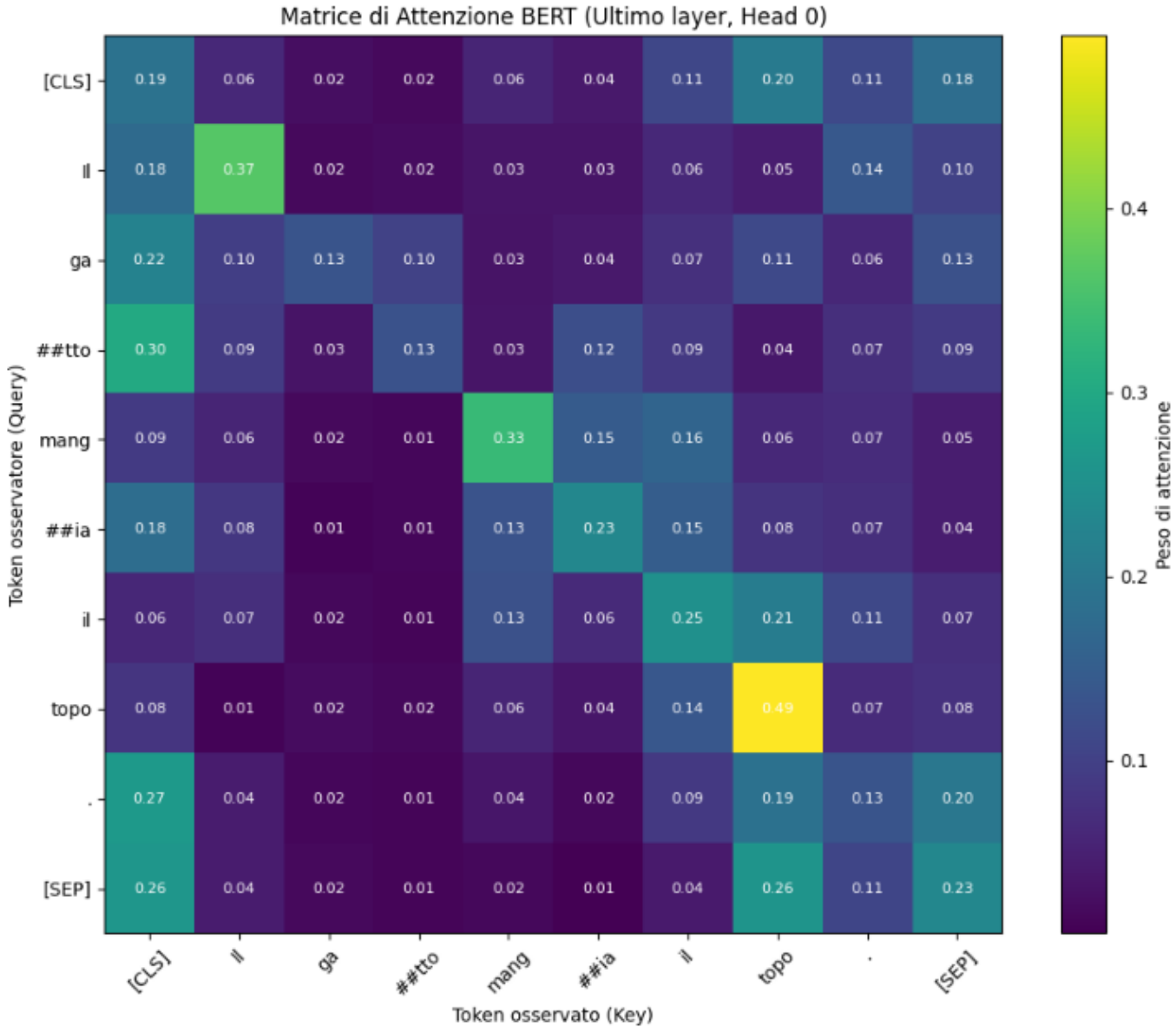
Questo approccio permette a BERT di:

- **gestire parole rare o nuove**
- **ridurre la dimensione del vocabolario**
- **generalizzare meglio su parole mai viste**

```
Token BERT: ['[CLS]', 'Il', 'ga', '##tto', 'mang', '##ia', 'il', 'topo', '.', '[SEP]']
Token ricostruiti: ['[CLS]', 'Il', 'gatto', 'mangia', 'il', 'topo', '.', '[SEP]']

🔹 Dimensione embedding finale: (10, 768)

📌 Esempio dimensioni Q, K, V:
Q: (10, 768)
K: (10, 768)
V: (10, 768)
```



# Multi-head attention

Nel modello BET-base il numero dei layer della rete è 12 così come i layer delle intestazioni/heads per avere i vantaggi di un approccio **Multi-head attention**:

- Permette di parallelizzare diverse prospettive
- Ogni head esplora relazioni diverse
- Migliora la capacità di rappresentare il contesto

Per esempio, osservare una frase con 12 lenti significa, per es. poter usarne una per capire chi parla, una seconda per individuare i verbi, una terza per calcolare le distanze semantiche tra le parole, ecc. che messe tutte assieme producono una comprensione potente!

# Tipi di attention

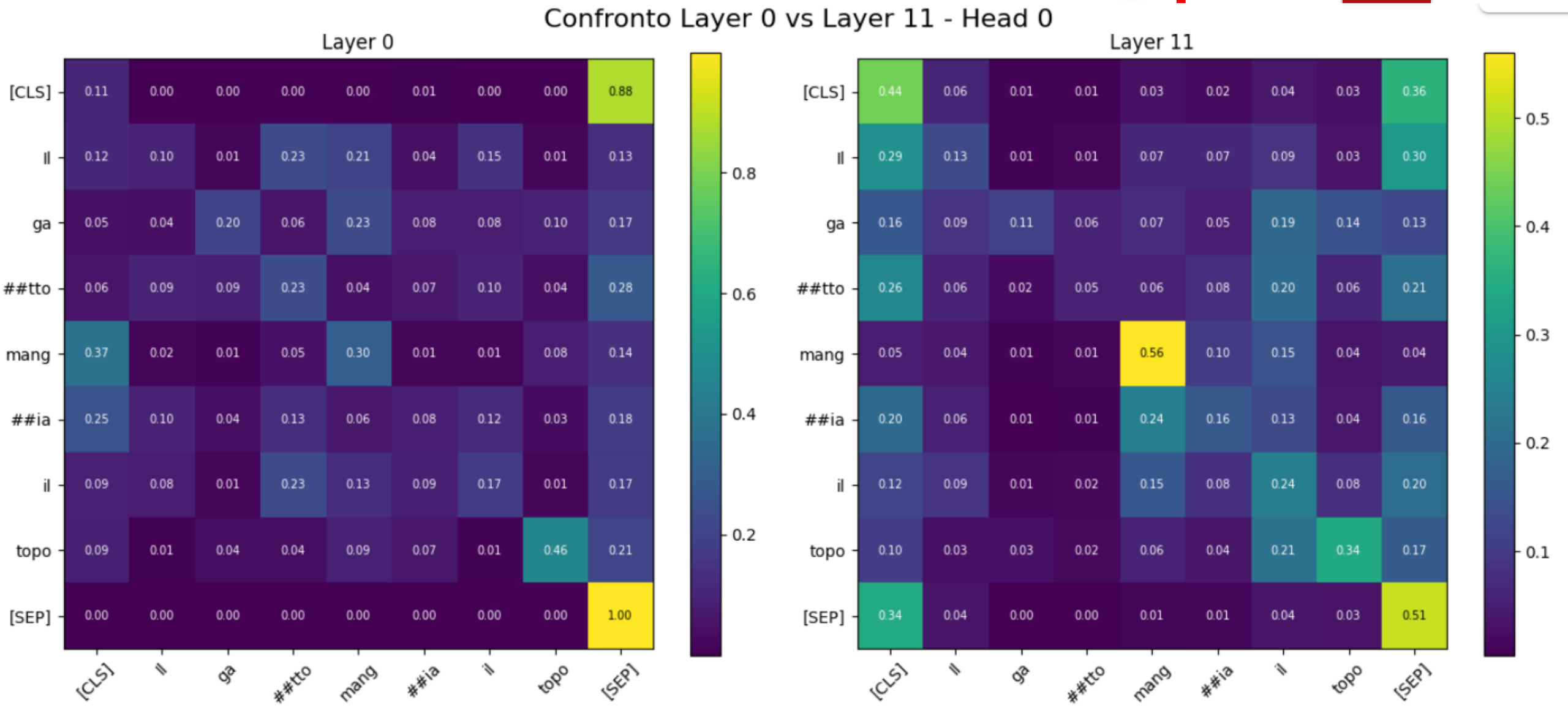
Differenze tra **encoder** e **decoder**:

- **BERT** (encoder-only): attenzione bidirezionale
- **GPT** (decoder-only): attenzione causale (solo verso il passato)

Se si vuole combinare le capacità di entrambi (ad esempio, per un modello che sia capace di comprendere e generare testo), **modelli più complessi come T5 o BART** vengono in aiuto, **poiché combinano elementi sia di encoder che di decoder, utilizzando un'architettura più versatile**.

Esempi con rete BERT e con layer regolabili disponibili su GitHub.

# Matrice di «ATTENZIONE»: heatmap



# State of Art in practical deep learning…

**PyTorch** e **TensorFlow** sono le due librerie più popolari per il deep learning, e vengono spesso usate per costruire modelli come CNN, RNN, e Transformers (inclusi i LLM).

Anche se fanno cose simili, ci sono differenze chiave in filosofia, sintassi e uso pratico.

# Prototipare algoritmi complessi

Recenti strumenti come **Jupyter Notebook**, **Google Colab**, **Spider**, ecc., assieme ai GPT, forniscono un fantastico ambiente dove sviluppare codice velocemente limitando il tempo necessario a risolvere problematiche di integrazione di tecnologie. Tuttavia, quest'ultimo aspetto potrebbe essere visto negativamente se l'obiettivo è acquisire competenze di programmazione ed integrazione di tecnologie.

**I notebook dovrebbero essere trattati come dei veri e propri quaderni: ci si abbozza qualcosa, si provano diverse iterazioni, ma quando è pronto per essere formalizzato, si pulisce un po' e si mette in un file py da gestire con un IDE tradizionale.**

# Testare algoritmi complessi

Ho sfruttato **Google Colab** per risolvere facilmente errori legati a:

- Installazione/conflitti di librerie Python (transformers, tf-keras, ecc.)
- Compatibilità tra keras, tensorflow e transformers
- Mancanza di pacchetti come matplotlib

**Google Colab** ha già preinstallati:

- tensorflow (compatibile con tf.keras)
- keras 2.x
- Possibilità di installare transformers con un semplice !pip install.

Alcuni difetti sono:

- Non aiuta ad acquisire competenze per lo sviluppo di software autonomo
- Propone funzionalità di AI che limitano l'apprendimento

# PyTorch vs TensorFlow

| Caratteristica | PyTorch | TensorFlow |
|---|---|---|
| 🔁 Esecuzione | *Dinamica* ("define-by-run") | *Statica* (grafi computazionali, v1) |
| 🧠 Facilità di debug | Più semplice (integrazione diretta con Python) | Più complicata (specie in TF 1.x) |
| 📜 Sintassi | Più "pythonic", leggibile | Più verbosa, simile a API di basso livello |
| 🧪 Popolarità in ricerca | Molto usato in ambito accademico | Dominante in produzione industriale |
| 📦 Librerie estese | TorchVision, TorchText, Hugging Face, etc. | Keras, TF Lite, TF Serving, TF Hub, etc. |
| 🛠️ Deployment | Meno ottimizzato per mobile/web | Più maturo su mobile (TensorFlow Lite, JS) |
| 🤖 Interfaccia ad alto livello | Nativa (modularità via nn.Module) | Keras integrato (in TF 2.x) |
| 📈 Grafi statici | No (ma TorchScript è un'opzione) | Sì, ottimizzabili (TensorFlow 2 include eager) |

# Project Work-1

Questo è un caso d'uso molto concreto e utile per:

**interpretare relazioni semantiche tra parole, escludendo i sub-token tecnici che derivano dalla tokenizzazione WordPiece di BERT** (come ##tto, ##ia, ecc.).

# Project Work-1

Supponiamo che i token siano:

**tokens = ['[CLS]', 'Il', 'ga', '##tto', 'mang', '##ia', 'il', 'top', '##o', '.', '[SEP]']**

Questi corrispondono a:

**parole = ['[CLS]', 'Il', 'gatto', 'mangia', 'il', 'topo', '.', '[SEP]']**

E le relazioni che ti interessano sono, ad esempio:

**"gatto" → "mangia"**

**"mangia" → "topo"**

Obiettivo:

Calcolare una **matrice di attenzione** compatta tra queste parole, escludendo i dettagli dei subtoken.

# Soluzione al Project Work-1

Serve una mappatura da tokens a parole. Ad esempio:

**token_to_word = [0, 1, 2, 2, 3, 3, 4, 5, 5, 6, 7]**

Significa che:

**'ga' e '##tto' → parola 'gatto' (indice 2)**

**'mang' e '##ia' → 'mangia' (indice 3)**

**'top' e '##o' → 'topo' (indice 5)**

**Aprire Google Colab:** https://colab.research.google.com/

**E giocare con gli esempi presenti qui:**

**https://github.com/penserini/ITS_AI_Specialist/tree/main/GoogleColab**

# Soluzione al Project Work-1

Comprimere la matrice di attenzione. Supponiamo di avere:

**attention_full[i][j] = attenzione da tokens[i] a tokens[j]**

Allora, per ottenere la matrice compatta (A_compact) tra parole:

**A_compact[p_i][p_j] = media (A_full[i][j] for i in token_indices[p_i] and j in token_indices[p_j])**

Con le due matrici:

**A_compatta[i][j]** → somma dei pesi di attenzione da parola i a parola j

**counts[i][j]** → quante volte quella somma è stata aggiornata

(cioè, quanti sub-token della parola i hanno puntato a quanti sub-token della parola j)

Si calcola la media:

**A_compatta[i][j] = A_compatta[i][j] / counts[i][j]**

Poi viene gestita la divisione per zero: np.where(counts == 0, 1, counts)

# Debug del Project Work-1

sol_projectwork_1.ipynb

File Modifica Visualizza Inserisci Runtime Strumenti Guida

Condividi Gemini

Comandi + Codice + Testo Esegui tutte

RAM Disco

```
outputs = model(**inputs)
attention = outputs.attentions[-1][0][0].nu

# Ricostruzione parole e mapping
parole = []
token_to_word = []
temp = ""
word_idx = -1
for tok in tokens:
    if tok.startswith("##"):
        temp += tok[2:]
    else:
        if temp:
            parole.append(temp)
        temp = tok
        word_idx += 1
    token_to_word.append(word_idx)
if temp:
    parole.append(temp)

# Matrice compatta
N = max(token_to_word) + 1
A_compatta = np.zeros((N, N))
counts = np.zeros((N, N))

for i in range(len(tokens)):
    for j in range(len(tokens)):
        wi = token_to_word[i]
        wj = token_to_word[j]
        A_compatta[wi][wj] += attention[i][j]
        counts[wi][wj] += 1

A_compatta /= np.where(counts == 0, 1, counts)

# Visualizza
plt.figure(figsize=(8, 6))
im = plt.imshow(A_compatta, cmap="viridis")
plt.colorbar(im, label="Peso di attenzione")
plt.xticks(np.arange(N), parole, rotation=45)
plt.yticks(np.arange(N), parole)
plt.title("Matrice di attenzione compatta (BERT)")
for i in range(N):
    for j in range(N):
        plt.text(j, i, f"{A_compatta[i, j]:.2f}", ha="center", va="center",
                 color="white" if A_compatta[i, j] < 0.5 else "black")
plt.tight_layout()
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarn
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (ht
You will be able to reuse this secret in all of your notebooks.
```

Variabili

Il controllo variabili potrebbe influire sulle prestazioni di runtime quando è aperto.

| Nome | Tipo | Forma | Valore |
|---|---|---|---|
| A_compatta | ndarray | (8, 8) | array([[0.1899409 , 0.06232141, 0.02160524, 0.04852103, 0.10982046, 0.20431143, 0.11317297, 0.18018031], [0.17548537, 0.36558774, 0.0199459 , 0.03347283, 0.06271099, 0.04600535, 0.14062132, 0.102751 |
| N | | | |
| attenti | | | 6, 0.05727906, 0.039763 , 0.10982046, 0.20431143, 0.11317297, 0.18018031], [0.17548537, 0.36558774, 0.01807061, 0.02182119, 0.03295397, 0.0339916 |
| counts | | | ., 2., 4., 4., 2., 2., 2., 2.], [2., 2., 4., 4., 2., 2., 2., 2.], [1., 1., 2., 2., 1., 1., 1., 1.], [1., 1., 2., 2., 1., 1., 1., 1.], [1., 1., 2., 2., 1., 1., 1., 1.], [1., 1., 2., 2., 1., 1., 1., 1.]]) |
| frase | | | |
| i | | | |
| im | | | > |
| input_ | | | 38127, 119, 102], dtype=int32) |
| inputs | | | y= array([[ 101, 10282, 11887, 15141, 23912, 10280, 10154, 88127, 119, 102]], dtype=int32)>, 'token_type_ids': <tf.Tensor: shape=(1, 10), dtype=int32, numpy= |
| j | | | |
| model | | | el object at 0x7e4f95ca0350> |
| model | | | |
| output | | | st_hidden_state=<tf.Tensor: shape=(1, 10, 768), dtype=float32, numpy= array([[[ 0.12642545, 0.17772663, -0.15400335, ..., 0.1338923 , 0.02114273, 0.12805 |
| parole | | | |
| temp | | | |
| tok | | | |
| token_ | | | |
| tokeni | | | -cased', vocab_size=119547, model_max_length=512, is_fast=True, padding_side='right', truncation_side='right', special_tokens={'unk_token': '[UNK]', 'sep_tol |
| tokens | | | |
| wi | | | |
| wj | | | |
| word_ | | | |

Tipo:ndarray
Forma:(8, 8)
Valore:array([[0.1899409 , 0.06232141, 0.02160524, 0.04852103, 0.10982046, 0.20431143, 0.11317297, 0.18018031], [0.17548537, 0.36558774, 0.0199459 , 0.03347283, 0.06271099, 0.04600535, 0.14062132, 0.10275175], [0.2622245 , 0.09367029, 0.09916957, 0.05747781, 0.08008617, 0.07619891, 0.0662941 , 0.10823125], [0.13474955, 0.07106299, 0.01475967, 0.21079125, 0.15656791, 0.07148875, 0.06973313, 0.04529589], [0.05981843, 0.07232931, 0.01535567, 0.09536526, 0.2500014 , 0.21113461, 0.11248748, 0.07278696], [0.08218027, 0.01158334, 0.01990295, 0.04573878, 0.13821825, 0.49356318, 0.06708743, 0.07608396], [0.26885846, 0.0423466 , 0.01443583, 0.02546899, 0.08788105, 0.18880385, 0.13329664, 0.19900377], [0.26291698, 0.04335932, 0.01608849, 0.01139174, 0.0407665 , 0.25775978, 0.10770892, 0.23252802]])

```
array([[0.1899409 , 0.06232141, 0.02160524, 0.04852103, 0.10982046,
        0.20431143, 0.11317297, 0.18018031],
       [0.17548537, 0.36558774, 0.0199459 , 0.03347283, 0.06271099,
        0.04600535, 0.14062132, 0.10275175],
       [0.2622245 , 0.09367029, 0.09916957, 0.05747781, 0.08008617,
        0.07619891, 0.0662941 , 0.10823125],
       [0.13474955, 0.07106299, 0.01475967, 0.21079125, 0.15656791,
        0.07148875, 0.06973313, 0.04529589],
       [0.05981843, 0.07232931, 0.01535567, 0.09536526, 0.2500014 ,
        0.21113461, 0.11248748, 0.07278696],
       [0.08218027, 0.01158334, 0.01990295, 0.04573878, 0.13821825,
        0.49356318, 0.06708743, 0.07608396],
       [0.26885846, 0.0423466 , 0.01443583, 0.02546899, 0.08788105,
        0.18880385, 0.13329664, 0.19900377],
       [0.26291698, 0.04335932, 0.01608849, 0.01139174, 0.0407665 ,
        0.25775978, 0.10770892, 0.23252802]])
```

Filtra variabili

Mostra valori nell'editor di codice
Quando lo strumento di controllo è aperto

Variabili Terminale

09:44 Python 3

# Bibliografia

**[Raschka et al., book 2021]** Sebastian Raschka, Vahid Mirjalili, «Machine Learning con Python – Nuova edizione – Costruire algoritmi per generare conoscenza», Apogeo. ISBN: 978-88-503-3524-4. 2021.

**[Lambert, book 2024]** Kenneth A. Lambert, «Programmazione in Python – Terza edizione», Apogeo education – Maggioli Editore. ISBN: 978-88-916-7143-1. 2024.

**[Penserini, GitHub 2025]** Loris Penserini, «Corso ITS Academy Turismo Marche: AI Specialist», https://github.com/penserini/ITS_AI_Specialist, 2025.