**1) 📊 Compact diagram of Transformer Encoder–Decoder**

Here’s a simple ASCII-style diagram showing the **data flow**:

Input Sentence (e.g. "Je suis étudiant")

│

[Token Embeddings + Positional Encoding]

│

┌─────────▼─────────┐

│ Encoder Layer │ (× N layers)

│ │

│ ┌─────────────┐ │

│ │ Self-Attn │◄─┤ each token attends to all others

│ └─────────────┘ │

│ │ │

│ Add + Norm │

│ │ │

│ ┌─────────────┐ │

│ │ FeedForward │ │ per-token MLP

│ └─────────────┘ │

│ │ │

│ Add + Norm │

└─────────▼─────────┘

│

Contextual token embeddings

│

──────────────────────────────────────────────

│

Previous target tokens (English so far)

│

[Token Embeddings + Positional Encoding]

│

┌─────────▼─────────┐

│ Decoder Layer │ (× N layers)

│ │

│ ┌─────────────┐ │

│ │ Masked Self │◄─┤ attends only to earlier outputs

│ │ Attention │ │

│ └─────────────┘ │

│ │ │

│ Add + Norm │

│ │ │

│ ┌─────────────┐ │

│ │ Encoder– │◄─┤ attends to encoder outputs

│ │ Decoder Attn│ │

│ └─────────────┘ │

│ │ │

│ Add + Norm │

│ │ │

│ ┌─────────────┐ │

│ │ FeedForward │ │

│ └─────────────┘ │

│ │ │

│ Add + Norm │

└─────────▼─────────┘

│

Decoder outputs per token

│

Linear projection + Softmax

│

Next word probability dist.

**2) 🧑‍💻 PyTorch / Hugging Face code snippet**

Here’s a tiny working example that uses BertTokenizer + BertModel to show **tokenization, embeddings, and attention matrices** for "This is a sample sentence for tokenization and encoding".

import torch

from transformers import BertTokenizer, BertModel

# Load pre-trained BERT (uncased)

tokenizer = BertTokenizer.from\_pretrained("bert-base-uncased")

model = BertModel.from\_pretrained("bert-base-uncased", output\_attentions=True)

# Example sentence

sentence = "This is a sample sentence for tokenization and encoding"

# Tokenize and encode

inputs = tokenizer(sentence, return\_tensors="pt")

# Run through model

with torch.no\_grad():

outputs = model(\*\*inputs)

# 1. Last hidden states (matrix: sequence\_length × hidden\_size)

last\_hidden\_states = outputs.last\_hidden\_state

print("Hidden states shape:", last\_hidden\_states.shape)

# 2. Attention weights from first layer, first head

attentions = outputs.attentions # tuple of (layers)

print("Number of layers:", len(attentions))

print("Shape of attention in first layer:", attentions[0].shape)

# (batch\_size, num\_heads, seq\_len, seq\_len)

# Get first head attention matrix

attn\_matrix = attentions[0][0, 0] # (seq\_len, seq\_len)

print("Attention matrix (layer 0, head 0):")

print(attn\_matrix)

# 3. Map token IDs back to tokens

tokens = tokenizer.convert\_ids\_to\_tokens(inputs["input\_ids"][0])

print("Tokens:", tokens)

**🔎 What you’ll see:**

* **Hidden states shape** → (1, seq\_len, 768) for BERT-base (768 hidden size).
* **Attention matrix shape** → (seq\_len, seq\_len) for one head (how each token attends to each other token).
* **Tokens** → WordPiece tokens like ['this', 'is', 'a', 'sample', 'sentence', 'for', 'token', '##ization', 'and', 'encoding'].

**Option 1: Google Colab (easiest, no install needed)**

1. Go to Google Colab.
2. Create a new notebook.
3. Copy–paste the code I gave you into a cell.
4. At the very top of your notebook, add this cell to install dependencies:
5. !pip install transformers torch
6. Run the cells — it will download the BERT model and show you the shapes + attention matrices. ✅

**⚡ Option 2: Local (your laptop / desktop)**

1. Make sure you have **Python 3.8+** installed.
2. Install the libraries in your terminal:
3. pip install transformers torch
4. Save the code in a file, e.g. bert\_example.py.
5. Run it with:

python bert\_example.py

/usr/local/lib/python3.12/dist-packages/huggingface\_hub/utils/\_auth.py:94: UserWarning: 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 (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session. You will be able to reuse this secret in all of your notebooks. Please note that authentication is recommended but still optional to access public models or datasets. warnings.warn( tokenizer\_config.json: 100%  48.0/48.0 [00:00<00:00, 3.03kB/s] vocab.txt: 100%  232k/232k [00:00<00:00, 2.71MB/s] tokenizer.json: 100%  466k/466k [00:00<00:00, 2.81MB/s] config.json: 100%  570/570 [00:00<00:00, 50.7kB/s] model.safetensors: 100%  440M/440M [00:16<00:00, 25.2MB/s] Hidden states shape: torch.Size([1, 12, 768]) Number of layers: 12 Shape of attention in first layer: torch.Size([1, 12, 12, 12]) Attention matrix (layer 0, head 0): tensor([[0.0533, 0.0942, 0.0381, 0.1342, 0.0334, 0.0505, 0.0788, 0.0370, 0.0419, 0.1040, 0.0486, 0.2859], [0.1777, 0.0362, 0.1360, 0.0273, 0.0755, 0.1090, 0.0597, 0.1076, 0.0740, 0.0614, 0.0658, 0.0697], [0.1185, 0.0485, 0.1061, 0.0415, 0.0622, 0.1139, 0.0606, 0.1768, 0.1228, 0.0496, 0.0546, 0.0449], [0.0840, 0.0918, 0.0956, 0.0668, 0.0701, 0.1243, 0.0685, 0.1182, 0.0710, 0.0875, 0.0659, 0.0561], [0.0490, 0.0431, 0.0724, 0.0140, 0.0761, 0.1614, 0.0086, 0.1235, 0.1611, 0.0337, 0.1858, 0.0712], [0.0584, 0.0406, 0.1110, 0.0229, 0.1427, 0.0820, 0.1046, 0.1164, 0.1172, 0.0656, 0.0863, 0.0522], [0.0933, 0.0750, 0.1194, 0.0662, 0.0917, 0.0811, 0.0855, 0.0752, 0.0634, 0.0642, 0.0415, 0.1435], [0.0688, 0.1140, 0.0753, 0.1021, 0.0729, 0.1207, 0.0226, 0.0592, 0.1455, 0.0360, 0.1338, 0.0490], [0.0538, 0.0542, 0.0806, 0.0297, 0.1343, 0.2037, 0.0220, 0.1471, 0.0805, 0.0164, 0.1230, 0.0546], [0.0823, 0.0949, 0.0937, 0.0774, 0.0581, 0.1013, 0.1051, 0.0776, 0.0645, 0.1006, 0.0440, 0.1006], [0.1270, 0.0363, 0.0820, 0.0174, 0.0816, 0.1566, 0.0333, 0.1430, 0.1378, 0.0327, 0.0940, 0.0581], [0.0875, 0.1202, 0.0678, 0.1378, 0.0445, 0.0842, 0.0549, 0.0589, 0.0797, 0.0972, 0.0443, 0.1230]]) Tokens: ['[CLS]', 'this', 'is', 'a', 'sample', 'sentence', 'for', 'token', '##ization', 'and', 'encoding', '[SEP]']

[**https://colab.research.google.com/drive/1NrI6KFAS6F9pUO14KnN\_vDeEm\_a56LnO#scrollTo=Y9ayrhTWKpWN**](https://colab.research.google.com/drive/1NrI6KFAS6F9pUO14KnN_vDeEm_a56LnO#scrollTo=Y9ayrhTWKpWN)

**Explanation — like I’m telling a 5-year-old 🧸**

Think of the whole thing as a small classroom of 12 kids (tokens). Each kid listens to other kids a little bit and then writes a note about what they now know. The numbers you see are just *how much each kid listened to every other kid*. Bigger number = listened more.

Below I’ll walk through each printed line and tell you what it means in plain words, then I’ll give you tiny code you can run in Colab to check and *see* things yourself.

**Line-by-line, simple meaning**

**/usr/local/... UserWarning: The secret HF\_TOKEN does not exist...**

* This is just a friendly warning from Hugging Face.
* **Meaning:** you didn’t provide a private key (HF\_TOKEN). That’s only needed if you want to access private models or upload things.
* **What to do:** you can ignore this for public models. If you want to log in, use huggingface-cli login or set a token in Colab secrets.

**tokenizer\_config.json: 100% / vocab.txt: 100% / tokenizer.json: 100% / config.json: 100% / model.safetensors: 440M/440M**

* These are **files being downloaded** from Hugging Face.
* **What they are:**
  + vocab.txt = the dictionary of tokens (words/subwords) the model knows.
  + tokenizer.json / tokenizer\_config.json = instructions on how to split text into tokens.
  + config.json = the model’s settings (how many layers, hidden size, etc.).
  + model.safetensors = the big file with the model’s learned knowledge (weights) — that’s why it’s hundreds of MB.
* The progress bars just show download progress.

**Hidden states shape: torch.Size([1, 12, 768])**

* **Translate:** we have 1 sentence (batch size = 1). It has 12 tokens. Each token is represented by a vector (a list) of 768 numbers.
* **Why 12 tokens?** Because your sentence got special tokens [CLS] at the start and [SEP] at the end, so the token list has length 12.
* **Analogy:** 12 kids, each with a notepad of 768 numbers.

**Number of layers: 12**

* This model (BERT-base) stacks 12 encoder layers — think of 12 rounds of discussion and note-writing. Each layer refines the notes.

**Shape of attention in first layer: torch.Size([1, 12, 12, 12])**

* **Breakdown of dims:** (batch\_size, num\_heads, seq\_len, seq\_len) → (1, 12, 12, 12).
  + Batch size = 1 (one sentence).
  + num\_heads = 12 (the model uses 12 different “attention viewpoints” or lenses).
  + seq\_len × seq\_len = 12 × 12 (each token can attend to every token).
* **Analogy:** 12 different pairs of glasses (heads). Each pair of glasses has a 12×12 table saying how much kid i listens to kid j.

**Attention matrix (layer 0, head 0): then a 12×12 table of numbers**

* This is **one** of those attention tables (layer 0, head 0).
* **Rows** = the token doing the listening (query).
* **Columns** = the token being listened to (key).
* **Each number** is between 0 and 1 and means “how much the row token focuses on the column token.”
* **Important:** each row sums to (approximately) 1 — they’re like slices of a pie distributed across the 12 tokens. The model decides how to slice the pie (softmax).

**Example reading:** look at the first row (for [CLS]): the largest number is 0.2859 in the last column — it means [CLS] is paying a lot of attention to the token in position 11 (that is [SEP]) under this particular head & layer. Other rows show different listening patterns.

**Why different heads?** Each attention head learns a different pattern. One head might focus on links between verbs and objects, another might strongly attend to punctuation, another to the start token — they split the job.

**Tokens: ['[CLS]', 'this', 'is', ... 'encoding', '[SEP]']**

* This prints the tokens corresponding to the 12 positions (index 0..11).
* Notice token and ##ization — WordPiece split a long word into two sub-tokens. That’s normal.
* You can use these token names to read the attention matrix: row 2 is token 'is', column 7 is token 'token', etc.

**What this all does behind the scenes (short)**

1. Tokenize text → list of tokens.
2. Convert tokens → embeddings (vectors).
3. For each encoder layer:
   * Compute attention matrices (one per head) telling *who listens to whom*.
   * Use those to build new token vectors (each token becomes a weighted blend of other tokens’ values).
   * Pass through a small per-token MLP (feed-forward).
4. After all layers, you get final contextual vectors (the Hidden states).

**Tiny checks you can run in Colab (copy–paste)**

These will help you *see* properties we talked about: that rows sum to 1, which token each row listens to most, and a heatmap.

# after you have `outputs` and `tokenizer` and `inputs` from the original run

import torch, numpy as np, matplotlib.pyplot as plt

# get layer 0, head 0 attention (same as printed)

attn = outputs.attentions[0][0, 0].cpu().numpy() # shape (seq\_len, seq\_len)

tokens = tokenizer.convert\_ids\_to\_tokens(inputs["input\_ids"][0])

# 1) confirm each row sums to ~1

print("Row sums:", np.round(attn.sum(axis=1), 6))

# 2) which token each row attends to the most?

most\_attended = attn.argmax(axis=1)

print("Most attended token per row:")

for i, j in enumerate(most\_attended):

print(f" row {i} ({tokens[i]}) -> col {j} ({tokens[j]}) weight={attn[i,j]:.4f}")

# 3) simple heatmap (layer 0, head 0)

plt.figure(figsize=(6,5))

plt.imshow(attn, aspect='auto')

plt.colorbar(label='attention weight')

plt.xticks(range(len(tokens)), tokens, rotation=90)

plt.yticks(range(len(tokens)), tokens)

plt.title("Layer 0 Head 0 Attention")

plt.show()

# 4) to see an averaged attention across all heads in layer 0:

avg\_heads = outputs.attentions[0][0].mean(0).cpu().numpy() # shape (seq\_len, seq\_len)

plt.figure(figsize=(6,5))

plt.imshow(avg\_heads, aspect='auto')

plt.colorbar(label='avg attention weight')

plt.xticks(range(len(tokens)), tokens, rotation=90)

plt.yticks(range(len(tokens)), tokens)

plt.title("Layer 0 Attention averaged over heads")

plt.show()

**A little extra intuition (so it *clicks*)**

* **Attention matrix = attention map** = who listens to who.
* **Multiple heads = multiple viewpoints.** Not every head will make immediate sense; some track grammar, some track punctuation, some track long-range dependencies.
* **Hidden states = final notes** each token writes after listening. If you look at the [CLS] hidden vector, that often summarizes sentence-level info (used for classification).

If you want, I can:

* Explain the output of the little checks above after you run them, OR
* Run a small simplified example here (with tiny 2-D embeddings) and show the math step-by-step (I can do that right now in the chat), OR
* Help you pick one head & layer and interpret *why* it attends as it does (we can inspect a few rows and reason about them).

Explain the output of the little checks above after you run them Also, Run a small simplified example here (with tiny 2-D embeddings) and show the math step-by-step (I can do that right now in the chat), OR

**ChatGPT said:**

Thought for 29s

Thought for 52s

Nice — I ran the checks on **your** attention matrix (layer 0, head 0) and also the tiny 2-D toy example. Below I’ll explain **every result** in plain, simple language (5-year-old style), then give a few quick next steps you can ask for.

**✅ What I ran**

1. **Checked row sums** (do the attention rows add to 1?)
2. **Found the single most-attended token per row** and the **top 3** for each row.
3. **Displayed a heatmap** of the attention matrix so you can “see” which entries are big/small.
4. **Toy example** with 3 tiny 2-D embeddings that shows the attention math step-by-step.

**Part A — explanation of the checks (your attention matrix, layer 0 head 0)**

**1) Row sums**

Output (rounded):

row 0 ([CLS]) sum = 0.999900

row 1 (this) sum = 0.999900

...

row 11 ([SEP]) sum = 1.000000

**Plain talk:** Each row is like a pie that a token slices up across all tokens. The slices add up to ~1. Tiny differences (0.9999 or 1.0001) are just normal rounding / floating-point precision — nothing to worry about. So yes — rows are valid probability distributions.

**2) Who each token listens to most (most-attended token per row)**

I extracted the *most attended* token and the *top 3* tokens for each row. Here’s the short, clear table:

* row 0 — [CLS] → **[SEP]** (weight 0.2859). top3: [SEP], a, and
* row 1 — this → **[CLS]** (0.1777). top3: [CLS], is, sentence
* row 2 — is → **token** (0.1768). top3: token, ##ization, [CLS]
* row 3 — a → **sentence** (0.1243). top3: sentence, token, is
* row 4 — sample → **encoding** (0.1858). top3: encoding, sentence, ##ization
* row 5 — sentence → **sample** (0.1427). top3: sample, ##ization, token
* row 6 — for → **[SEP]** (0.1435). top3: [SEP], is, [CLS]
* row 7 — token → **##ization** (0.1455). top3: ##ization, encoding, sentence
* row 8 — ##ization→ **sentence** (0.2037). top3: sentence, token, sample
* row 9 — and → **for** (0.1051). top3: for, sentence, [SEP]
* row10 — encoding → **sentence** (0.1566). top3: sentence, token, ##ization
* row11 — [SEP] → **a** (0.1378). top3: a, [SEP], this

**Plain talk / intuition:**

* [CLS] (the sentence-summary token) is *paying strong attention* to [SEP] in this head — some heads use [SEP] / sentence end as a cue for sentence-level info.
* token ↔ ##ization attending each other is expected: they are two pieces of the same word (“tokenization”) so they "talk" a lot to each other.
* sentence, sample, encoding appear often as high-attention targets in this head — this head seems to focus on those semantic words.
* Different rows focus on different tokens: that’s normal — each row shows what *that token* considered most important in this head.

**3) Heatmap**

* The heatmap shows the same numbers visually.
* **Bright spots** = large attention weight.
* **Diagonal bright** (token→itself) means the token mostly keeps its own information. Off-diagonal bright cells show cross-token influence (token A strongly listening to token B).

**Plain talk:** imagine a table where each row is someone listening; bright squares show who they listen to the most. The image you saw is that table drawn as a colored picture.

**Short conclusion for this head**

* This particular head (layer 0, head 0) has patterns like: [CLS]↔[SEP], token/##ization linking, and several tokens paying attention to sentence/encoding.
* Other heads (the model has 12) will show different patterns (grammar, punctuation, position, coreference, etc.). To really understand the model you usually look at several heads and layers, not just one.

**Part B — toy example (exact math, step-by-step)**

I ran the little 2-D example with tokens ["Je","suis","étudiant"] and identity Q/K/V (so easy to follow). Here’s what happened and what the numbers mean.

**Inputs (embeddings)**

Je = [1.0, 0.0]

suis = [0.0, 1.0]

étudiant = [1.0, 1.0]

**1) Raw scores = Q × Kᵀ / sqrt(d)**

We compute dot products and scale:

[[0.707107, 0. , 0.707107],

[0. , 0.707107, 0.707107],

[0.707107, 0.707107, 1.414214]]

**Plain talk:** these numbers measure similarity. Example: first row says “Je” is similar to itself and to “étudiant” (both 0.7071) but not similar to “suis” (0).

**2) Softmax per row → attention weights**

Apply softmax to make each row sum to 1:

[[0.401112, 0.197776, 0.401112],

[0.197776, 0.401112, 0.401112],

[0.248255, 0.248255, 0.503490]]

**Plain talk:** now each row is a probability slice. For example, “Je” now listens ~40% to itself, ~20% to “suis”, and ~40% to “étudiant”.

**3) Context vectors = weights × V**

Multiply weights by values to get each token’s new vector:

Je -> [0.802224, 0.598888]

suis -> [0.598888, 0.802224]

étudiant -> [0.751745, 0.751745]

**Plain talk:** each token’s new vector is a *blend* of the original word vectors according to who it listened to. So "Je" becomes a mix of "Je" and "étudiant", etc.

**Final takeaways (simple)**

* **Rows sum to 1** → each token distributes 100% of its attention across tokens.
* **Top targets** tell you who influences each token most in that head.
* **Wordpiece pairs** (like token + ##ization) usually attend to each other — that’s good and expected.
* **Different heads** do different jobs. Don’t judge the whole model by one head; look at many heads/layers to get the big picture.