Layers + Logits + Decoding (Session 2)

Session 1 was about how the model understands what it has seen so far. Session 2 is about how it turns that understanding into a reply.

Layers

Self-Attention

Step 1

Select the **Self-Attention** tab. In the open the **Sentence** dropdown at the top and pick "*The capital of Germany is Berlin. The capital of France is*".

The row of clickable token chips below the dropdown re-renders into the 12 tokens of this sentence: `The`, `capital`, `of`, `Germany`, `is`, `Berlin`, `.`, `The`, `capital`, `of`, `France`, `is`. The Generation panel below also resets to 0/10 tokens generated. The Layer and Head dropdowns reset to 6 / 0 (the default for this sentence).

> Notice the model doesn't see this as one long string. It sees it as twelve tokens. Most of them are whole words with a leading space and the leading space is part of the token. The thing we're trying to do is figure out which token the model picks next, and we already have a strong hunch what it should be: `Paris`. The whole point of the walkthrough is to see *how* the model gets to `Paris`.

Step 2

Look at the **Token Positions Across Layers** panel. Make sure the slider all the way to the left so it reads **0**.

Each blue dot is one token from the sentence, plotted in 2D (the real vectors live in 768 dimensions, but this is just a 2D approximation so we can render a rough visualization). One dot is red and bigger than the others which is the last token. The slider says zero, which means we haven't done any transformer work yet. What you see on screen is what the model *thinks* each token is right now.

Step 3

Keep refining the prompt

Drag the **Layer** slider from 0 to 12, pausing at a couple of intermediate values (try 3, 6, 9, 12). At each tick, the data visualization redraws with the token positions taken from a *deeper* set of internal representations. Tokens drift around. Tokens that were on top of each other at layer 0 separate. The red `is` dot in particular tends to move noticeably, and by the final layer it's sitting somewhere quite different from where it started.

Each tick is one layer of the transformer doing its job. A layer's job is basically: "look at the other tokens you're allowed to look at, and update your own picture of yourself based on what you find." Then the next layer does it again, on the updated picture. The model has twelve of these layers stacked on top of each other.

> Two things to notice:

> First, the dots that started piled up like the two `capital` tokens pull apart. After a few layers, the model has stopped thinking of them as the same word. One of them now means "the capital of Germany," the other means "the capital of France." The word on the page is the same. The internal vector is not.
>
> Second, watch the red dot. The red dot is the *last* token in the sentence. By layer 12, it has wandered a long way from where it started. That's the model gradually building up an answer to the question "given everything I just read, what should come next?" and storing that answer right at the last position.

Step 4

Update every token, then use the last position

Move the slider to **12**.

We are now twelve layers in. Every token has its own updated vector now. The model has refined its picture of every single position in the sentence.

But here's the thing.

When the model is about to guess the next word, it doesn't use all twelve token positions. It only looks at *one* of them: the last position. Why? Because in a left-to-right language model, the last position is the one that stands for "what should come next after everything so far." Every other position is just there to feed information forward into this one.

Basically, all the other tokens in the sentence did the work, and this red dot is the answer.

To summarize so far:

- The prompt comes in
- Prompt gets tokenized into tokens (may or may not be a word)
- Every token gets a token vector
- Each token vectors get refined by twelve rounds of attention
- At the end of all the rounds of attention, the model basically says "okay, give me the vector at the very last position."

Step 5 Q/K/V

Still numbers, not words
Inside the model, this is still just a long list of learned numbers, not readable text. You can think of it as the model's internal working notes before it turns those notes into an actual next token.

Q, K, V are three vectors every token produces at each attention layer (Query, Key, and Value).

From last session we talked about each token's Q is compared against every other token's K to get attention scores, those scores are softmaxed into weights, and the weights are used to mix everyone's Vs into a new vector for that token.

Change the **Layer** dropdown from 6 to **11**. The Q/K/V vectors are only exported at layers 0, 6, and 11, so pick 11 for the "deepest" view. Then click the **Show Q / K / V Vectors** button. Three colored bar strips appear in a new box.

Now click the last token chip, `is`, to populate them. Each strip is a row of tiny colored cells running left to right. Each cell is one number from a 64-dimensional vector. Some cells are red (positive), some blue (negative), darker means bigger magnitude.

This is what the inside of the model actually looks like at the last position, deep in layer 11. There's no word `Paris` written in here. The model's best guess of 'what should come after the capital of France is' is encoded as a list of numbers like this one.

> The whole job Logits and Decoding is to take this vector and turn it back into an actual word you can read.

Logits (logistic unit) and Decoding

After all 12 layers finish, every token position has a final 768-dimensional vector which is the model's "main state" at that position. The Q/K/V from layer 11 were intermediate computations that helped shape that state.

To get the logits, the model takes the final main state vector at the last position and performs a matrix multiplication on it against an unembedding matrix (transposed embedding matrix) and produces 50,257 numbers, one score for every token in the vocabulary.

Each of those 50,257 numbers is a *score* attached to a specific token ID. Once decoding picks the winning entry, the model has a single token ID. To turn that token ID back into something you can read, the model looks it up in the same tokenizer vocabulary table from Session 1. That string is what gets appended to the reply. That last lookup step (token ID back to text) is sometimes called *detokenization*. It's the exact inverse of what the tokenizer did to your input in Session 1.

Generation

Step 1

Scroll down to the **Generating the next token** panel. We're still on the same sentence: `The capital of Germany is Berlin. The capital of France is`. The "Generated so far" box shows the prompt unchanged, the counter reads `0 / 10 tokens generated`, and a column of ten horizontal bars is already drawn below it.

> If you use one of the other sentences and click on the highlighted token, you may notice some weird results, especially at intermediate layers. The weird tokens are real entries in GPT-2's vocabulary, and they exist because the tokenizer was trained on slightly different text than the model itself was trained on (GPT-2's tokenizer was trained on a corpus of Reddit-linked web pages. They were Reddit usernames (SolidGoldMagikarp, petertodd, TheNitromeFan), JavaScript variable names from boilerplate scraped pages (soDeliveryDate), ad-banner text (SPONSORED), Counter-Strike strings, weird Unicode mojibake (ĸł), etc. Frequent enough to earn a token slot). So they got vocabulary slots, but their embeddings barely got updated. They show up in this panel because we're doing a similarity lookup at an intermediate layer, where the model's internal state doesn't cleanly correspond to any one word, and any random direction in the space can win the lookup.

Step 2

Each row is one candidate token from the model's 50,257-token vocabulary, with a percentage on the right. The highlighted top bar is the model's best guess.

> The model just took its vector at the last position and compared it against every word in its vocabulary, giving each one a raw score. Those raw scores are called `logits`. The percentages on the bars are what those scores look like after one more step called `softmax`, which is a probability calculation which we discussed in Part 1.

Step 3

Click **Step**. The token `Paris` gets appended to the "Generated so far" box in blue. The counter ticks to `1 / 10`. The bars redraw, and the candidates and the percentages have all changed. The model just ran the entire pipeline again, on the slightly longer prompt `The capital of Germany is Berlin. The capital of France is Paris`, and produced new logits.

Step 4

Click **Step** a few more times. Each click does the same thing: pick the top, append, redo. It goes all the way back to the tokenization step we discussed in Part 1. The counter climbs, the "Generated so far" box grows, the bars keep redrawing. **This is the entire autoregressive generation loop.** ChatGPT, Claude, Gemini, and every mainstream chatbot you've heard of are autoregressive decoder-only (architecture) transformers.

Step 5 Sampling and Temperature

Click **Sample**. Each click is a fresh roll from the same step-0 distribution. Most of the time you'll get `Paris`, because `Paris` has by far the highest probability. But every so often you'll get `London`, `Marseille`, or one of the other top candidates instead. Over enough clicks the frequencies will roughly match the percentages on the bars.

> This is what real chatbots actually do. That's why ChatGPT gives you a slightly different answer every time you ask the same question, even when nothing else has changed.
>
> The dial that controls how "wide" the sampling is, meaning how often the model picks the second or third tier instead of the top, is called the *temperature*. (You can think of it as a creativity dial.) Turn it all the way down to zero and the model becomes deterministic (greedy decoding). Turn it up and the model takes more risks, which is good for creative writing. The loop itself is the same. Score every word, pick one, glue it on, repeat. We're just changing how picking works.

Step 6 Stop Conditions

How do models decide to stop talking?

Models stop on a special end-of-text token which is a special "I'm done talking now" symbol. As soon as that token scores highest, the loop ends. They can also stop on a max length, so a runaway model can't go forever.

Conclusion

That's the whole pipeline. Prompt, tokenization, tokens in, twelve layers to refine the token vectors, the last position becomes a vector, the vector becomes logits, a next token is selected, that token gets appended, and we loop. Fifty thousand candidate words, one click, repeat a few hundred times and you eventually have a sentence.