CPT and SFT: What Are We Teaching the Model?

Fengnan Deng · September 2026

Suppose we want to build a legal assistant. We have a collection of legal documents and a smaller set of carefully written questions and answers. Both are useful, but they teach different things. The documents expose the model to the language and content of the field. The answers show it how to respond to a user.

That is the main distinction between **continued pretraining (CPT)** and **supervised fine-tuning (SFT)**. CPT continues learning from text; SFT learns from demonstrations. Underneath, both usually train the model to predict the next token—a word or a piece of a word.

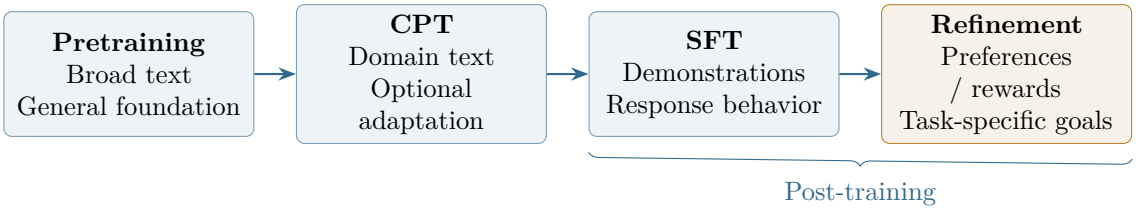

Figure 1: A common route from a general language model to a useful assistant. CPT and later refinement are optional; recipes vary.

CPT: give the model more to read

Pretraining builds a model's foundation from a broad collection of text. CPT starts from that trained model and keeps going, often with a more focused corpus [1]. For our legal assistant, this might include judgments, statutes, and commentary.

The training task is simple: given the text so far, predict what comes next. For a document with tokens $x_1, \ldots, x_T$, the average loss is

$$\mathcal{L}_{\text{CPT}} = -\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}).$$

Here $x_{<t}$ means all tokens before position t, and p_θ is the model's predicted probability. The loss gets smaller when the model assigns more probability to the actual next token. The text itself supplies the targets, so we do not need to write answers for every passage.

CPT can make specialist language and patterns more familiar to the model. It does not automatically make the model a good assistant, though: reading a judgment and answering a user's question are different tasks. We should check both domain performance and whether general abilities have been retained.

A useful variation: fill in the middle

Sometimes we want a model to complete a missing passage rather than continue the end of a document. **Fill-in-the-middle (FIM)** makes this possible by rearranging the training text [2]:

$$[\text{prefix} \mid \text{middle} \mid \text{suffix}] \quad \longrightarrow \quad [\text{prefix} \mid \text{suffix} \mid \text{middle}].$$

Special markers identify the pieces. The model reads both sides of the gap before generating the missing middle. Think of completing a function when the code above *and* below it is already given. Training still uses next-token cross-entropy; the order of the text has changed.

MTP: give each position more than one prediction task

Another variation is **multi-token prediction (MTP)**. The idea is to get more training signal from each position by adding predictions farther ahead. DeepSeek-V3 does this with a chain of small modules attached to the main model [4].

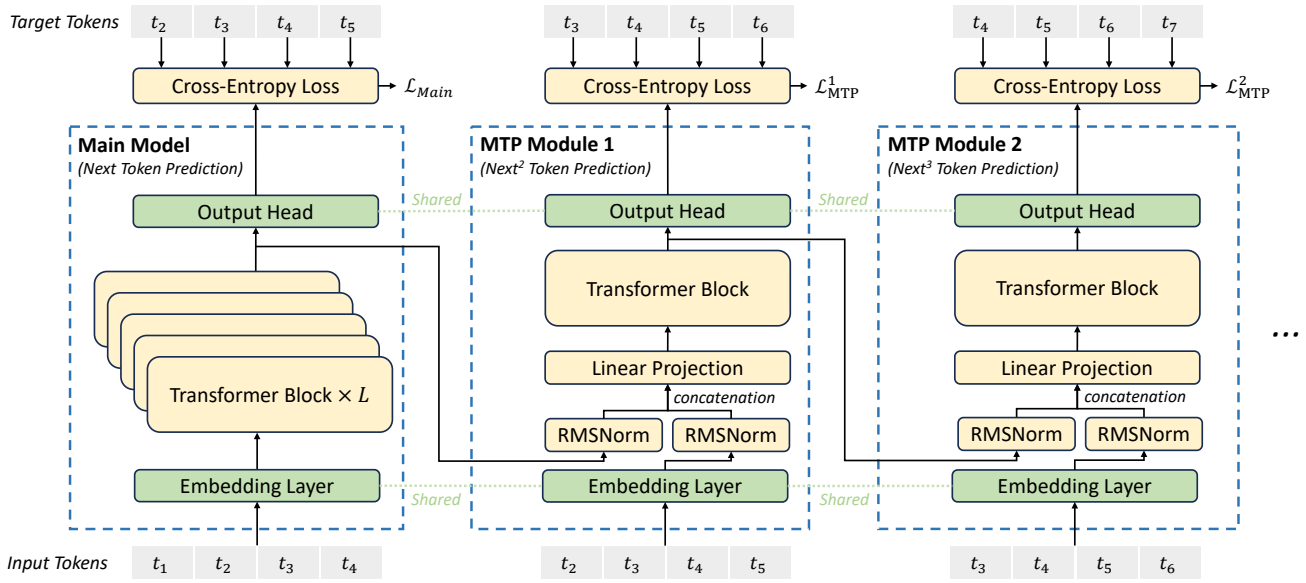

Figure 2: The sequential MTP design from Figure 3 of the DeepSeek-V3 Technical Report [4], reproduced from DeepSeek-AI (2024). Read the three blocks from left to right. The green embedding and output layers share parameters.

Let's follow one position in the toy sequence `The cat sat down`. For simplicity, treat each word as one token.

1. **The main model** reads `The cat` and predicts `sat`. It also produces a hidden representation: a vector summarizing the prefix.
2. **The first MTP module** takes that vector together with the actual next word, `sat`, and predicts `down`.
3. **The second module** takes the first module's vector together with `down` and predicts the following token, if there is one.

The important detail is where those extra words come from. During training, we take them from the document. A module sees the word *before* its target, never the target itself. This is the same teacher-forcing idea used in ordinary language-model training.

Inside each module, the two vectors are normalized, joined, and projected back to the model's hidden size. A causal Transformer block processes the result, and the shared output head predicts the token. Each module has its own projection and Transformer block. Unlike several independent heads guessing from the same prefix, these modules pass information along a chain.

Each prediction gets a cross-entropy loss. With D extra modules, the training objective is

$$\mathcal{L}_{\text{train}} = \mathcal{L}_{\text{next token}} + \frac{\lambda}{D} \sum_{k=1}^{D} \mathcal{L}_{\text{MTP},k}.$$

$\mathcal{L}_{\text{MTP},k}$ is the average negative log-probability of the correct target at depth k; padding and invalid targets are ignored. The weight λ controls how much the extra tasks matter. Their gradients also flow back into the main model, encouraging it to learn representations useful for later predictions.

The extra modules need not stay at inference time. They can be removed, or used to propose tokens for speculative decoding, where the main model checks the proposals.

SFT: show the model how to answer

For our legal assistant, SFT means showing it questions paired with good answers. These demonstrations teach it to explain clearly, follow instructions, and use the right format [3].

For a prompt q and an answer $a_1, \ldots, a_A$, the usual response-only loss is

$$\mathcal{L}_{\text{SFT}} = -\frac{1}{A}\sum_{j=1}^{A} \log p_\theta(a_j \mid q, a_{<j}).$$

The model sees the prompt, but this loss scores only the answer. Earlier answer tokens come from the demonstration during training and from the model during generation. Chat markers teach it where responses start and stop.

SFT can teach knowledge and skills as well as style, so the substance of the demonstrations matters too.

When the reasoning is much longer than the answer

A reasoning example may contain a prompt, a thinking trace, and a final answer. If we average the loss over every output token, a long trace gets most of the weight. With 900 reasoning tokens, 100 answer tokens, and 4 control tokens, the answer receives only $100/1004 \approx 10\%$ of the total weight.

We can instead average within each part first. Let $\bar{\ell}_C$, $\bar{\ell}_R$, and $\bar{\ell}_A$ be the mean token losses for generated control markers, reasoning, and the answer. Then use

$$\mathcal{L}_{\text{seg}} = \lambda_C \bar{\ell}_C + \lambda_R \bar{\ell}_R + \lambda_A \bar{\ell}_A, \qquad \lambda_C + \lambda_R + \lambda_A = 1, \quad \lambda_s \geq 0.$$

For example, weights $(0.1, 0.4, 0.5)$ give the answer half the total weight, even when the reasoning is long. These numbers illustrate the idea; they are not a default recipe.

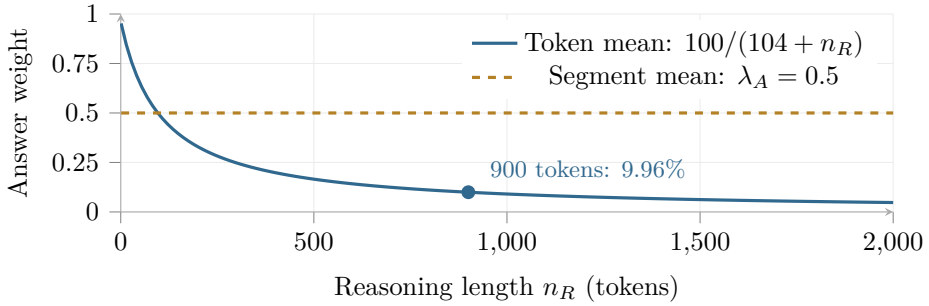

Figure 3: Calculated loss weights for 4 control tokens and 100 answer tokens. Segment weighting keeps the answer's share fixed as reasoning grows.

Average within each segment, then across examples; omit absent segments and renormalize the remaining weights. The weights control training emphasis, not exact gradient contributions or generated reasoning length. Tune them against held-out answer quality and format.

References

[1] S. Gururangan et al. Don't Stop Pretraining: Adapt Language Models to Domains and Tasks. ACL, 2020.

[2] M. Bavarian et al. Efficient Training of Language Models to Fill in the Middle. 2022.

[3] L. Ouyang et al. Training Language Models to Follow Instructions with Human Feedback. 2022.

[4] DeepSeek-AI et al. DeepSeek-V3 Technical Report. 2024.