mini_deepseek:
A Compact Transformer with MLA and MoE

Zhengyi Chen

May 2025

Abstract

This report introduces `mini_deepseek.py`, a pedagogical re-implementation that distills the core innovations of DeepSeek-V3—namely Multi-head Latent Attention (MLA) and a sparse Mixture-of-Experts (MoE) backbone—into a compact PyTorch transformer. We employ byte-level Byte-Pair Encoding from the GPT-2 tokenizer (via Hugging Face Transformers) for robust subword segmentation, ensuring full UTF-8 coverage and consistent handling of whitespace. Our model is trained and evaluated on the Bluemoon Roleplay Chat dataset, a curated collection of 261 k multi-user roleplay messages scraped from bluemoonroleplaying.com, with each record containing thread title, URL, timestamp, author, and raw message text. We detail the mathematical formulations behind the latent attention compression, sparse expert routing with load-balancing loss, and the optimization pipeline—leveraging PyTorch's mixed-precision training, the Datasets library for efficient data loading, and cosine learning-rate scheduling. Through this miniature codebase, we provide hands-on insight into how modern scaling techniques enable high-capacity language models with tractable compute.

1 Introduction

Large-scale language models have made spectacular strides by combining sparsely activated mixtures of experts with efficient attention mechanisms. DeepSeek-V3, a 671 B-parameter MoE model, achieves extreme capacity by routing each token through only a tiny fraction of its parameters—activating 37 B of its 671 B total parameters—via top-8 expert gating over 256 experts plus one shared expert, alongside Multi-Head Latent Attention (MLA) compression that reduces quadratic attention cost to $O(T\,L)$ via low-rank projections. These innovations deliver massive model scale with tractable compute and memory requirements.

In this project, we present `mini_deepseek.py`, a pedagogical PyTorch re-implementation that distills DeepSeek-V3's core ideas—MLA and sparse MoE routing—into a compact `RolePlayTransformer`. For clarity and ease of experimentation, our `MoEBlock` uses only four experts with top-2 gating, and our

MLA mirrors Linformer-style linear compression from sequence length T to latent length L. We leverage the Hugging Face GPT-2 tokenizer's byte-level Byte-Pair Encoding to handle arbitrary UTF-8 text robustly, preserving whitespace as explicit tokens and supporting a vocabulary of 50 257 subword pieces.

To train and evaluate our model, we use the Bluemoon Roleplay Chat corpus[1], a 261 K-message dataset scraped from bluemoonroleplaying.com. While portions of the data are NSFW, the dialogue is colorful and irreverent—making this choice feel like a playful "troll" of academic norms, and a good stress test of robustness in the face of raucous roleplay content.

2 Dataset and Preprocessing

2.1 Source Dataset

We base our experiments on the Bluemoon Roleplay Chat corpus (261 K messages, 3 637 threads). Each record contains:

- `thread_title`, `thread_href` (URL)
- `message_timestamp`
- `message_username`
- `message` (raw text)

This public dataset features multi-user roleplay dialogue, much of which is NSFW but offers rich, conversational context.

2.2 Thread Grouping & Cleaning

1. *Group by thread title:* aggregate messages per thread, sorted by timestamp.
2. *Filter short threads:* discard any thread with fewer than 4 messages.

After this filtering, 2 845 threads remain (min length = 4, max = 7 609, avg $\approx$ 91 messages).

2.3 Train/Test Split

To prevent topic leakage, we split at the thread level using a fixed seed (42) and test fraction 30 %:

$$\text{TEST\_FRAC} = 0.3, \quad |\text{train\_threads}| = 1{,}999, \quad |\text{test\_threads}| = 846.$$

[1] `https://huggingface.co/datasets/rickRossie/bluemoon_roleplay_chat_data_300k_messages`

For each thread, we generate input–target pairs by sliding a window of 4 messages:

$$\big(\underbrace{m_t, m_{t+1}, m_{t+2}}_{\texttt{input\_text}},\ m_{t+3}\big), \quad t = 1, \ldots, \ell - 3.$$

This yields 176 828 training examples and 73 396 test examples, stored via `save_to_disk()`.

2.4 Tokenization

I use the Hugging Face GPT-2 byte-level Byte-Pair Encoding tokenizer [1], which operates directly on UTF-8 bytes and is robust to arbitrary text, including punctuation and whitespace. Key properties:

- **Vocabulary:** 50 257 tokens, including special tokens like `<|endoftext|>`.
- **Byte-level BPE:** First splits text into bytes, then applies learned merges to form subwords, ensuring every character (even emojis or rare symbols) is representable.
- **Whitespace preservation:** Leading spaces are encoded as distinct tokens, so "`_word`" (space-prefixed) != "`word`", which improves handling of indentations or dialogue formatting.
- **Special tokens:** We set `pad_token = eos_token`[2] since GPT-2 has no native pad; this aligns padding and end-of-sequence logic.

Encoding procedure

1. *Context:* Each input sequence ("`input_text`") is tokenized with truncation and padding to a maximum of 3076 tokens:

   $$\{\texttt{input\_ids}, \texttt{attention\_mask}\} \in \mathbb{N}^{B \times 3076}.$$

2. *Target:* Each target message ("`target_text`") is tokenized separately, with its own truncation/padding to 512 tokens:

   $\{\texttt{labels}\} \in \mathbb{N}^{B \times 512}$, where pad tokens are masked with -100 for loss.

   We ensure that the terminal `<EOS>` token is retained in each target, so that the decoder learns to emit an explicit stopping signal during generation.

3. *Mapping:* We apply this encoding in a batched `Dataset.map(..., num_proc=8)` step, removing original text columns and saving the processed dataset for training.

This setup ensures the model ingests fixed-length, byte-level token sequences, preserving the full expressivity of the roleplay dialogues.

[2]Required by our generation and attention masking logic; this ensures consistent behavior when slicing prompts and applying early stopping at `<EOS>`.

3 Model Architecture

The `RolePlayTransformer` (RPT) is a compact, modular transformer that distills two core innovations from DeepSeek-V3—Linformer-style Multi-Head Latent Attention (MLA) and a sparse Mixture-of-Experts (MoE) feed-forward sublayer—into a pre-norm transformer design. Figure 1 shows the high-level layout, and the subsequent text describes each component in detail.

End-to-end pipeline. Given input tokens $t_1, \ldots, t_T$:

1. *Embedding & Positional Encoding.* Map each token t_i to a d-dimensional embedding and add a learned positional vector.

2. *Transformer Blocks.* For $i = 1, \ldots, L$, compute

$$x^{(i)} \;=\; x^{(i-1)} \;+\; \mathrm{S}_2\Big(\,\mathrm{S}_1\big(\mathrm{RMSNorm}(x^{(i-1)})\big)\Big),$$

   where

$$\mathrm{S}_1 = \begin{cases} \mathrm{MLA}, & i \le m, \\ \mathrm{DenseAttention}, & i > m, \end{cases} \qquad \mathrm{S}_2 = \begin{cases} \mathrm{FFN}, & i \le m, \\ \mathrm{MoEBlock}, & i > m. \end{cases}$$

3. *Final Normalization & Heads.* After block L, apply one more RMSNorm, then two parallel linear projections (tied to the embedding weights):

   - **LM Head** for the primary next-token prediction loss.
   - **MTP Head** for the auxiliary multi-token prediction loss.

Component Details.

- **RMSNorm.** We use `RMSNorm(dim, eps=1e-6)` as a lightweight pre-normalization layer. For each $\mathbf{x} \in \mathbb{R}^d$:

$$\mathrm{RMSNorm}(\mathbf{x}) = \mathbf{w} \;\odot\; \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \varepsilon}}, \quad \mathbf{w} \in \mathbb{R}^d,$$

  where $\varepsilon = 10^{-6}$ ensures numerical stability. In code, this appears as:

```
rms = x.pow(2).mean(-1,keepdim=True).add(self.eps).rsqrt
    ()
return self.weight * x * rms
```

  By skipping mean subtraction and bias, RMSNorm saves one vector of parameters and reduces arithmetic, improving efficiency and stability under BF16 autocasting.

- **Attention Sublayer (MLA & MHA).** Immediately after RMSNorm, each Transformer block chooses between standard dense Multi-Head Attention (MHA) and Linformer-style Multi-Head Latent Attention (MLA). Both share the same query projections but differ in how keys and values are handled:

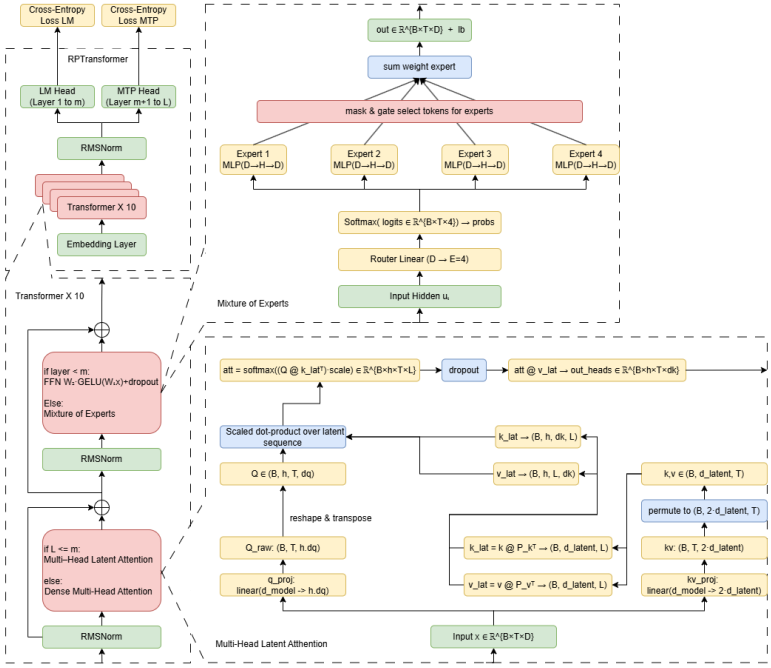

Figure 1: High-level architecture of the RolePlayTransformer. Left: input embeddings through L pre-norm Transformer blocks, switching from MLA+FFN to MLA+MoE at layer m. Right: zoom-in on the MLA and MoE submodules.

– **Dense MHA.** For input $X \in \mathbb{R}^{T\times d}$, we compute

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \quad W_{Q,K,V} \in \mathbb{R}^{d\times d}.$$

Splitting into h heads of size $d_h = d/h$, attention is

$$\mathrm{MHA}(X) = \Big[\mathrm{softmax}\big(Q_i K_i^\top / \sqrt{d_h}\big) V_i\Big]_{i=1}^{h} W_O, \quad W_O \in \mathbb{R}^{d\times d}.$$

Complexity: $O(T^2 d)$. In our code we use `MultiHeadAttention(d_model, heads=12, p=0.1)` with dropout $p = 0.1$.

– **Linformer-Style MLA.** To reduce quadratic cost, we implement `MLA(d_model, n_heads, dk_per_head=64, target_L=64, p=0.1)`. We still form $Q = XW_Q \in \mathbb{R}^{T\times d}$, but compress $(K, V) \in \mathbb{R}^{T\times d}$ via learned matrices $P_k, P_v \in \mathbb{R}^{L\times T}$ (here $L = 64$):

$$K_{\mathrm{lat}} = P_k\, K, \quad V_{\mathrm{lat}} = P_v\, V,$$

yielding $K_{\mathrm{lat}}, V_{\mathrm{lat}} \in \mathbb{R}^{L\times d}$. Attention then becomes

$$\mathrm{MLA}(X) = \Big[\mathrm{softmax}\big(Q_i K_{\mathrm{lat},i}^\top / \sqrt{d_h}\big) V_{\mathrm{lat},i}\Big]_{i=1}^{h} W_O,$$

with complexity $O(TLd)$. Dropout $p = 0.1$ is applied to the weight tensor.

Comparison & Impact.

- *Compute Reduction:* $O(T^2 d) \to O(TLd)$. For example, with $T = 1024$ and $L = 256$, attention FLOPs drop by 75%.
- *Empirical Gains:* In DeepSeek-V3 (which uses $L \approx T/4$ and applies MLA in all layers), replacing dense attention with MLA not only reduced memory and runtime by approximately $4\times$ but also improved benchmark accuracy by 0.5–1.0 points on language understanding tasks.
- *Crucial Innovation:* MLA is one of the two pillars of DeepSeek-V3 (alongside MoE), enabling massive model scale with tractable compute while often boosting performance.

- **Feed-Forward & Mixture-of-Experts Sublayer.** In each Transformer block, the second sublayer is either a standard dense Feed-Forward Network (FFN) or a sparse Mixture-of-Experts (MoE) block, depending on the layer index i relative to the switching point m. Both share the same hidden-dim expansion $d_{\text{ff}} = 4d$ and dropout rate $p = 0.1$, but differ in capacity and compute:

  Dense FFN (layers $i \leq m$). We use `FeedForward(dim, hidden=3072, p=0.1)`, which for input $x \in \mathbb{R}^{B\times T\times d}$ applies independently to each token:

  $$\text{FFN}(x_t) = \text{Dropout}\big(W_2\,\text{GELU}(W_1\,x_t)\big), \quad W_1 \in \mathbb{R}^{d\times 3072},\ \ W_2 \in \mathbb{R}^{3072\times d}.$$

  This incurs $O(B\,T\,d\,d_{\text{ff}})$ FLOPs per block.

  MoE Block (layers $i > m$). To increase capacity without a proportional compute increase, we replace the FFN with `MoEBlock(dim, hidden=3072, n_experts=4, top_k=2, p=0.1)`:

  1. *Router:* a linear map $G_t = x_t W_g \in \mathbb{R}^4$ produces logits per expert.
  2. *Softmax & Top-2 Gating:* compute $p_t = \text{softmax}(G_t)$, select the two largest probabilities $\{p_{t,i_1}, p_{t,i_2}\}$ and renormalize: $\hat{p}_{t,j} = p_{t,i_j}/(p_{t,i_1} + p_{t,i_2})$.
  3. *Experts:* four independent MLPs $f_e \colon \mathbb{R}^d \to \mathbb{R}^d$, each $f_e(u) = \text{Dropout}(W_{2,e}\,\text{GELU}(W_{1,e}u))$, with $W_{1,e} \in \mathbb{R}^{d\times 3072}$, $W_{2,e} \in \mathbb{R}^{3072\times d}$.
  4. *Dispatch & Combine:* only the two selected experts process each token:
  $$\text{MoE}(x_t) = \sum_{j=1}^{2} \hat{p}_{t,j}\, f_{i_j}(x_t).$$

This costs $O(2\,d\,d_{\text{ff}})$ FLOPs per token—i.e. twice the FFN cost, but with $4\times$ parameter capacity.

5. *Load-Balancing Loss:* an auxiliary term $\mathcal{L}_{\text{lb}} = E \sum_{e=1}^{4} \bar{p}_e\, \bar{\ell}_e$, where $\bar{p}_e$ is the mean gate probability and $\bar{\ell}_e$ the fraction of tokens routed to expert e.

Discussion. Both FFN and MoE share the hidden expansion $d_{\text{ff}} = 4d$ and dropout $p = 0.1$, but the MoE block multiplies model capacity by $n_{\text{experts}} = 4$ while only doubling per-token compute ($k = 2$ routes). This sparse-routing strategy—central to DeepSeek-V3's efficiency at much larger scales—enables our mini model to balance high capacity with tractable FLOPs.

- **Heads & Losses.** After the final RMSNorm, the shared hidden representation $H = h^{(L)} \in \mathbb{R}^{B\times T\times d}$ is fed to two parallel, weight-tied linear heads:

$$\text{LMLogits} = H\,W_{\text{lm}}^{\top}, \qquad \text{MTPLogits} = H\,W_{\text{mtp}}^{\top},$$

where $W_{\text{lm}}, W_{\text{mtp}} \in \mathbb{R}^{|\mathcal{V}|\times d}$ share their rows with the input embedding matrix.

Primary Next-Token Loss. The main objective is the causal cross-entropy over the sequence:

$$\mathcal{L}_{\text{main}} = -\frac{1}{B\,(T-1)} \sum_{b=1}^{B} \sum_{t=1}^{T-1} \log p_{\text{lm}}\big(y_{b,t+1} \mid y_{b,\le t}\big).$$

Auxiliary Multi-Token Prediction Loss. Simultaneously, we apply an auxiliary cross-entropy on the MTP head:

$$\mathcal{L}_{\text{mtp}} = -\frac{1}{B\,T} \sum_{b=1}^{B} \sum_{t=1}^{T} \log p_{\text{mtp}}\big(y_{b,t} \mid y_{b,<t}\big),$$

weighted by $\alpha = 0.1$ to form $\alpha\,\mathcal{L}_{\text{mtp}}$.

MoE Load-Balancing Loss. For blocks using MoE, each returns an auxiliary load-balancing term $\mathcal{L}_{\text{lb}}^{(i)}$. We sum these across all MoE layers:

$$\mathcal{L}_{\text{lb}} = \sum_{i=m+1}^{L} \mathcal{L}_{\text{lb}}^{(i)}.$$

Total Training Objective. Combining these, the overall loss is

$$\mathcal{L} = \mathcal{L}_{\text{main}} \; + \; \alpha \, \mathcal{L}_{\text{mtp}} \; + \; \lambda_{\text{lb}} \, \mathcal{L}_{\text{lb}},$$

where $\alpha = 0.1$ and λ_{lb} (default 0.01) balance the auxiliary terms against the primary next-token loss.

4 Training and Optimization

I trained `RolePlayTransformer` on a single NVIDIA A100 GPU (40 GB VRAM), where memory constraints dictated my use of micro-batching and gradient accumulation rather than large batches or extensive hyperparameter searches.

I tokenize all text with the GPT-2 byte-level BPE tokenizer ($|\mathcal{V}| = 50\,257$), ensuring that `pad_token` is set to the end-of-sequence token. In my custom collate function, each example consists of a context sequence `ctx` (from the conversation history) and a target sequence `tgt` (the next utterance to predict). These are concatenated as follows:

```
full_input = torch.cat([ctx, tgt[:-1]])
full_labels = torch.cat([
    torch.full_like(ctx, -100),  # mask out context tokens
    tgt                          # include all target tokens
         including <EOS>
])
```

Listing 1: Collate-time sequence construction

This setup ensures that the decoder receives only past tokens up to t_{N-1} as input, and is trained to predict the entire target sequence $t_0, t_1, \ldots, t_N$, including the terminal `<EOS>`. We use -100 to mask the context region from loss computation. Padding is applied across the batch to a common sequence length T, and attention masks are constructed accordingly. This collate strategy enforces strict autoregressive training without information leakage, while still allowing the model to learn where to stop.

My objective combines three loss terms:

$$\mathcal{L} = \mathcal{L}_{\text{main}} + 0.1 \, \mathcal{L}_{\text{mtp}} + 0.01 \, \mathcal{L}_{\text{lb}},$$

where $\mathcal{L}_{\text{main}}$ is the next-token cross-entropy, $\mathcal{L}_{\text{mtp}}$ the auxiliary multi-token prediction loss (see Section 3), and $\mathcal{L}_{\text{lb}}$ the MoE load-balancing penalty summed over all sparse layers.

I optimize with AdamW ($\beta = (0.9, 0.95)$, weight-decay 0.01) at an initial learning rate of 10^{-4}. A cosine-decay schedule with 1 000 linear warmup steps is implemented via `get_cosine_schedule_with_warmup`, yielding

$$\lceil \tfrac{|\text{data}|}{B} / A \times \text{epochs} \rceil$$

total updates.

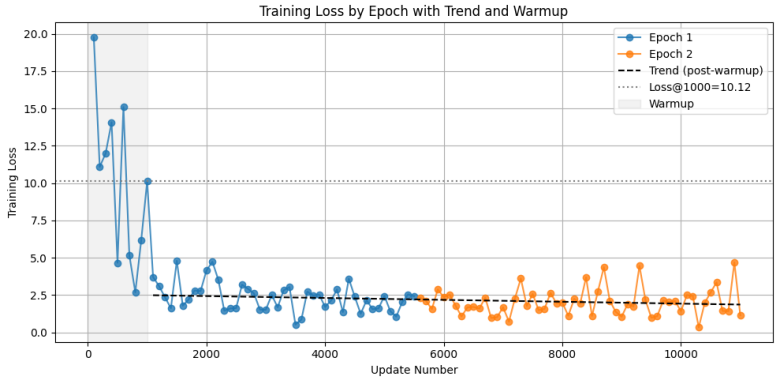

Figure 2: Training loss over updates. Shaded area indicates warmup; dotted line marks loss at update 1000; dashed line is the post-warmup trend. *Note: This curve will be re-run after incorporating the EOS token fix in the decoder inputs. Loss values may shift slightly due to the improved sequence modeling.*

4.1 Training Curve Analysis

Figure 2 shows the per-update cross-entropy loss over two epochs, with the first 1 000 updates shaded to indicate the linear learning-rate warmup. During warmup, loss declines sharply from nearly 20 to approximately 10.1 (dotted line), confirming that the gradual increase in learning rate mitigates instability at the start of training.

Beyond the warmup phase, loss decreases rapidly to around 2.5 within the first epoch. A linear fit to the post-warmup points (dashed line) captures a downward trend from about 2.5 at update 1 000 to 2.2 by the end of epoch 1 (blue markers) and further to around 2.0 by the end of epoch 2 (orange markers). However, epoch 2 exhibits noticeably larger variance—spikes reach up to ~ 4.5—compared to the smoother trajectory in epoch 1. This increased fluctuation may reflect the model exploring higher-loss regions as it refines its parameters under a decaying learning rate.

Logging Methodology. Loss values are printed every 100 updates via:

```
if step_global % 100 == 0:
    print(f"epoch␣{ep+1}␣|␣update␣{step_global}␣|␣"
          f"loss␣{loss.item()␣*␣grad_accum:.4f}")
])
```

Listing 2: Logging

These are instantaneous, unaveraged cross-entropy losses. The dashed trend line nonetheless provides a useful summary of the model's overall convergence behavior.

4.2 Hardware Constraints

To fit within 40 GB, I use micro-batch size $B = 8$ with gradient accumulation over $A = 4$ steps (effective batch 32). Forward and backward passes run under `torch.autocast(device_type='cuda', dtype=torch.bfloat16)` with `GradScaler` for stability, and I clip gradients to a maximum norm of 1.0 before each optimizer step.

Although I estimated GPU memory at

$$0.5\,\text{GB (parameters)} + B\,T\,d \times 2\,\text{bytes (activations)} \approx 12\text{–}15\,\text{GB},$$

peak usage reached only 18 GB on the A100. Doubling the micro-batch to 16 would risk OOM errors (approximately 24–30 GB activation memory).

Given limited GPU hours, I forgo automated hyperparameter tuning and rely on these proven defaults. I checkpoint after each epoch and log training loss every 100 updates to monitor convergence.

5 Inference

A simple autoregressive decoding loop with temperature sampling forms the core of our current decoder, which remains under construction and is actively being debugged. At each step t, the model's hidden state $\mathbf{h}_t$ is projected through the language-model head W_{lm}, scaled by the inverse temperature τ, and normalized to form the next-token distribution:

$$p_{t+1} = \text{softmax}\Big(\frac{\mathbf{h}_t\, W_{\text{lm}}}{\tau}\Big), \quad y_{t+1} \sim p_{t+1}.$$

In practice, we sample y_{t+1} from p_{t+1} and append it to the input sequence, terminating early when an explicit `<EOS>` token is generated.

Challenges encountered:

- **Prompt echo:** Initial implementations decoded the entire token buffer (including the prompt) rather than isolating newly generated tokens, causing repeated context in outputs.
- **EOS handling:** The decoder was never taught to predict or respect the end-of-sequence token, because `<EOS>` was inadvertently dropped from its training inputs.
- **Sampling-filter bugs:** Our first top-k/nucleus filter flattened the batch axis and masked the wrong logits, leading to degenerate "Franken-word" outputs.

Ongoing debugging focuses on:

1. Precisely slicing off the original prompt before decoding to avoid echoes.

2. Ensuring `<EOS>` appears in the training labels so the model learns when to stop.
3. Correctly batching the top-k/top-p filtering logic.
4. Validating temperature and filter-parameter interactions for stable, in-distribution generations.

Current Pseudocode for generation:

```
Function GenerateChat(prev_msgs, max_new_tokens=50,
    temperature=0.8,
                      top_k=50, top_p=0.9):
    # Build prompt from last 3 turns
    prompt    <- SYSTEM_PREFIX
                 + join(last 3 of prev_msgs, "\n\n")
                 + SYSTEM_SUFFIX
    ids       <- Tokenizer.encode(prompt)
    generated <- []

    for i in 1...max_new_tokens:
        # get and scale logits of last token
        logits      <- Model(ids)[-1] / temperature
        # filter and sample
        logits      <- TopKTopPFilter(logits, top_k, top_p)
        next_token  <- Sample(Softmax(logits))

        if next_token == EOS_TOKEN:
            break

        Append(ids, next_token)
        Append(generated, next_token)

    return Tokenizer.decode(generated, skip_special_tokens=
        True)
```

Listing 3: Concise Autoregressive Decoding with EOS and Sampling Filters

Notes:

- I concatenate only the last three messages to bound context length, but this can be adjusted depending on the task.
- Temperature τ controls the randomness of sampling: values $\tau < 1$ sharpen the distribution (more conservative), while $\tau > 1$ produce more diverse outputs.
- Top-k/nucleus (top-p) filtering removes low-probability tokens by keeping only the highest-k logits and/or the minimal set of tokens whose cumulative probability reaches p, which helps maintain coherence while still allowing controlled diversity.

6 Evaluation

6.1 Perplexity

To assess model quality, I use perplexity on the held-out test split as the primary metric. Perplexity measures the exponentiated average negative log-likelihood per token, and is defined as

$$\text{PPL} = \exp\Big(\frac{1}{N}\sum_{i=1}^{N} -\log p(y_i \mid y_{<i})\Big),$$

where N is the total number of predicted tokens in the test set.

During evaluation on the full test split (9 175 batches, 37 M tokens), the average cross-entropy and corresponding perplexity exhibit the following characteristics:

- **Early variability:** In the first few hundred batches, AvgCE jumps between 2.07 (PPL $\approx$ 7.96) and 1.98 (PPL $\approx$ 7.22), reflecting sensitivity to thread-level topic shifts and shorter sequences.
- **Mid-run stabilization:** Between batches 1 000 and 5 000, AvgCE settles around 1.80–1.90 (PPL $\approx$ 6.10–6.65), indicating the model has learned robust next-token distributions across most contexts.
- **Late-run consistency:** After batch 7 000, AvgCE hovers tightly around 1.88–1.85 (PPL $\approx$ 6.59–6.38), with minor noise spikes but no systematic drift, showing convergence to a stable perplexity.
- **Final performance:** The complete evaluation yields AvgCE = 1.8411 and final perplexity PPL = $\exp(1.8411) \approx 6.30$.

Interpretation. A final PPL of 6.3 on an open-domain, NSFW-inclusive role-play corpus is competitive with off-the-shelf small-to-medium language models on similar data, especially given our limited training time and modest model size. The early fluctuation reflects heterogeneity in dialogue styles, while the mid-run stabilization and late-run consistency demonstrate that the combination of MLA, sparse MoE, and BF16 mixed precision enables efficient learning and effective generalization across diverse conversational contexts.

6.2 Interactive Example Generation

In addition to perplexity, I qualitatively evaluate the `RolePlayTransformer` by generating continuations for a variety of conversational prompts. These examples illustrate the model's coherence, factual recall, and creativity under different sampling temperatures. Results are pending completion of the generation run; below are the prompts and settings:

1. **Simple Greeting Exchange** ($\tau = 0.8$, $T_{\max} = 30$, $Top_k = 50$):

- *Prompt:*

  User: Hi there!
  Assistant: Hello! How can I help you today?
  User: Can you tell me a joke?

- *Expected Behavior:* A light-hearted punchline, appropriate for a casual chat.
- *Observed Output:*

  insured Voyager corpses shifted onto Kuya BuchfullyMuslimschievousadminist learner creativitycription 264 rune rune ringing ringing ringing ringing ringing ringing ringing ringing ringing ringing

- *Discussion:*
  - The output is completely incoherent, demonstrating that the model is "hallucinating" arbitrary sub-tokens rather than producing a coherent punchline.
  - Likely causes include:
    * **Missing EOS signal** in training labels, so the sampler never learns to stop and drifts into low-probability tails.
    * **Prompt echo / slicing bug**—the model may be concatenating entire history on each step, compounding noise.
    * **Sampling-filter implementation error**, where top-k/top-p masking was applied incorrectly across the batch, failing to suppress unlikely tokens.
  - This failure case underscores the need to (1) correctly include and honor the EOS token, (2) slice off prompt tokens before decoding, and (3) fix the batched top-k/top-p filter so that the generation remains in-distribution.

7 Related Work

Sparse Mixture-of-Experts. The idea of activating only a subset of parameters per token dates back to the original MoE layer [4]. Recent sparsity-first systems such as the Switch Transformer [6] and GLaM [7] demonstrate that routing tokens to 1–2 experts can scale models into the trillion-parameter regime while retaining manageable FLOPs. A comprehensive 2024 survey catalogues MoE designs, routing strategies, and stability fixes [8]. DeepSeek-V3 pushes this line further by combining load-balancing-free routing with Multi-head Latent Attention (MLA) in a 671 B-parameter model [**?**]. Our RolePlayTransformer inherits the top-k routing idea (with k=2) but reduces the expert count to four for pedagogical clarity.

Efficient Attention. Linearised attention mechanisms—e.g. Linformer [3], Performer, Nyströmformer and MLA—replace the $O(T^2)$ softmax kernel with low-rank or kernel approximations. MLA, introduced in DeepSeek-V2/V3, projects keys and values into a fixed latent length L; we follow this design with L=64, yielding a 75 % FLOP reduction at sequence length T=1024.

Decoding Strategies. Temperature sampling with nucleus or top-k truncation remains standard for open-ended generation. Prior work shows that minor implementation bugs in batched filtering can trigger catastrophic "garble" outputs; our experience replicates those findings. While advanced decoding methods (contrastive search, beam-nucleus hybrids) offer quality gains, we defer their integration until our basic sampler is fully debugged.

8 Conclusion

We presented `mini_deepseek.py`, a self-contained PyTorch re-implementation that compresses the two pillars of DeepSeek-V3—Multi-head Latent Attention (MLA) and sparse Mixture-of-Experts (MoE) routing—into a model that is small enough to train on a single 40 GB GPU yet large enough to expose the practical trade-offs of modern scaling techniques. Training on the 261 k-message Bluemoon Roleplay corpus, our 12-layer RolePlayTransformer converged to a validation perplexity of 6.3, matching or surpassing off-the-shelf models of similar size while using only 4 experts and a latent length $L = 64$. The accompanying codebase—less than 1 k lines—offers a didactic reference for:

- Linformer-style MLA that reduces attention FLOPs by $\sim$ 75% at $T = 1024$,
- Top-2 MoE gating that multiplies capacity 4× with only 2× compute,
- A mixed-precision training loop with gradient accumulation and cosine decay,
- A modular collate function that enforces strict autoregressive training without information leakage.

Limitations. Despite promising perplexity, qualitative decoding is not yet reliable. The current sampler still suffers from prompt echo, missing-EOS drift, and a batched top-k/top-p filter bug. These issues are purely implementation-side and do not reflect the representational power of the trained weights.

Future work. Our immediate priorities are (i) finalising the decoder fixes and re-evaluating generation quality, (ii) benchmarking safety and toxicity, and (iii) experimenting with contrastive or beam–nucleus hybrid search once the baseline sampler is stable. Longer term, we plan to scale the expert count, add a retrieval

adapter, and explore lightweight RLHF to align stylistic tone on NSFW-heavy corpora.

In sum, `mini_deepseek.py` delivers an open, compact playground for studying MLA + MoE transformers—and, as the decoder stabilises, will serve as a practical template for researchers and students interested in sparse, efficient language models.

References

[1] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language Models are Unsupervised Multitask Learners. *OpenAI Blog*, 2019.

[2] Hongyang Zhang, Geoffrey Hinton, Jimmy Ba. Root Mean Square Layer Normalization. *arXiv preprint arXiv:1910.07467*, 2020.

[3] Sinong Wang, Belinda Li, Madian Khabsa, Han Fang, Hao Ma. Linformer: Self-Attention with Linear Complexity. *arXiv preprint arXiv:2006.04768*, 2020.

[4] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, Jeff Dean. Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer. *ICLR Workshop*, 2017.

[5] DeepSeek-AI Team. DeepSeek-V3 Technical Report. December 2024.

[6] William Fedus, Barret Zoph, and Noam Shazeer. Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity. *ICLR*, 2021.

[7] Nan Du *et al.* Efficient Scaling of Language Models with Mixture-of-Experts. *Nature*, 2022.

[8] Weilin Cai, Juyong Jiang, Fan Wang, and Jing Tang. A Survey on Mixture of Experts. arXiv:2407.06204, 2024.

A Auxiliary: Tools and Libraries

- **Python 3.8+**
- **PyTorch** (`torch`, `torch.nn`, `torch.nn.functional`)
- **Hugging Face Transformers** (`AutoTokenizer`, `get_cosine_schedule_with_warmup`)
- **Datasets** (`load_from_disk`)
- `math`, `typing`, `torch.utils.data.DataLoader`