CS 839 HW1

Ivy Zhu

September 2025

1 GPT-2 parameter counts

1. *Write symbolic formulas for embeddings, attention, MLP, and LayerNorms. Give a total in terms of V, E, H, L, P, where: V = vocabulary size, E = model/embedding dimension, H = number of attention heads, L = number of transformer layers, and P = maximum positional indices (context length). [8 pts]*

   - Embeddings = wte + wpe = V * E + P * E

     wte: lookup the embedding of the input tokens in a V * E matrix.

     wtp: lookup the embedding of the positions of the tokens in a P * E matrix.

   - Attention = c_attn + c_proj + attn_dropout + resid_dropout = $3E^2 + 3E + E^2 + E + 0 + 0 = 4E^2 + 4E$

     - c_attn = $E \times 3E + 1 \times 3E = 3E^2 + 3E$
       This layer projects the normalized embeddings from ln_1 to Q, K, V.
       It's $y = xW^T + b$, where x is the input embedding from the last layer ($1 \times E$), and W^T ($E \times 3E$) and b ($1 \times 3E$) are trainable.
     - c_proj = $E \times E + E = E^2 + E$
       A Conv1D layer that combines the output of attention heads.
       Multi-head attention splits up the embedding dimension, not duplicate it, so the input dimension here is still E. W^T is a $E \times E$ matrix and b is a $1 \times E$ vector.
     - attn_dropout = 0
       A dropout layer, nothing learnable
     - resid_dropout = 0
       A dropout layer, nothing learnable

   - MLP = c_fc + c_proj + act + dropout = $4E^2 + 4E + 4E^2 + E + 0 + 0 = 8E^2 + 5E$

     "Feed-forward layer" or "multi-layer perceptron", applied to each token separately.

- c_fc $= E \times 4E + 4E = 4E^2 + 4E$
  "Up-projecting": turn E into 4*E. So it's a matrix of $E \times 4E$. Plus a bias with size 4*E.
- c_proj $= 4E \times E + E = 4E^2 + E$
  "Down-projecting": turn it back into the dimension of embeddings. So it's a matrix of $4E \times E$. Plus a bias with size E.
- act = 0
  Apply NewGELUActivation, nothing to learn.
- dropout = 0
  Dropout layer, nothing to learn.

• LayerNorms (within each GPT2Block) = ln_1 + ln_2 = 2 * E + 2 * E = 4 * E
  There's also a final LayerNorm layer (ln_f), which is also 2 * E.
  Normalize the input across the embedding dimension. It's calculated for each embedding by $y = \frac{x - E[x]}{\sqrt{Var[x] + \epsilon}} * \gamma + \beta$, and the only trainable parameters are γ and β, which each has the same dimension as the embedding (E).

• Total $= V * E + P * E + L * (12E^2 + 13E) + 2E$

2. *Plug in the hyperparameters for GPT-2. Report totals and a breakdown by component as concrete parameter counts (numbers). Verify with code and explain any mismatch. [6 pts]*

   Hyperparameters:

   • E (embedding dimension) = 768
   • V (vocabulary size) = 50257
   • H (number of attention heads) = 12
   • L (number of transformer layers) = 12
   • P (maximum positional indexes) = 1024

   Per-component parameter counts:

   • Embeddings $= (V + P)E = (50257 + 1024) * 768 = 39,383,808$
   • LayerNorms $= 4E = 4 * 50257 = 3,072$
   • Attention $= 4E^2 + 4E = 2,362,368$
   • MLP $= 8E^2 + 5E = 4,722,432$

   Total per layer = LayerNorms + MLP + Attention = 7,087,872

   Total = Embeddings + L * (Total per layer) + ln_f = 124,439,808

   Counting programatically (in gpt2.py count_params()) gives the same result.

3. *Estimate totals for GPT-2 Medium, GPT-2 Large, and GPT-2 XL using documented (E, H, L). Briefly explain which terms dominate scaling.*

   - GPT-2 Medium: L = 24, E = 1024, H = 16.
     Total = Embeddings + L * (Total per layer) + ln_f = 354,823,168
   - GPT-2 Large: L = 36, E = 1280, H = 20.
     Total = Embeddings + L * (Total per layer) + ln_f = 774,030,080
   - GPT-2 XL: L = 48, E = 1600, H = 25.
     Total = Embeddings + L * (Total per layer) + ln_f = 1,557,611,200

   Scaling explanation:

   - H (number of attention heads) doesn't change the number of parameters because it's only used to split up the embeddings.
   - E (embedding dimension) scales up fast because each layer scales up at $O(E^2)$.
   - L (number of transformer layers) also scales up by linearly scaling up $O(L)$ after the $O(E^2)$.

2 Modern model study

1. *Report the exact configuration you use (cite the source). [4 pts]*

   gpt-oss-20b

   huggingface

   Model Card

   Blog Posts that helped understanding the design and architecture:

   - https://cameronrwolfe.substack.com/p/gpt-oss
   - From GPT-2 to gpt-oss: Analyzing the Architectural Advances

   Configuration

2. *Derive component-wise formulas (as in Problem 1) consistent with the chosen architecture, and compute both per-layer and total parameter counts using your conventions. [20 pts]*

   Hyperparameters:

   - L (number of transformer layers): 24
   - E (embedding dimension) = 2,880
   - V (vocabulary size) = 201,088
   - *head_dim* (Key/Query/Value head dimension) = 64
   - *num_q_heads* (number of query heads) = 64

- num_kv_groups (number of key-value groups) = 8
- num_MoE_blocks (number of MoE blocks) = 32
- $experts\_per\_token = 4$

Parameter counts per component:

- Embedding = $V * E$
- Transformer layer = $E + E * head\_dim * num\_q\_heads + head\_dim * num\_q\_heads + 2 * (E * head\_dim * num\_kv\_groups + head\_dim * num\_kv\_groups) + num\_q\_heads + E + E * num\_MoE\_blocks + num\_MoE\_blocks + num\_MoE\_blocks * (3E^2 + 3E$
  - Input RMSNorm = E
  - Grouped Query Attention = $E * head\_dim * num\_q\_heads + head\_dim * num\_q\_heads + 2 * (E * head\_dim * num\_kv\_groups + head\_dim * num\_kv\_groups) + num\_q\_heads$
    - Q projection = $E * head\_dim * num\_q\_heads + head\_dim * num\_q\_heads$
      Project dimension of E to dimension of Query Dimension * Number of query heads.
      So there's a matrix of $E * head\_dim * num\_q\_heads +$ a vector of $head\_dim * num\_q\_heads$ (bias)
    - K projection = $E * head\_dim * num\_kv\_groups + head\_dim * num\_kv\_groups$
      Project dimension of E to dimension of Query Dimension * Number of key-value groups.
      So there's a matrix of $E * head\_dim * num\_kv\_groups$ + a vector of $head\_dim * num\_kv\_groups$ (bias)
    - V projection = $E * head\_dim * num\_kv\_groups + head\_dim * num\_kv\_groups$
      Same as K
    - Output linear projection = $head\_dim * num\_q\_heads * E + E$
      Project the attention heads back onto the dimension of embeddings.
    - Learned bias for each attention head = num_q_heads
      According to the model card, "Each attention head has a learned bias in the denominator of the softmax, similar to off-by-one attention and attention sinks, which enables the attention mechanism to pay no attention to any tokens".
  - Post attention RMSNorm = E
  - MLP/MoE = $E * num\_MoE\_blocks + num\_MoE\_blocks + num\_MoE\_blocks * (3E^2 + 3E)$
    - router = $E * num\_MoE\_blocks + num\_MoE\_blocks$
      The router selects a small set of feed forward modules for each token.
      gpt-oss uses a standard linear router projection.

* num_MoE_blocks * Each Feed Forward
  For each Feed Forward: $3E^2 + 3E$ parameters
  · Linear Layer: $E * E + E = E^2 + E$ parameters
    There are three linear layers.

  Not all are always active.

- Final RMSNorm $= E$

  RoPE is applied on every transformer layer, not only on the input. However, unlike the positional embedding in GPT2, RoPE is not learned, so there's no parameters in RoPE.

Calculation:

- Token Embeddings: 579,133,440
- Attention: 26,550,144
  - Q_proj: 11,800,576
  - K_proj: 1,475,072
  - V_proj: 1,475,072
  - Out_proj: 11,799,360
  - Bias per head: 64
- MLP: 796,631,072
  - Router: 92,192
  - Each expert/feed forward: 24,891,840
- each RMSNorm: 2,880

Per layer parameter count: 823,186,976

Total = Embedding + L * Per layer count + final RMSNorm = 20,335,622,208

Total $= V * E + L * (E + E * head\_dim * num\_q\_heads + head\_dim * num\_q\_heads + 2 * (E * head\_dim * num\_kv\_groups + head\_dim * num\_kv\_groups) + num\_q\_heads + E + E * num\_MoE\_blocks + num\_MoE\_blocks + num\_MoE\_blocks * (3E^2 + 3E) + E$

Calculation code is submitted in gpt-oss.py

The total calculation doesn't include unembed because it's not in the configuration. However, unembed is included in the model card, which causes the difference in the total.

3. *Report the final totals as concrete parameter counts (numbers). If MoE, also report active parameters per token. [12 pts]*

Total (without unembed) = 20,335,622,208

Total (with unembed) = 20,914,757,184

Active Parameters per token: only 4 expert per token.

- active_MLP = router + expert_per_token * each_expert = 99,659,552

- active_per_layer = RMSNorm + Attention + RMSNorm + active_MLP = 126,215,456
- active_total = embed + L * active_per_layer + RMSNorm = 3,608,307,264

4. *Provide code that demonstrates a programmatic check. [12 pts]*

   Programmatic check is submitted in gpt-oss.py

5. *Identify the key components that differ from GPT-2. For each:*

   - *Summarize the design motivation in your own words. [12 pts]*
   - *Provide evidence of effectiveness from the paper/tech report (ablation, benchmark, or efficiency claim). [12 pts]*

   Key component differences:

   (a) Positional embedding: GPT-2 uses a learnable positional embedding layer, while GPT-oss uses RoPE, which doesn't have any learnable parameters, and supports bigger context length. It provides a bigger context length, reduces the number of learnable parameters, and also supports relative positions.

   Evidence: longer context length. GPT-2 has 1024 maximum positional indexes and needs to learn the parameters for positional embedding, while GPT-oss has 131,072 maximim position embeddings without the need of training it.

   (b) Normalization layers: GPT-2 uses LayerNorm, while GPT-oss uses RMSNorm (root mean square normalization), which reduces the cost in computing mean and variance. RMSNorm is becoming popular recently because of its lower computational cost.

   Evidence: According to Root Mean Square Layer Normalization, "RMSNorm yields comparable performance against LayerNorm but shows superiority in terms of running speed with a speed-up of 7%-64%."

   (c) Attention: GPT-2 uses multihead attention, while GPT-oss uses grouped query attention, which reduced the number of key-value pairs, and also reduced the number of parameters required to calculate keys and values.

   Evidence: It reduced the number of parameters of projecting V and K from 11,800,576 (if use multi-head) to 1,475,072.

   Also, unlike GPT-2, which keeps the total dimension of all query heads the same as the dimension of embeddings, GPT-oss expanded the total query dimension to 64 *64 = 4,096, which is bigger than the dimension of embedding (2,880). The motivation might be to provide bigger exposure to the attention.

   Evidence: https://arxiv.org/html/2508.16700v2, compared to dense baselines, GPT-oss-20B has higher throughput and lower energy consumption.

(d) Feed forward: GPT-2 contains a simple feed foward module, while GPT-oss-20B has a MoE module that contains 32 individual experts, with a router layer that decides which experts to be used for each token. The MoE architecture increases the performance by including a large number of experts, while keeping the computation low by only activate a small subset of them for each token.

Evidence: OpenAI claims that "Each model is a Transformer which leverages mixture-of-experts (MoE) to reduce the number of active parameters needed to process input".

6. *Give a short overall summary (bullets or a short paragraph) of the main design trends you observe. [8 pts]*

- Overall, the architecture design has remained very similar, with the same types of basic components (embedding, normalization layers, attention, and MLP), but each with improvements.
- The number of layers has decreased, while the complexity of each layer has increased.
- The size and portion of MLP has significantly increased, but there's also effort to keep computational cost low while retain the advantage of dense MLP.
- There's effort on changing individual modules to reduce computational cost or to make it more parallel-friendly, and removing modules that has been proving to have worse effect than expected.
- Context length has increased significantly.