Assignment 5: Capacity

CS 312 — Ungraded · Due: November 10, 1pm (quiz time)

Basic setting. Unless a problem states otherwise, use the course next-token prediction setup on the provided DCLM-style data. Compare models under the same data, token budget, and evaluation protocol. Lower validation loss is better. In this assignment, also account for training compute, inference compute, and stored parameters separately.

Instructions. Submit a single PDF report to Gradescope by the deadline above, with a maximum of 20 pages. As in A4, this assignment is ungraded; you may cite your report for partial credit on the quiz.

For each problem, analyze the supplied experiments and any new experiments you ran, then write a short, approximately one-paragraph synthesis. Report the comparisons, hyperparameters, best validation losses, and diagnostic evidence. Give plots clear axes and labels. Tune central hyperparameters for new variants and record divergent runs as well as successful ones. State the FLOP-counting convention used in efficiency comparisons.

Code release and provided runs. Use the `a5_release` branch, under `experiments/a5_capacity/`. Short model files use the native transformer and training recipe; metrics run separately on checkpoints.

Problem	Supplied material	Suggested new runs
Depth vs width	35 ratio-grid shapes, additional shapes and matched extremes	0
Low-rank model	Dense reference; head/CPLM code and gradient-energy metric	∼20
MoE	Dense reference; router/balancing code and health metrics	∼40
Looping	Dense reference; sharing/norm/recall/switching code and signal metrics	∼40

Plan for roughly **100 new full-budget training runs**, with most devoted to MoE and looping. Analyze the supplied depth/width sweeps without repeating them. For low rank, spend a smaller budget on the rank comparison and the three head interventions, plus roughly 4–6 short SpamLang runs.

These estimates include targeted LR trials and a few repeat seeds for central comparisons. Start new variants from a known LR and tune selectively. Reuse trained checkpoints for diagnostics and for evaluating loop counts not used during training; these evaluations do not count as new training runs.

Released depth/width references. Completed histories and retained checkpoints are available at:

- **W&B project:** `hashimoto-group/public-alchemy-a5`, tag `a5-capacity-v1`.
- **Modal directory:** `/root/shared_data/a5_capacity/v1`, mounted read-only from volume `hard-dl-dclm-v1` in environment `cs312-shared-data`.

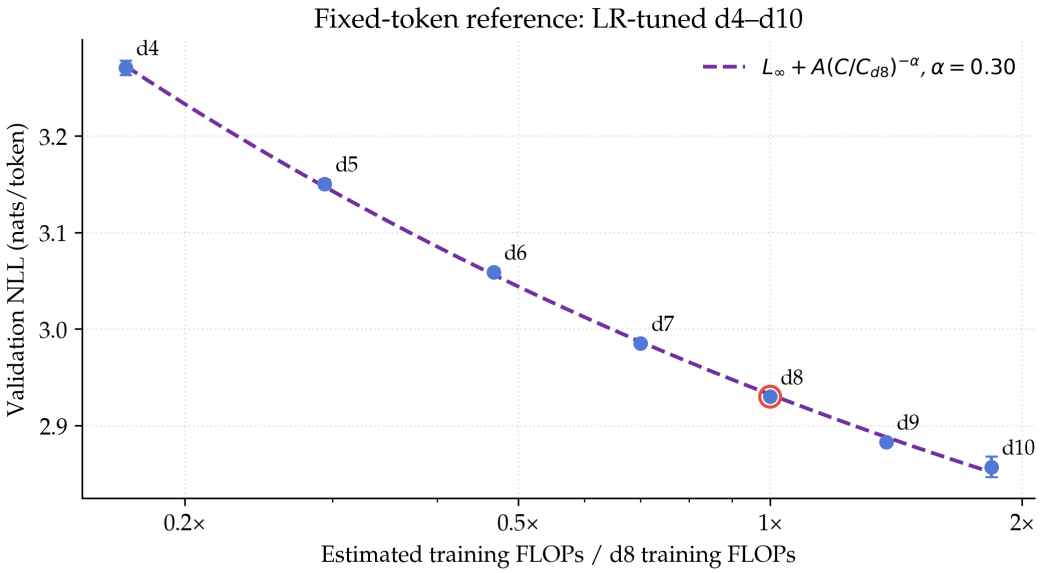
All completed LR histories and retained checkpoints are supplied, including both parameter-matched extremes. **We are still training the full grid for all of the width/depth combinations.** The available release is sufficient for Problem 1; you do not need to wait for additional trials. This is **not a complete five-LR sweep at every shape**: search budgets differ, and each extreme uses LR 0.003 and seed 42. More completed trials are appended to `manifest.json`, which records counts, publication time, architectures, losses, public links and checkpoint checksums. Pruned trials retain their histories. W&B is public; Modal requires course-workspace access.

Use `p1_depth_width.py` and `measure_default.py` in the code folder for reference loading and diagnostics. Record the manifest publication time and run IDs. Other problems provide code, not promised training runs.

Reasoning about capacity

We will reason about where the transformer is limited in capacity and how to increase its expressivity. We now also care about compute efficiency: an improvement obtained with more parameters or computation needs to be compared with spending those resources on a larger ordinary model.

For context here, is a reference ladder that trains models d4 through d10. Model dD has D blocks and width $64D$, with its own LR search at a fixed budget of 600,000 sequences. Note that proper compute-optimal pre-training such as Hoffmann et al. (2022) would also vary the data budget across scale.



614.4M token slots; width = 64 × depth. Error bars: seed SD at fixed selected LRs. Fit shown only inside the measured range.

Let $\widehat{L}_{\text{ref}}(C) = L_\infty + A(C/C_0)^{-\alpha}$ be a fitted reference curve in estimated training FLOPs, where C_0 is the d8 cost. Its inverse gives the *dense-equivalent cost*

$$C_{\text{eq}}(L) = C_0 \left(\frac{A}{L - L_\infty} \right)^{1/\alpha}.$$

For methods a, b with losses L_a, L_b and actual estimated costs C_a, C_b, define efficiencies relative to this yardstick by

$$\epsilon_a = \frac{C_{\text{eq}}(L_a)}{C_a}, \qquad \frac{\epsilon_b}{\epsilon_a} = \frac{C_{\text{eq}}(L_b)}{C_{\text{eq}}(L_a)} \frac{C_a}{C_b}.$$

At equal cost, a lower-loss method has a larger dense-equivalent cost and a larger efficiency.

We will explore four directions:

- Allocate parameters between depth and width.
- Modify the hidden-to-vocabulary readout and its gradient path.
- Store more parameters while activating only a few experts.
- Reuse parameters across depth through looping.

Depth versus width

Problem 1: Depth vs width

We will explore depth and width at the standard 600,000-sequence budget. Use the supplied runs and checkpoints; no new training is expected for this problem. The supplied ratio grid has depths $D \in \{4, 5, 6, 7, 8, 9, 10\}$ and widths

$$W = \rho D, \qquad \rho \in \{16, 32, 64, 128, 256\}.$$

These are $\{1/4, 1/2, 1, 2, 4\}$ times each depth's default width $64D$. Attention uses 64-dimensional heads where possible; narrow widths use head dimension $\gcd(W, 64)$ to preserve the projection width.

(a) Visualize every architecture's best available loss against estimated training FLOPs.

(i) Starting from d8, suppose extra compute may buy only depth or only width. Which direction looks more useful at several budgets?

(ii) In addition to the previous item, we can take the extreme where we can take $W \to \infty$ at fixed depth or $D \to \infty$ at fixed width. In theory, both can express any target function. Is either of the limits preferable?

(iii) If you commit to a fixed ratio $W/D = \rho$, which ratio would you choose? Does the answer change with budget?

(b) In the extreme case, we can take our d8 and compare an extra wide network (1 block with width 1280) with an extra skinny and deep network (62 blocks with width 192). Both alternatives are roughly parameter matched: 34.25M and 34.75M parameters, respectively, versus d8's 34.61M. Use their supplied runs at LR 0.003 along with the same-LR d8 control. In this case, which option yields lower loss? Does this agree with the previous part?

Readout bottlenecks

Our vocabulary has 4096 entries but hidden width 512. Across contexts, a linear head $z = Wh$ limits logit rank to 512 and log-probability rank to at most 513: the softmax bottleneck.

In this problem, we will measure this bottleneck and try to understand whether addressing the bottleneck can improve our downstream performance. For the sake of analysis, we can artificially strengthen this bottleneck with the same backbone to rank r by reparameterizing the head as $W_r = A_r B_r$ with $A_r \in \mathbb{R}^{4096\times r}$ and $B_r \in \mathbb{R}^{r\times 512}$.

Problem 2: Low-rank model

Compare ranks $r \in \{32, 64, 128, 256\}$ and the full head on native d8. Reuse the supplied dense baseline and train targeted head variants, starting from a known LR.

(1) **Measuring the bottleneck.** We will follow the treatment in Godey and Artzi (2026).

(i) **Gradient energy.** On matched held-out positions, compute $g_t = \text{softmax}(z_t) - e_{x_{t+1}}$ and measure the fraction of gradient energy outside the head's column space:

$$E_r = \frac{\sum_t \|g_t - UU^\top g_t\|_2^2}{\sum_t \|g_t\|_2^2},$$

where U (from the head SVD $W_r = U\Sigma V^\top$) spans the head's column space (use at most r left singular vectors for a factorized head). Plot E_r against rank and compare with the independent random-subspace reference $1 - \text{rank}(W_r)/4096$. What fraction of the energy lies outside the head's subspace?

(ii) **SpamLang.** Draw $u \sim \text{Uniform}(\{1, \ldots, V\})$ independently for each sequence and repeat it 64 times. After the first symbol, next-token prediction is deterministic. A low-rank head can represent this copying rule, so difficulty learning it need not imply insufficient expressivity.

Use a one-layer, four-head model, 102,400 training sequences, 2,048 independent validation sequences, and 800 updates at batch size 128. Compare vocabularies at width 64 and widths at vocabulary 4096 using `training_config()` in `p2_readout.py`. Plot loss curves with the $\log V$ reference. When does the model struggle or succeed on SpamLang? Does width rescue larger vocabularies? Account for independent draws per symbol, rather than counting repetitions as new draws.

(2) **Addressing the bottleneck.** Try all three methods with full and constrained vocabulary heads:

(i) **MoS:** mix four softmax distributions, each with rank-r logits and learned mixture weights $\pi(h) = \text{softmax}(Gh)$:

$$p(y \mid h) = \sum_{j=1}^{4} \pi_j(h) \, \text{softmax}(A_{j,r} B_{j,r} h)_y.$$

Hopefully, this nonlinear mixture of softmaxes can increase our effective width.

(ii) **Sigsoftmax:** for the same logits $z = W_r h$, use $p = \text{softmax}(z + \log \sigma(z))$, where σ is the sigmoid; see Kanai et al. (2018). Hopefully, this nonlinearity can reduce the representational bottleneck of a low rank matrix.

(iii) **CPLM (Copy-Paste Language Model):** add a second prediction path that copies tokens from the prefix instead of predicting only through the fixed vocabulary head. A learned query/key pointer scores causal positions $i \leq t$ and a learned sink, then normalizes these scores together. Sum its attention mass over occurrences of each token, $m_t(y) = \sum_{i \leq t : x_i = y} a_{ti}$, and mix with the vocabulary LM:

$$p(y) = \left(1 - c_t + c_t a_{t\varnothing}\right) p_{\text{LM}}(y) + c_t m_t(y).$$

Here c_t is the probability of a learned `<copy>` option in a softmax over that option and the vocabulary, and $a_{t\varnothing}$ is the pointer's sink probability. The sink returns its mass to the LM. This adds a gradient path through the pointer as well as the vocabulary head. Use the supplied gate/sink design from Godey and Artzi (2026). Constrain only the vocabulary branch; keep pointer width 512 and full-width hidden inputs fixed across ranks.

Plot loss and improvement over ordinary softmax against the (potentially additionally constrained) rank. How much does each method help as a function of rank? By looking at the losses, how much does each method increase effective rank by?

Mixture of experts

Though we would like to train larger models, this comes with the tradeoff of costing more compute. Sparsely activated models allow us to increase parameter count while keeping expert compute per token fixed, apart from routing overhead. In this problem, we will explore an incredibly primitive MoE in the context of our d8. Note: Due to our suboptimal implementation, we will not achieve identical run throughput. In principle, this is possible with proper performance optimizations that we did not implement, so track FLOPs instead of end-to-end runtime (as ML researchers, we can heed this reminder).

Problem 3: Mixture of experts

Replace each MLP with four full-size SwiGLU experts, activating one per token. Keep attention, width, depth and tokens fixed.

(a) **Simple routing.** Compare dense, uniform random top-1, frozen arbitrary top-1, learned top-1, and a full-four-expert mixture that evaluates all experts. The frozen control uses a random scoring matrix held fixed throughout training; backbone and expert weights still learn.

For learned top-1, use $p = \text{softmax}(s)$, $e = \arg\max_j p_j$, and output $4p_eE_e(x)$. Random routing uses unit weights. Include a matched-combine control; hard argmax with unit weights passes no ordinary task gradient to the router. Account for the full mixture's extra compute.

(b) **Improve the router.** Explore controlled changes to the architecture. You should study these methods in isolation and when composed with each other.

(i) **Auxiliary load balancing:** add $\lambda\, 4 \sum_e f_e \bar{p}_e$, where f_e is the detached assignment fraction and $\bar{p}_e$ the mean softmax probability. Sweep its coefficient; see Switch Transformers.

(ii) **Normalized routing:** apply RMSNorm to the hidden state before the router's linear projection, keeping the experts' inputs unchanged. This controls the scale of the routing logits.

(iii) **Noisy routing:** add zero-mean Gaussian noise to the routing logits during training, before softmax and expert selection. Vary the noise scale; disable noise at evaluation.

(iv) **Shared initialization:** initialize all four experts with identical MLP weights, then let them learn independently. This starts routing from experts that compute the same function.

(v) **Quantile balancing:** follow Jianlin Su's derivation and Marin's study. Route with a bias estimated from previous steps, update it without gradients, and compute combine weights from unbiased scores. Aggregate over the effective batch consistently.

In addition to the loss of the MoE, track **three router diagnostics**:

(i) **Load imbalance:** $\text{std}_e(f_e) / \text{mean}_e(f_e)$, where f_e is the selected-assignment fraction.

(ii) **Normalized probability entropy:** $\mathbb{E}_t[-\sum_e p_{t,e} \log p_{t,e}] / \log 4$.

(iii) **Mean largest routing probability:** $\mathbb{E}_t[\max_e p_{t,e}]$.

For quantile routing, use unbiased softmax probabilities.

(c) **Design your own router variants.** Propose and implement your own router variants. Predict their mechanisms and effects, then compare with a matched control from (a) or (b), using held-out loss and routing diagnostics. Try to understand which methods fail or succeed to build a mental model for what interventions are most important for an MoE.

Looping

Coming from the opposite direction of MoE's, we are curious about whether we can maintain the same capacity while reducing the total parameter count. A looped model achieves this by taking a single layer and applying it multiple times sequentially instead of having unique parameters. In this problem, we will explore what we gain or lose from utilizing looping.

Problem 4: Looping

Reduce distinct parameters while keeping block applications fixed. Protect one initial block (the *prelude*) and one final block (the *coda*). Six middle applications use a core of $6/R$ blocks repeated R times:

$$h_0 = P(E[x]), \qquad h_{k+1} = F_{6/R}(h_k), \quad k = 0, \ldots, R-1, \qquad z = W_o N(C(h_R)).$$

$F_{6/R}$ is a stack of ordinary residual transformer blocks. There is no additional outer residual unless explicitly named.

Passes R	Distinct core blocks	Distinct total blocks	Applications
1	6	8	8
2	3	5	8
3	2	4	8
6	1	3	8

(a) **Capacity at fixed applications.** Train or analyze these four sharing patterns, tuning LR separately. $R = 1$ is ordinary d8. Compare loss, stored parameters and compute. As a separate checkpoint test, evaluate fewer/more passes, recording the changed inference compute.

(b) **Signal propagation.** Compare activation RMS, gradient RMS, and relative residual-update size, per pass with ordinary d8. Then, to improve the signal propagation, consider utilizing the boundary map:

$$B(h) = \text{RMSNorm}(h) + \alpha a.$$

Here $a = h_0$ is the prelude output, reused at every boundary, and α controls its injection strength. Compare identity, norm alone, injection alone, and both. Identify which interventions work best and track the same statistics as before. While keeping the architecture fixed to 1 prelude and 1 coda, try additional looping designs that may further improve performance. As a canonical setting, you can consider a single block that gets looped 6 times.

(c) **Test generalization.** Take the looped models we've trained and try evaluating them at loop counts we haven't trained at. Which loop counts perform best for our model?

(d) **Extra deep looping.** Fix our setting to three unique blocks: the prelude, coda, and core. First, train a series of models with different inner loop counts to extrapolate loss as loop count increases significantly (extrapolate to at least 18 inner applications of the core). Then, try to come up with architectural designs that can improve the hypothetical depth 20 architecture.

(e) **Switching sharing during training.** At the midpoint of a standard run, switch in both directions: (i) a six-pass, one-block core to six independent middle blocks, and (ii) an ordinary d8 to a six-pass, one-block core. Keep one prelude and one coda. For (i), initialize each independent block with a separate copy of the shared block's parameters. For (ii), tie the middle blocks using their parameter-wise mean, or justify another rule. Keep any boundary maps unchanged through the switch and reset optimizer state. Include two controls that reset Adam at the same point but keep their original sharing pattern or controls where you start with random weights during the second phase instead of getting the first phase. Does it help to start with sharing and specialize later, or to start with unique layers and compress later? Use the supplied `loop-to-unique`, `unique-to-loop`, `loop-reset`, and `unique-reset` presets in `sweeps.py`; each switches after update 4,688 of 9,375. Build an intuition for when this switching operation could be beneficial if we already have a model lying around.

Optional exploration

As in A3, use an observation from your experiments to design a new outcome-prediction question. This optional problem can earn quiz bonus credit if staff use your question, including a modified version, on a future quiz.

Problem 5: Your own prediction problem

Develop a question about capacity that tests an insight beyond the comparisons already specified. Budget roughly 5–10 runs if you choose to investigate a new direction; existing checkpoints and short continuations may suffice.

(a) Write a self-contained numerical or multiple-choice question, in the style of earlier quizzes. Specify the architecture, training recipe, comparison and metric so that executing the experiment determines an objective answer. Explain your prediction using evidence from this assignment.

(b) Supply a compact code diff (fewer than 100 lines) and executable code that produces the answer. Include the measured result and a brief explanation of whether it agrees with your prediction. Use a matched control and record the hyperparameters needed to reproduce the comparison.

Aim for a question that is surprising but fair: another student should be able to reason about its outcome using what this assignment teaches.