CS336 Assignment 3 (scaling): Scaling Laws

Version `1.0.0`

CS336 Staff

Spring 2025

1 Assignment Overview

In this assignment, you will gain some hands-on experience with language model scaling laws.

The setting. You are in charge of training ClosedAI's next-generation language model. This model will take the GDP of a small country to train and melt an Antarctic ice shelf, so you want to make sure you get it right. Concretely, you are given a fixed compute budget (measured in FLOPs), and your goal is to produce a model with the lowest training loss (i.e., a *compute-optimal* model). To achieve this goal, you will need to figure out how to best tradeoff (1) training a larger model versus (2) training on more tokens.

Scaling laws, which empirically relate language model training loss to model size and the amount of compute used for training, are frequently used to project this trade-off [Kaplan et al., 2020, Hoffmann et al., 2022]. In this assignment, you will construct scaling laws to estimate the compute-optimal model size and its corresponding hyperparameters for a modest FLOPs budget of 1e19. Rather than training the models yourself, you will query a "training API" with (1) model hyperparameters (i.e., number of Transformer layers, embedding size, number of heads, batch size, learning rate) and (2) the desired training FLOPs; the API will return the final training loss from training on the corresponding number of tokens. The FLOPs budget for fitting our scaling laws is an additional 2e18 FLOPs (20% of our big run FLOPs budget). The training API accepts a wide range of hyperparameter values, and it is your part of your job to figure out what parts of the search space to explore in order to effectively use the scaling laws budget.

What the code looks like. This writeup is available on GitHub at:

`github.com/stanford-cs336/assignment3-scaling`

Please `git clone` the repository. If there are any updates, we will notify you and you can `git pull` to get the latest.

How to submit. You will submit the following files to Gradescope:

- `writeup.pdf`: A complete description of your methodology for fitting a scaling law and using it to predict the optimal model size for the given FLOPs budget. The write-up should be detailed enough to reproduce your results.
- `code.zip`: Contains all the code you've written to fit your scaling law and compute your estimates.

In addition, submit your (1) predicted optimal model size, (2) the training hyperparmeters to use, and (3) the model's predicted training loss to this Google form: https://forms.gle/sAUSLwCUETew2hYN6

Part of your grade on the assignment will be determined by the performance of your predicted optimal model.

2 Scaling Laws Review

We will review one of the approaches for fitting scaling laws from the Chinchilla paper [Hoffmann et al., 2022]. The central question is: given a compute budget C, which we will use to train a large language model, which choice of hyperparameters (model size, number of training tokens, etc.) will lead to the lowest training loss? The main challenge is how to extrapolate from experiments done at a smaller scale to larger scale. For your own work in the second part of the assignment, you are welcome to incorporate ideas from other references, such as Kaplan et al. [2020] and Yang et al. [2022].

2.1 Scaling Laws from IsoFLOPs profiles

Recall that the compute budget for training a Transformer with N parameters on a dataset of D tokens is approximately $C = 6ND$. The IsoFLOPs approach to scaling laws in Hoffmann et al. [2022] works as follows: for each compute budget C, train language models of varying sizes N given compute budget C (with data size $D = C/(6N)$), producing a final training loss L.

This produces a set of runs that used the same number of FLOPs C_i, but have varying model sizes N_{ij}. Empirically, Hoffmann et al. [2022] observe a quadratic relationship between the final training loss L_{ij} and model size N_{ij} given a fixed compute budget C_i. One intuition for this is as follows: When N_i is extremely small, the model can't fit the data regardless of how much compute we spend, so the final training loss in this regime is high. As we increase model size, the final training loss drops smoothly, until a certain point where our model starts to be too large and we can't afford to take enough gradient steps within C FLOPs to train it effectively (as an extreme example, as $N_i \to \infty$, even taking a single gradient step would exceed our compute budget C, and training would thus stop at very high loss). The method for finding scaling laws will consist of determining the optimal model and dataset sizes for each C_i, then fitting a power law that predicts N and D given a target C, often a larger budget than what was used in the previous runs.

To fit the power laws, we use data points given by finding the model size $N_{opt}(C_i)$ that minimizes the training loss for each budget C_i, using the set of runs that used that budget (its "IsoFLOPs profile"). This procedure gives us a sequence of pairs $\langle C_i, N_{opt}(C_i)\rangle$ (and an analogous one for D_{opt}). We use these data points to fit power laws $N_{opt} \propto C^a$ and $D_{opt} \propto C^b$, and use these to extrapolate the compute-optimal model and dataset sizes to our target compute budget.

Problem (`chinchilla_isoflops`): 5 points

Write a script to reproduce the IsoFLOPs method describe above for fitting scaling laws using the final training loss from a set of training runs. For this problem, use the (synthetic) data from training runs given in the file `data/isoflops_curves.json`. This file contains a JSON array, where each element is an object describing a training run. Here are the first two runs for illustrating the format:

```
[
    {
        "parameters": 49999999,
        "compute_budget": 6e+18,
        "final_loss": 7.192784500319437
    },
    {
        "parameters": 78730505,
        "compute_budget": 6e+18,
        "final_loss": 6.750171320661809
    },
```

```
    ...
]
```

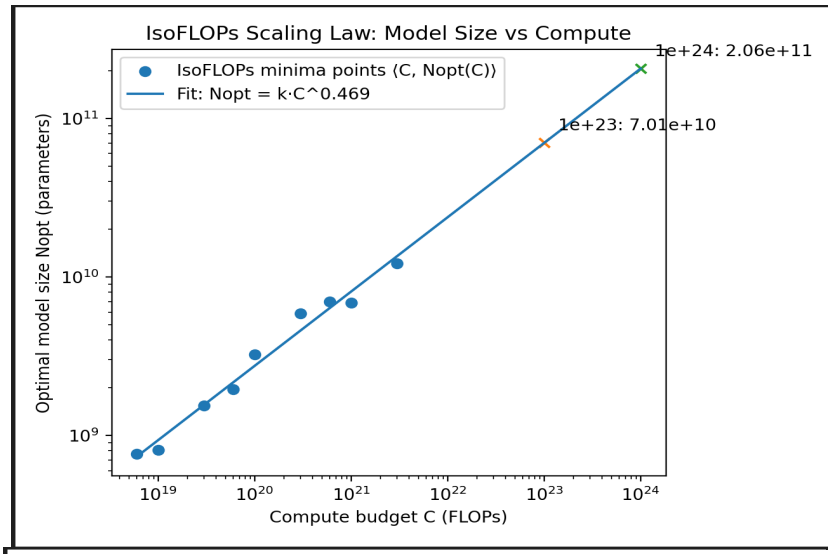
For fitting the scaling laws, the `scipy` package (and `scipy.optimize.curve_fit` in particular) might be useful, but you're welcome to use any curve fitting method you'd like. While Hoffmann et al. [2022] fits a quadratic function to each IsoFLOP profile to find its minimum, we instead recommend you simply take the run with the lowest training loss for each compute budget as the minimum.

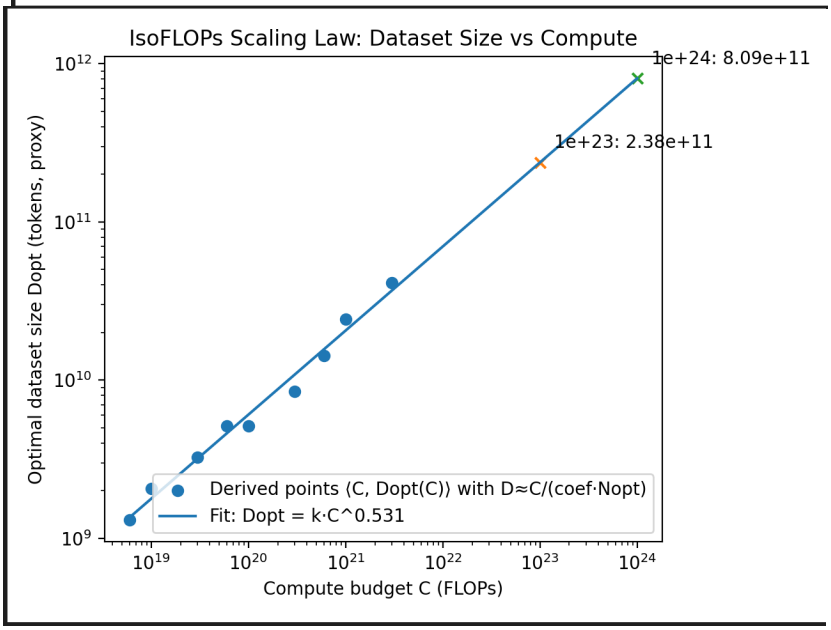
1. Show your extrapolated compute-optimal model size, together with the $\langle C_i, N_{opt}(C_i)\rangle$ points you obtained. What is your predicted optimal model size for a budget of 10^{23} FLOPs? What about for 10^{24} FLOPs?

   Deliverable: A plot showing your scaling law for model size by compute budget, showing the data points used to fit the scaling law and extrapolating up to at least 10^{24} FLOPs. Then, a one-sentence response with your predicted optimal model size.

2. Show your extrapolated compute-optimal dataset size, together with the $\langle C_i, D_{opt}(C_i)\rangle$ data points from the training runs. What is your predicted optimal dataset size for budgets of 10^{23} and 10^{24} FLOPs?

   Deliverable: A plot showing your scaling law for dataset size by compute budget, showing the data points used to fit the scaling law and extrapolating up to at least 10^{24} FLOPs. Then, a one-sentence response with your predicted optimal dataset size.





3 Constructing Scaling Laws

To construct your scaling laws, you will use our training API to query training results for experimental configurations (§3.1). Your goal is to use the scaling law to accurately predict the optimal model size and the associated training loss for a FLOPs budget of 1e19. To set the hyperparameters for the predicted optimal model size, we recommend analyzing how hyperparameters affect the training loss for smaller-scale settings. Be sure to carefully plan your runs before you get started—once you exceed the scaling laws budget of 2e18, the training API will refuse further requests.

Construct a scaling law to accurately predict the optimal model size, its hyperparameters, and the associated training loss for a FLOPs budget of 1e19. To construct your scaling laws, you will use our training API to query the final training loss for various experimental configurations (§3.1); you may not query more than 2e18 FLOPs worth of experiments for fitting your scaling law. This is hard cap that will be enforced by the API.

Deliverable: A typeset write-up that contains a complete description of your approach and methodology for fitting a scaling law. In addition, it should describe how you use the scaling law to predict the optimal model size for the given FLOPs budget, and your predicted values. The write-up should include commentary about why you made particular design decisions, and the description should be detailed enough to reproduce your approach and results.

Note on batch size: We place essentially no constraints on the hyperparameters you may report under the FLOPs budget of 1e19, other than the following requirement: **your batch size must be either 128 or 256.** This is done to ensure that runs have reasonably high model FLOPs utilization. If we have issues with out-of-memory errors when running your reported hyperparameter configuration, we will either use gradient accumulation or scale the number of data parallel GPUs to maintain your desired batch size.

To help you get started, we recommend thinking about at least the following questions. Your writeup should contain additional commentary about how decisions were made for each factor below:

- Given your fixed scaling laws budget of 2e18, how did you decide which runs to query?
- How did you fit your scaling law? Describe the concrete method or methods you used. In particular, it will likely to be useful to familiarize yourself with the approaches used in Kaplan et al. [2020] and Hoffmann et al. [2022].
- How well does your scaling law fit the experimental data?
- For our given FLOPs budget of 1e19, what optimal model size does your scaling law predict? What is the predicted loss?
- If you were to train a model with your predicted optimal number of parameters, what hyperparameters would you use? To estimate the number of non-embedding parameters for a given model hyperparameter configuration, use $12 n_{\text{layer}} d_{\text{model}}^2$.

In addition to the report, submit your (1) predicted optimal model size, (2) the training hyperparameters to use including either batch size 128 or 256, and (3) the model's training loss to this Google form: https://forms.gle/sAUSLwCUETew2hYN6. Part of your grade on the assignment will be determined by the performance of your predicted optimal model.

3.1 Training API

You will use this training API to query the final training loss for scaling law experiments that you'd like to run. Your API key is the same as the SSH public key that you gave us at the start of the quarter (without any newlines). Note that you must be on the Stanford network to query this API, so you may have to use a VPN. As a sanity check, you should be able to see the API documentation page at http://hyperturing.stanford.edu:8000/docs and validate that your API key works with the `/total_flops_used` endpoint (see below for more details). The training API has the following endpoints:

`GET /loss`: given the specifications of a training run, return the final training loss. This endpoint has the following parameters:

- `d_model`: An integer value between [64, 1024].
- `num_layers`: An integer value between [2, 24].
- `num_heads`: An integer value between [2, 16].
- `batch_size`: An integer value, one of {128, 256}.
- `learning_rate`: A float value between [1e-3, 1e-4].
- `train_flops`: An integer value, one of {1e13, 3e13, 6e13, 1e14, 3e14, 6e14, 1e15, 3e15, 6e15, 1e16, 3e16, 6e16, 1e17, 3e17, 6e17, 1e18}.
- `api_key`: your string API key.

Note that querying this endpoint with a previously-queried experimental configuration does not incur extra FLOPs used toward your budget. This endpoint returns a JSON object with the following keys and values:

`loss`: A float training loss, the result of training with the given experiment configuration.

`total_flops_used`: A float with the total number of FLOPs used by all queries issued by your API key.

If a hyperparameter value is not within the acceptable range, a 404 response is returned with an accompanying message. To query the API, issue an HTTP GET request (output truncated for legibility):

```
>>> import requests
>>> config = {
...     "d_model": 1024,
...     "num_layers": 24,
...     "num_heads": 16,
...     "batch_size": 128,
...     "learning_rate": 0.001,
...     "train_flops": int(1e16),
...     "api_key": <YOUR_API_KEY_HERE>
... }
>>> requests.get("http://hyperturing.stanford.edu:8000/loss", config).json()
{'loss': 9.07103488440561, 'total_flops_used': 10000000000000000}
>>> invalid_config = {
...     "d_model": 9999,
...     "num_layers": 24,
...     "num_heads": 16,
...     "batch_size": 128,
...     "learning_rate": 0.001,
...     "train_flops": int(1e16),
...     "api_key": <YOUR_API_KEY_HERE>
... }
>>> requests.get("http://hyperturing.stanford.edu:8000/loss", invalid_config).json()
{'message': 'd_model must be in range [64, 1024], got 9999'}
```

`GET /total_flops_used`: Given an API key, returns a single float—the total number of FLOPs that this API key has used across all of its previously-queried runs. This endpoint has the following parameters:

- `api_key`: your string API key.

If an API key has no queries, a 422 response is returned with an accompanying message. To query the API, issue an HTTP GET request (output truncated for legibility):

```
>>> import requests
>>> requests.get("http://hyperturing.stanford.edu:8000/total_flops_used",
↪  {"api_key": <YOUR_API_KEY_HERE>}).json()
10000000000000000
>>> requests.get("http://hyperturing.stanford.edu:8000/total_flops_used",
↪  {"api_key": "<INVALID_API_KEY>"}).json()
{'message': 'Invalid API key provided: INVALID_API_KEY'}
```

`GET /previous_runs`: Given an API key, returns a list of all previous runs queried with that API key. This endpoint has the following parameters:

- `api_key`: your string API key.

This endpoint returns a JSON object with a key `previous_runs` containing an array of all your previous run configurations and their results.

To query the API, issue an HTTP GET request:

```
>>> import requests
>>> requests.get("http://hyperturing.stanford.edu:8000/previous_runs", {"api_key":
↪  <YOUR_API_KEY_HERE>}).json()
{'previous_runs': [{'d_model': 1024, 'num_layers': 24, 'num_heads': 16,
↪  'batch_size': 128, 'learning_rate': 0.001, 'train_flops': 10000000000000000,
↪  'loss': 9.07103488440561}]}
>>> requests.get("http://hyperturing.stanford.edu:8000/previous_runs", {"api_key":
↪  "<INVALID_API_KEY>"}).json()
{'message': 'Invalid API key provided: INVALID_API_KEY'}
```

3.2 Training Run Details

The training runs were performed with a Transformer language model. The model architecture largely matches that of assignment 1, with a few slight differences. We provide the code for this model in the file `cs336_scaling/model.py` for your reference.

The key differences versus the assignment 1 model are the use of:

1. Absolute position embeddings instead of Rotary Position Embeddings (RoPE).
2. Layer normalization instead of RMSNorm.
3. Feedforward networks composed of a linear layer, a GeLU nonlinearity, and another linear layer, instead of SwiGLU (which involves 3 linear layers). We used hidden dimension $d_{\text{ff}} = 4d_{\text{model}}$.
4. Attention and residual dropout (see `cs336_scaling/model.py` for details).

We also used untied input and output embeddings. The models were trained on the SlimPajama dataset, and the model uses a byte-level BPE tokenizer with 32K vocabulary items (also trained on SlimPajama). The models' context length is 512. Attention and residual dropout were applied with probability 0.1. The models were trained with the AdamW optimizer with weight decay of 0.01 and gradient clipping of 1.0. We used a cosine learning rate schedule to decay the initial learning rate by 10×, with a cosine cycle length matching the total number of training steps. No learning rate warm-up was used.

References

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020. arXiv:2001.08361.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022. arXiv:2203.15556.

Greg Yang, Edward J. Hu, Igor Babuschkin, Szymon Sidor, Xiaodong Liu, David Farhi, Nick Ryder, Jakub Pachocki, Weizhu Chen, and Jianfeng Gao. Tensor Programs V: Tuning large neural networks via zero-shot hyperparameter transfer, 2022. arXiv:2203.03466.