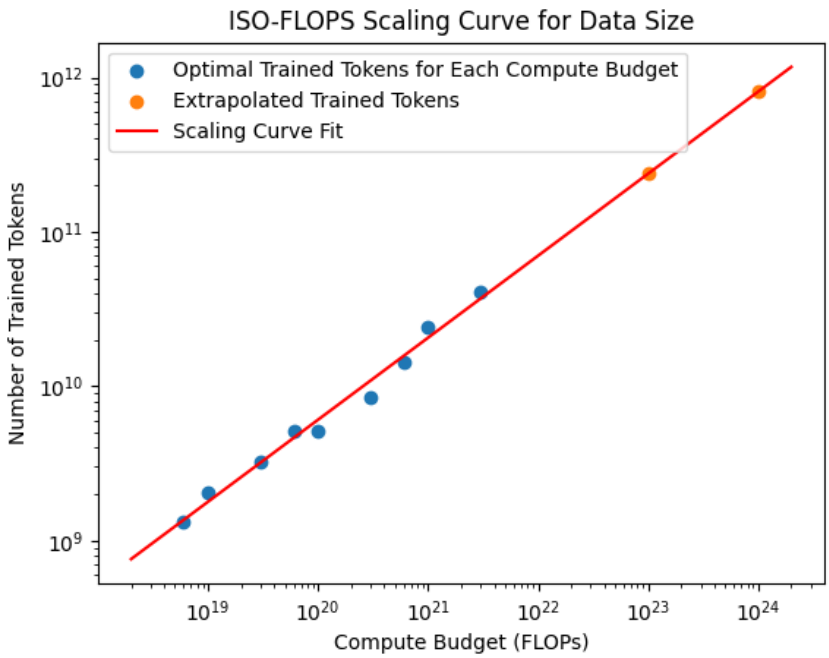
CS 336: Language Modelling from Scratch

Assignment 3

May 9, 2024

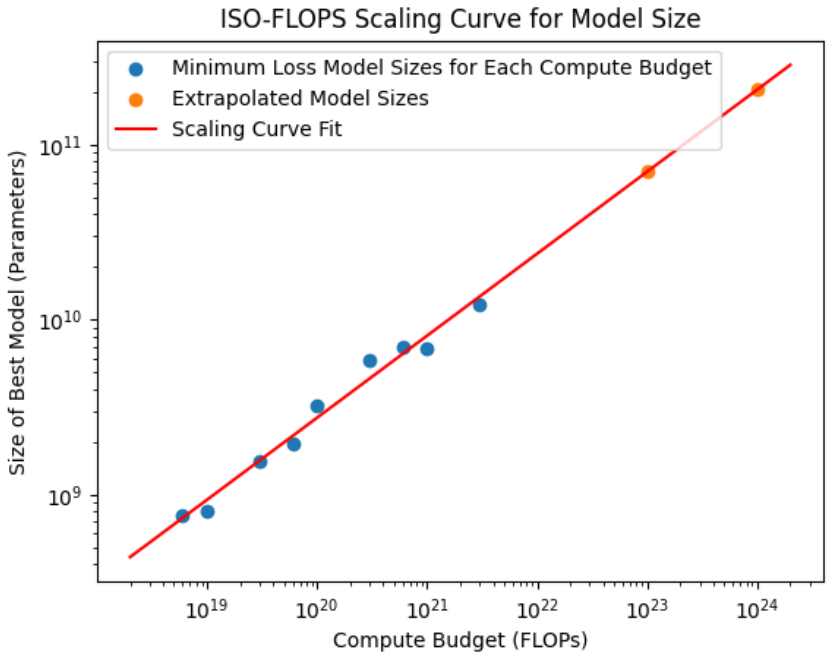
SUNet ID: mattreed
Name: Matt Reed

1 IsoFlops



$$t = 0.143 * f^{0.53} \tag{1}$$

The predicted optimal training tokens for 10^{23} FLOPS is $2.40 * 10^{11}$ tokens, and the predicted optimal training tokens for 10^{24} FLOPS is $8.08 * 10^{11}$.



$$p = 1.16 * f^{0.47} \tag{2}$$

The predicted optimal model size for 10^{23} FLOPS is $7.01 * 10^{10}$ tokens, and the predicted optimal training tokens for 10^{24} FLOPS is $2.06 * 10^{11}$.

2 Scaling Laws Approach

For this task, I was allotted $2 * 10^{18}$ FLOPS to determine the optimal model size and hyper-parameters for a run of 10^{19} FLOPS. In order to do this, I conducted a hyper-parameter sweep for a large amount of smaller models and extrapolated these findings for larger models. In parallel with Chinchilla's methods, I decided to run Isoflops sweeps for each possible compute level, find the optimal model and hyperparameter values for each, then finally extrapolate scaling laws from each of those optimal values to find the optimal values for the larger run.

Once the optimal values are found, extrapolating the scaling laws becomes trivial. A similar process was used above with synthetic data to determine the optimal model and dataset sizes for a large training run. We would just need to expand this process by extrapolating all model dimensions and hyperparameters instead of purely total model parameters. However, finding these optimal combinations is not easy. For each level of compute, there are roughly a million different unique combinations of inputs that can be tested:

$$(1024 - 64) * (24 - 2) * (16 - 2) * 2 * 2 = 1182720.$$

Obviously we do not have enough compute to run a million, or even close to a million, test runs for each compute level, especially for higher total FLOPS. For example, at the $1 * 10^{17}$

level, we could only run 20 trial runs before exhausting all compute resources for training. Therefore, I needed to implement more intelligent ways of exploring the hyperparameter space that could rely upon sparse sampling. These are the methods that I attempted:

Gradient Update Policy Optimization: I took inspiration from reinforcement learning paradigms and attempted to conduct gradient descent on the hyperparameter space using Monte Carlo gradient estimation. The procedure for each level of compute is as follows:

1. Initialize the hyperparameters randomly.
2. Calculate the loss at the current point in the hyperparameter space.
3. Calcualate the loss at a randomly chosen point close to the first point.
4. Use the change in loss to create a gradient with respect to each of the hyperparameters and increment or decrement them accordingly.
5. Repeat 2-5 at this new point until convergence.

This procedure kind of worked, but it had a few problems. Firstly, it assumes that the hyperparameter-loss space is convex which it probably is not (in general it probably would not be, but I also have no idea what kind of interpolation you guys did for the API). Secondly, even if stochastic gradient descent was noisy enough to approach a decent minimum, the estimation method is still a terrible approximation of the true gradient. This is because I had to rely on sparse sampling (due to computation limits) as well as fit the model within the discrete confines of the API input space. Gradient descent does not work well when there are for example only 2 choices for batch size and learning rate. The third problem is that the approach is a little suspicious to be using at all. The analogy of hyperparameters as parameters for a policy with an associated simulated utility function is loose, but it still worked well enough for me to get descent results for some of the smaller compute levels. However, the cost for each run was

$$2 * \text{flops} * \text{num\_steps}$$

which is far too steep for larger runs. For higher compute budgets, I had to invent more clever methods.

Coordinate Descent: My second approach was to instead of optimizing all the hyperparameters simultaneously, just to optimize one at a time until convergence. The procedure is as follows:

1. Initialize the hyperparameters randomly.
2. Loop through all parameters repeatedly, doing steps 3-4 for each, until convergence.
3. Fix all parameters in place except for the chosen parameter.

4. Conduct a binary search over the 1 dimensional hyperparameter space until an optimal value is found.

This method worked quite well, and even resulted in a few of the optimal runs I ended up using for my Chinchilla-like extrapolation done later on; however, it again had a few issues. One, it performs best when the loss landscape is smooth and unimodal, which ours isn't necessarily. If it was, then it would necessarily reach a global minimum, but since it probably isn't, it may get stuck in some local minimum. Furthermore, some parameters should be optimized jointly, like model size parameters. For example, if we already have the optimal model size determined, but not the correct width to depth determined, then coordinate descent won't change anything because the depth and width are already the best given the other parameters, but not the best they could be in theory. The third problem is again that this uses a lot of computation because the cost is:

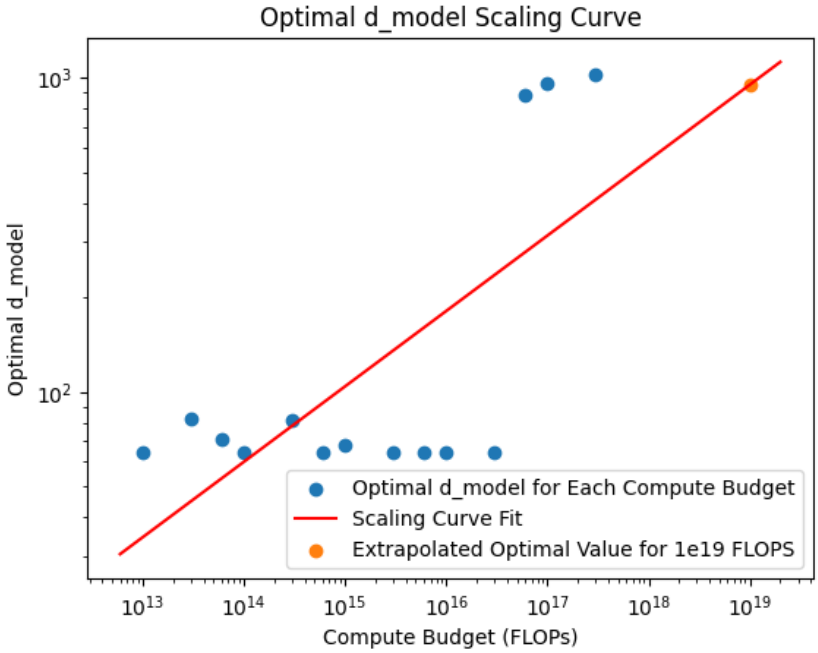
$$\text{num\_iter} * \text{flops} * \sum_{p \in \text{params}} \log_2 |\text{sample\_space}(p)|$$

Bayesian optimization using Gaussian Processes: Since evaluating each training loss is expensive, it is helpful to rely on a process that avoids approximating other metrics like gradients and instead attempts to directly learn the posterior distribution over hyperparameters that leads to the best training loss. In order to balance exploration and exploitation, points are first randomly sampled uniformly from the entire hyperparameter space, then promising areas are explored further until the best combination is found (within n function calls, so may not find the actual optimal value). This method assumes a multi-modal Gaussian distribution over the hyperparameter-loss space and improves the approximation with each added sample. This is still bottlenecked by compute since for longer training runs, we still only have a few samples to get it right. This is why it's important to have an educated guess for the prior before starting the process. For this, I used the scaling curves derived in the previous section to approximate the optimal number of parameters (which I now realize may be futile since that data was fabricated). I then used the conventional wisdom that the width to depth ratio should be roughly 125 and the width to number of heads ratio should be roughly 64. This I fed into the model to serve as the initial first guess. Finally, I decreased the number of samples for each compute level so that each level used approximately the same amount of the total compute budget (1.5%).

3 Scaling Curves

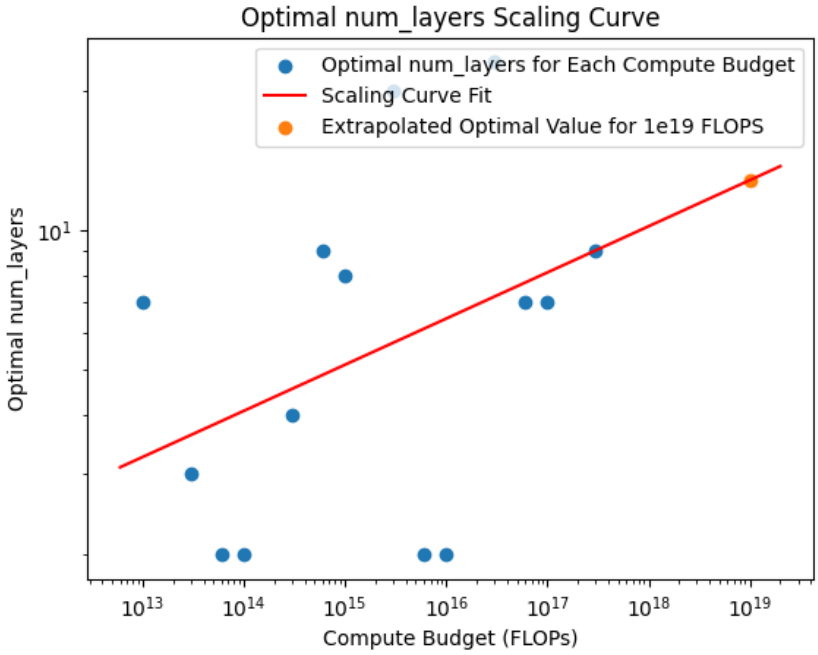
Once I found the approximate optimal hyperparameter settings for each compute budget, I fit a least squares regression line on the log-log plot of each parameter against total traing FLOPS. This led to some questionable fits either since the optimal model size and hyperparameters do not scale linearly in the log-log space or (more likely) my previously described

approaches did not find the optimal selections for each compute budget. The scaling laws for each hyperparameter are shown below along with the extrapolated value for the large training run of 10^{19} FLOPS.



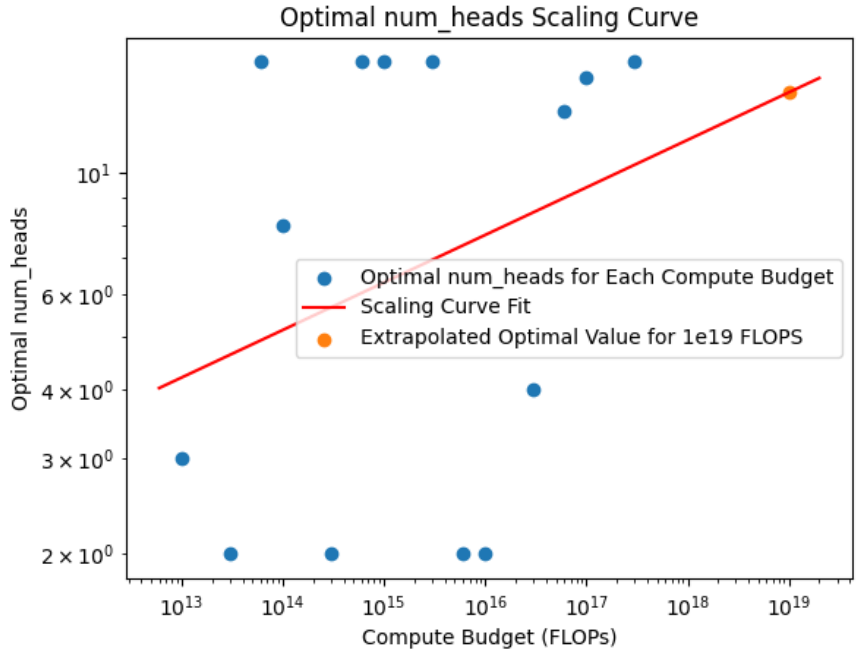
$$d = 0.0260 * f^{0.24} \tag{3}$$

Extrapolated best model dimension: 951.504.



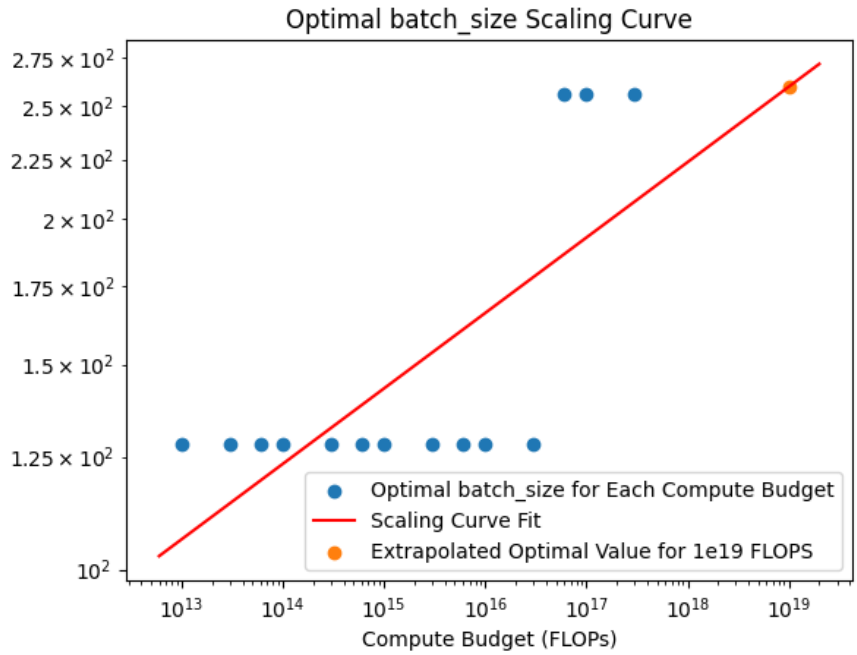
$$l = 0.166 * f^{0.10} \tag{4}$$

Extrapolated best number layers: 12.814.



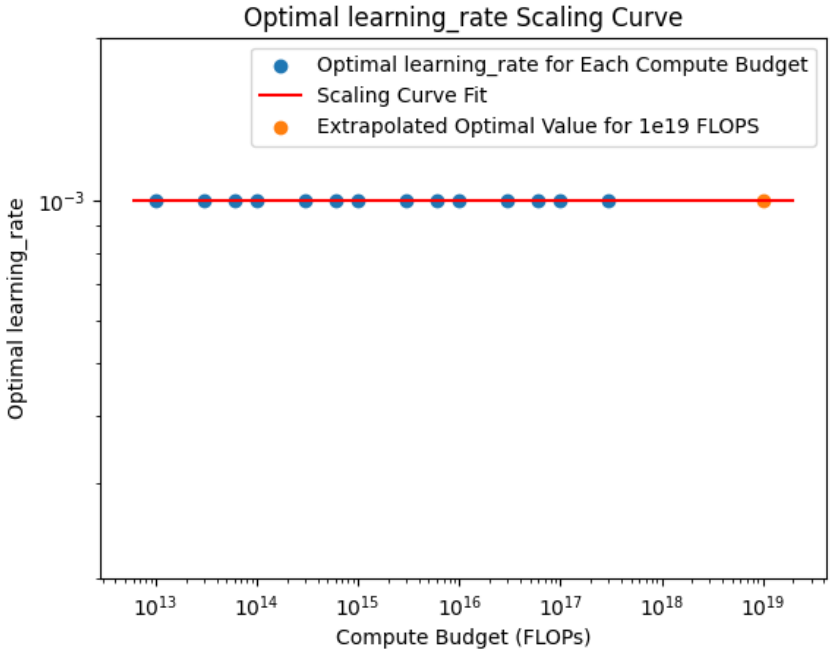
$$h = 0.310 * f^{0.09} \tag{5}$$

Extrapolated best number heads: 14.069.



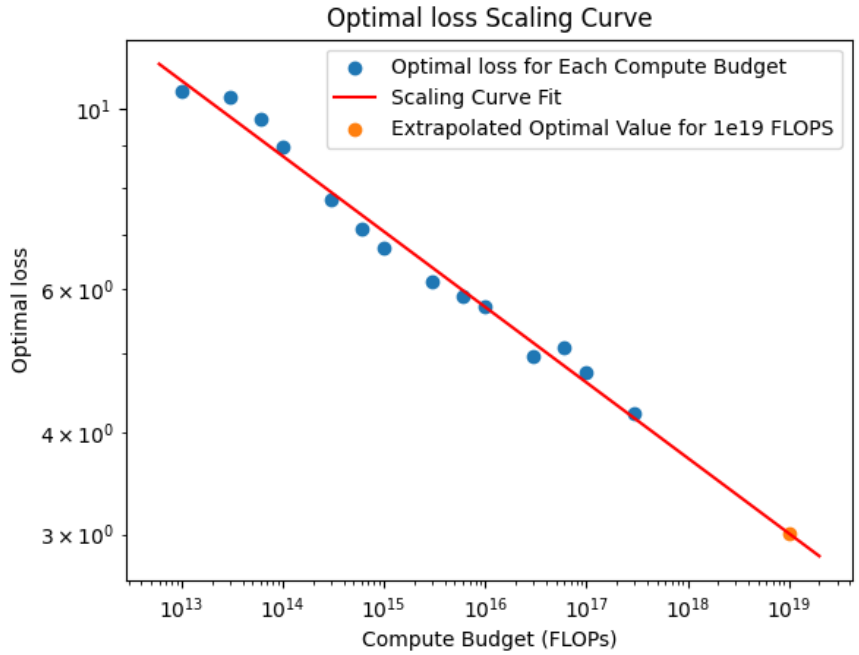
$$b = 15.3 * f^{0.06} \tag{6}$$

Extrapolated best batch size: 259.868



$$b = 0.001 \tag{7}$$

Extrapolated best learning rate: 0.001.



$$b = 173 * f^{-0.09} \tag{8}$$

Extrapolated Loss for a model with the given hyperparameters: 3.008.

4 Final Notes

These are obviously not very impressive results, so I'm not getting published any time soon. My final predictions for the optimal 10^{19} FLOPS model (after rounding to the nearest API input value) were 952 for the model dimension, 14 for the number of heads (which is conveniently an even divider for the d_model), 13 for the number of layers, 256 for batch size and 0.001 for the learning rate. This leads to a total number of non-embedding parameters of 141,383,424.