Understanding Large Language Models

An Engineer's Handbook

Irhum Shafkat

2024-03-30

Table of contents

Executive Summary

Starting in late 2022, large language models (LLMs) began receiving immense public interest, as the technology had finally matured to the point where it could be used to create consumer products such as ChatGPT (Vincent 2022). This book captures many recent lines of thought on how LLMs are expected to keep evolving: its five chapters cover different stages of the creation process of an LLM, as follows:

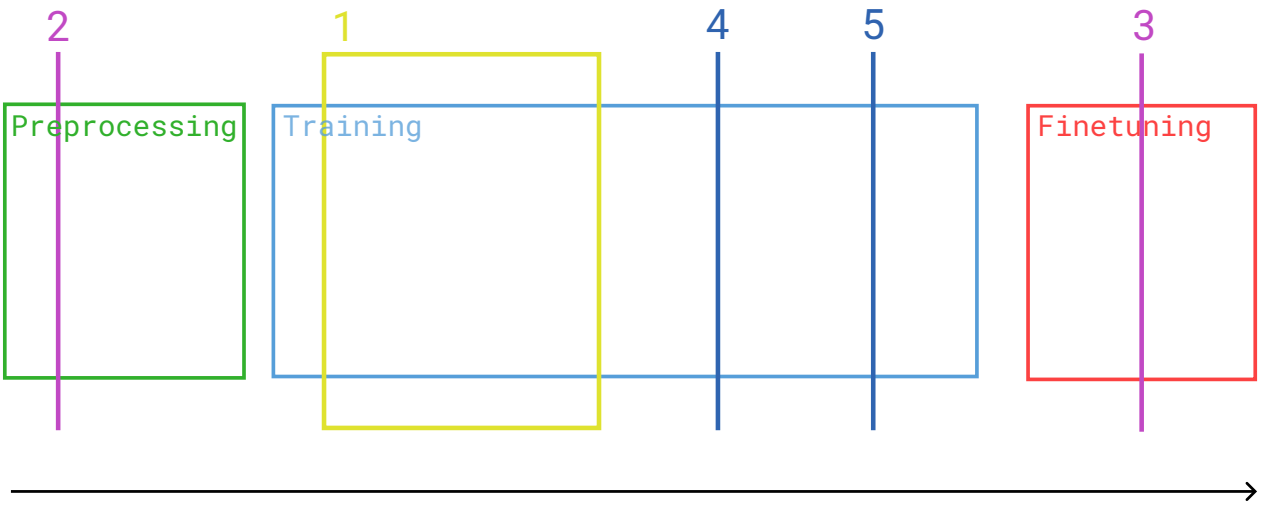


Figure 1: Chapters 1 (on Computation), 4 and 5 (on Scaling) can be placed inside the Training stage of an LLM, whereas Chapters 2 and 3 (both on Optimization) live on opposing ends of the pipeline. Chapter 1 is considerably more broader in scope than the other four, as reflected in the diagram.

The five chapters are better thought of organized into three different thematic areas (as opposed to their exact chronological stage): computation, optimization and scaling. Specifically:

1. Computation: This is the focus of Chapter 1 (Neural Language Models). This theme explores what a language model is, and how the next-token prediction objective induces a large range of behaviors which are of practical use. A particular focus here is the connection between compression and "intelligence": even an n-gram model can be used for language modeling, but what makes neural network models special is their ability to predict probabilities for inputs that are similar, but not the same as in the training data.
2. Optimization: This is the focus of Chapters 2 (The Quirks of Tokenization) and 3 (LoRA and Weight Decay). Although the two topics sit on opposing ends of the LLM creation pipeline, they provide two different lenses on how an LLM can be improved: either generally, by using

better representations of the input in all cases (Chapter 2), or by adjusting the model for a specific use case (Chapter 3).

3. Scaling: Much of the recent progress in LLMs have been driven by scaling, which entails keeping the same base architecture for a model (most commonly, Transformers (Vaswani et al. 2017)), but increasing model size and amount of data jointly to produce more performant models. Chapter 4 (Tensor Parallelism with `jax.pjit`) looks at the engineering challenges that arise when a model is so big it must be "split up" across a GPU/TPU supercomputer. Chapter 5 (Chinchilla's Implications) looks at the current science of how to scale models, where extensive empirical research by Hoffmann et al. (2022) now suggests a "1:1 scaling" between model size and data.

Overall, by exposing so much of the inner workings of LLMs across the entire pipeline, I hope the reader will find themselves more capable of asking thoughtful questions about LLMs, and of finding new ways to improve them further.

1 Neural Language Models

At their core, language models (LMs) are simply functions that assign a probability to a snippet of text. A common variant are autoregressive LMs: functions which produce a probability for the "next token" conditioned on some already written text, that is $p(\text{token} \mid \text{context})$. Such a models can be immensely powerful, as many tasks can be re-written to fit into a "next token prediction" problem, for instance:

- **Factual Retrieval**: If we want the answer to `"Where's the Eiffel Tower located?"`, then $p(\text{"Paris"} \mid \text{context})$ should be the largest value of all possible $p(\cdot \mid \text{context})$. If we then select `Paris` (as it has the highest conditional probability), we have the correct answer.
- **Calculator**: If the context snippet is `"What's 16 times 4?"`, we should expect the highest probability token for $p(\cdot \mid \text{context})$ to be `64`. This allows the model to act as a calculator!
- **Translation**: If the context snippet is `"English: Hello! Spanish:"`, we should expect "¡Hola!" to be the highest probability outcome for $p(\cdot \mid \text{context})$. Selecting `¡Hola!`, we repurpose the language model into a translation engine.
- **Simulation**: If we have the snippet `"The outcome of this six-sided die roll is"`, we should expect $p(4 \mid \text{context}) = \frac{1}{6}$. This allows the simulation of probabilistic events.

If we had an *infinite* amount of text data, observing *every possible sentence* proportional to their "true probability", estimating these probabilities is straightforward.

- For instance, if we have a corpus with sentences of the form `"The outcome of this six-sided die roll is [x]"`, and $\frac{1}{6}$th of them have `x=4`, we could estimate $p_\theta(4 \mid \text{context}) = \frac{1}{6}$.
- Likewise, if in our infinite corpus we have `"What's 16 times 4? [x]"`, and it is always `"What's 16 times 4? 64"`, we could estimate $p_\theta(64 \mid \text{context}) = 1$

Unfortunately, we do not have an infinite corpus. Even with a massive text corpus, we likely won't observe most valid sequences of say, 100 words, even once since there's an exponentially large number of sequences.

This chapter focuses on how neural networks allow us to build language models that can produce "good" next token probabilities for sequences that are "similar" (but not the exact same) as sequences in the training set, as first introduced in Bengio et al. (2003). Reusing language from human learning, this is akin to *near* transfer (Schunk 2011, 320), where there is substantial overlap between the "training" and "test" contexts.

1.1 Levels of Analysis

Within any sufficiently complicated technical system, there are multiple levels that interact with each other, but are also distinct *from* each other. Disambiguating between them can provide a clean mental model to reason about them. I find it helpful to apply Marr's levels of analysis (Marr and Poggio 1976) in such settings, and applying them here[1]:

- **Computation:** This is the abstract computation *any* language model does, regardless whether it's using a lookup table, a transformer or a bio-inspired computer. Specifically for an autoregressive language model, it is the estimation of the conditional probabilities for the next token conditioned on the current tokens.
- **Representation:** There's multiple distinct ways to *parametrize* a language model. In this post we'll explore and compare two: explicitly parametrizing each conditional probability, and using a neural network. We also need to decide how to *represent* raw text as individual tokens (is a single word a token? a single character?) and we'll explore this too.
- **Implementation:** Once we've decided on a representation for our functions and data, we actually need to implement them in code to get them running on hardware. This is the Python/JAX code we use in this article, and in turn all the kernels they execute to carry out the computation.

This chapter specifically focuses between the computational and representational levels, where we explore how a language model parametrized by a neural network stores next token probabilities *implicitly* in its weights[2].

The scenario of language modeling is a special case of operating over the joint probability distribution of m discrete random variables. Specifically:

- **Support**: We have a sequence of random variables $(W_1, W_2, ..., W_m)$, where each variable represents a "word", more commonly called a token. Each token can be one of V possible values, where V is the size of the "vocabulary".
- **Parametrization**: There is a *true* probability distribution the samples in our datasets are generated from. For example, the sequence ('I', 'like', 'cats') is sampled with probability p('I', 'like', 'cats'). However, we do not know p, so we use an approximation p_θ.
- **Sampling**: We assume any text datasets we have to be samples from this true distribution.
- **Estimation**: To *estimate* the value of the parameter θ, we find values that *maximize* the likelihood of our dataset.

In brief, a language model is simply a function that assigns probabilities to length-m sequences of text. To reinforce this, let's look at an example based on an actual dataset.

[1]This breakdown is subjective, and strongly dependent on the problem you're looking at. If we were optimizing raw PTX instructions instead: the Python/JAX code may define the *computation*, the PTX being the *representation* and the GPU architecture-specific hardware operations the *implementation*. One person's implementation level can easily be someone else's computation level.

[2]This chapter heavily uses the knowledge of joint probabilities over several discrete distributions. A primer can be found in Appendix A

1.2 Dataset

We use the dataset from Karpathy (2015), which is a 4.6MB file containing all the works of Shakespeare. Here's the first 60 characters in the dataset:

```
import requests

url = 'https://cs.stanford.edu/people/karpathy/char-rnn/shakespeare_input.txt'
text = requests.get(url).text
text[:60]
```

```
'First Citizen:\nBefore we proceed any further, hear me speak.'
```

1.2.1 Tokenization

Before building a language model, we must choose how to represent our raw text as individual "tokens" in a "sequence". This requires concretely defining what the vocabulary for each token is:

- **Word-level**: If each entire English word is a token, then the size of the vocabulary V grows quickly. This dataset alone has over 60,000 unique words.
- **Character-level**: If each individual *character* is a token, V is small (here, there's only 67 unique characters, including whitespaces like `\n`.). However, for a given amount of information, m needs to be much larger since there's more tokens. For instance, "the weather in Berlin" is 4 words, but 21 characters.
- **Subword-level**: The most common way to build language models today is to use *subword* tokenization, such as using the SentencePiece tokenizer (Kudo and Richardson 2018). These keep small words intact, but break up longer words into smaller "subwords".

For this article we'll be tokenizing at the character level: it's straightforward, and will allow us to directly visualize the probability matrices. Breaking up the first 15 characters into individual tokens, we have:

```
print([c for c in text[:15]])
```

```
['F', 'i', 'r', 's', 't', ' ', 'C', 'i', 't', 'i', 'z', 'e', 'n', ':', '\n']
```

Then, converting each unique token into a unique integer, we have:

```
vocab = sorted(set(text))
char2idx = {c: i for (i, c) in enumerate(vocab)}
tokens = [char2idx[c] for c in text]

print(tokens[:15])
```

```
[18, 49, 58, 59, 60, 1, 15, 49, 60, 49, 66, 45, 54, 10, 0]
```

Note that our input pipeline looks like this:

$$\text{raw text} \rightarrow \text{tokens} \rightarrow \text{integers}$$

Now suppose we've set out to build a language model with context length $m = 1024$; that is, a model can assign a probability to all character sequences of length 1024. Even here, we run into combinatorial explosion: with $V = 67$ characters and context length $m = 1024$, there's 67^{1024} possible sequences.

Most of those sequences would have zero probability (on a smaller scale, "big city"[3] is a valid sequence of length 8, while "xcsazmad" is not), but even among the valid sequences, you'd need a truly staggering number of samples to even observe every valid sequence. Since this is infeasible, let's try simplifying.

1.3 Chain Rule

Using the chain rule (as shown in Section A.4.1), we can represent any joint probability as a product of *conditional* probabilities. With $m = 1024$, we have:

$$\begin{aligned} p(w_1, w_2, ..., w_{1024}) &= p(w_{1024} \mid w_{1:1023}) p(w_{1023} \mid w_{1:1022}) ... p(w_2 \mid w_1) p(w_1) \\ &= \prod_{i=1}^{1024} p(w_i \mid w_{1:i-1}) \end{aligned}$$

In this case, it means iteratively computing the probability of each token conditioned on *all* the previous tokens. On its own, this is of little help. As before, although we're "splitting up" the joint probability tensor, there's still $V^{1024} - 1$ possible variables *in total*, since each "next token prediction" would need a unique conditional probability vector for each combination of previous words $w_{1:i-1}$.

1.4 Unigram

What if we pretended, just like coin tosses, each token was independent of its previous tokens? This is *not* true: We likely have $p(\text{'h'} \mid \text{'t'})$ greater than $p(\text{'h'})$, as "th" appears in "the", one of the most common words in the English language. Knowing prior tokens definitely changes the probability of the next token.

But continuing forward with this naive assumption, we write:

[3] is a character too, so there's 8 in total.

$$p(w_1, w_2, ..., w_{1024}) = p(w_{1024} \mid w_{1:1023})p(w_{1023} \mid w_{1:1022})...p(w_2 \mid w_1)p(w_1)$$
$$\approx p(w_{1024})p(w_{1023})...p(w_2)p(w_1)$$

That is, the joint distribution is the product of the marginal distributions. Since each marginal $p(\cdot)$ has $V = 67$ outcomes, they have 66 parameters. We make *another* naive assumption: each of the X_i's are identically distributed[4]. This means these 66 parameters are *shared* across all 1024 terms in the product above.

Note that with these two assumptions, we're no longer estimating the probability of length $m = 1024$ sequences, but rather $m = 1$. We're then assuming we can approximate the probability of a longer sequence with the products of these individual probabilities.

1.4.1 Estimation

With the two naive assumptions above (independence + sharing across timesteps) we have 66 parameters that need to be estimated. We know the maximum likelihood estimate here from the previous section: count the proportion of times an outcome happened, among the total.

Let's first split the corpus (of 4,573,338 tokens) into a training set (first 4 million) and a validation set (remaining 573,338).

```
train_tokens, valid_tokens = tokens[:4000000], tokens[4000000:]
```

Then, to estimate the parameters, we count the number of times the token appears and divide by the total[5]:

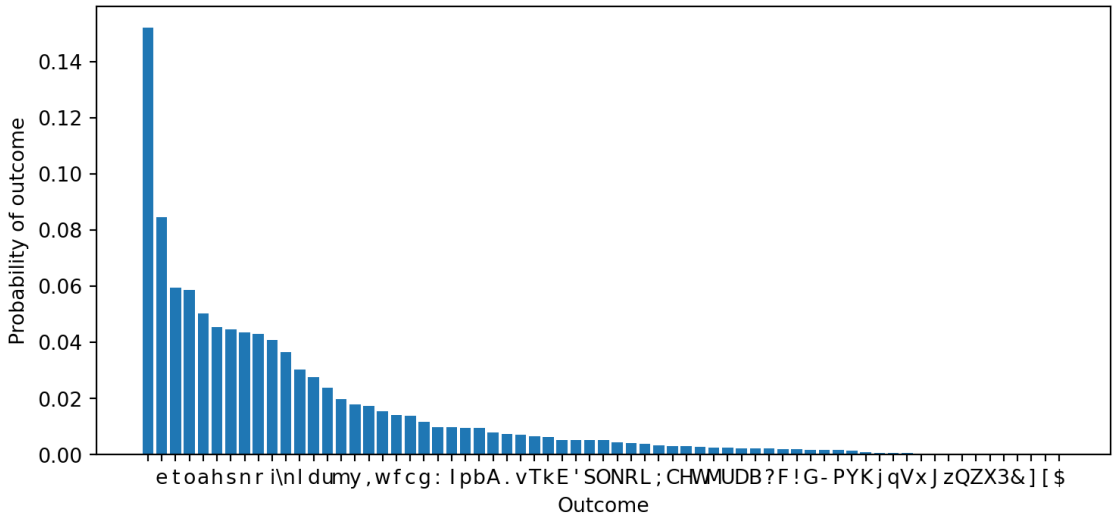
```
import numpy as np

# Create "initial" counts
counts = np.zeros(shape=(len(vocab),)) + 0.1

# Loop over tokens in dataset
for token in train_tokens:
    counts[token] += 1

# Normalize to 1
params = counts / counts.sum()
```

We can then visualize this one-variable probability distribution, in the following bar graph:

[4]This carries on the coin logic: the coin doesn't care if it's the 1st or the 100th toss, the probability for the outcome heads remains the same.

Likewise, X_{562} having the same distribution as X_1 means the probability for an outcome (e.g. $p('t')$) is the same at both timesteps.

[5]Note that the Chinchilla paper uses a more detailed approach to calculating C than Kaplan et al. (2020)'s $C \approx 6ND$ approximation, explained in Appendix F. The $C \approx 6ND$ approximation is within 10% across two orders of magnitude (Table A4), so it's still a good mental model!



As we see, the most "likely" token is `' '`, followed by `e`, with the lowest probability one being `$`.

1.4.2 Evaluation

How good of a model of language is this? Even though *very* constrained, we can see it's correctly "learned" a qualitative aspect of English: that `e` is the most commonly occurring letter. But how do we know these frequencies are reliable, and not due to idiosyncrasies in the first 4 million tokens[6]?

One metric commonly used for language models is perplexity, which works as follows:

- For each token, compute the log probability of that token under the model.
- Take the mean of the log probability across all tokens.
- Take the negative exponential of this mean.

In code, we have[7]:

```
def perplexity_unigram(probs_vec, tokens, start=7):
    log_probs = [np.log(probs_vec[token]) for token in tokens[start:]]
    return np.exp(-np.mean(log_probs))
```

Note that perplexity maxes out at V (here, 67) when every token has probability $\frac{1}{V}$ (that is, uniform) under the model. It has a minimum at 1, when the model assigns a probability of 1 to *every* token;

[6]For instance, if `?` appears 10x more in the training set compared to the overall corpus, the maximum likelihood estimate $p_\theta('?')$ will be 10x larger than $p('?')$.

[7]Note that we begin computing perplexity starting at the 8th token, for both the training and validation sequences. Later in the article we'll build a model estimating probabilities conditioned on the 7 previous tokens; adjusting now means the perplexity comparisons across all models remain comparable.

that is, it *perfectly* predicts the sequence[8].

Computing the perplexity over the training and test sets, we have `train_ppl=27.53`, and `valid_ppl=27.14`. One way to interpret this value is that the model would be "choosing" between `27.14` outcomes (out of 67) at every step if asked to reproduce the validation sequence (lower is better).

1.4.3 Generation

Now that we've estimated the parameters, we can generate a new sequence. Let's generate a new sequence of length 30:

```
sequence = ''

for _ in range(30):
    sequence += np.random.choice(vocab, p=params)

sequence
```

```
'okst\n .uar hf  ett   se onhe P'
```

As we can see, this isn't very English like: after all, it's treating every token as a 67-way coin toss, with no regard for the previous tokens. Let's make this more realistic.

1.5 Bigrams

Instead of assuming each token is independent, let's assume that a token w_i and tokens $w_{i-2}, w_{i-3}...$ are conditionally independent, given token w_{i-1}. This means if we know the immediately previous token, knowing tokens more previous will *not* change the probability. We have the approximation:

$$\begin{aligned} p(w_1, w_2, ..., w_{1024}) &= p(w_{1024} \mid w_{1:1023}) p(w_{1023} \mid w_{1:1022}) ... p(w_2 \mid w_1) p(w_1) \\ &\approx p(w_{1024} \mid w_{1023}) p(w_{1023} \mid w_{1022}) ... p(w_2 \mid w_1) p(w_1) \end{aligned}$$

Again, this is a faulty approximation: $p(\text{'e'} \mid \text{'h'}, \text{'t'})$ is greater than $p(\text{'e'} \mid \text{'h'})$, as knowing the first two letters `th` would give us much stronger confidence in the completion `the` than just the previous letter `h`. But it is better than assuming complete independence between tokens.

[8] 1 is a computational minimum. In practice, language *itself* has a minimum perplexity larger than 1, and even the best model we'll ever build won't go lower.

Analogously, even with a perfect estimate of $\theta = 0.6$ for the biased coin earlier, we *cannot* perfectly predict the sequence of heads and tails in a series of tosses; there is randomness *inherent* to the process itself.

For each conditional probability term, there's $V(V-1)$ parameters as we need one vector for each possible "prior" token. If we share these parameters across timesteps as before[9], then there's $V(V-1)$ parameters in *total* to be estimated to be able to compute the conditional probabilities. With $V = 67$, this is 4422 free parameters.

1.5.1 Estimation

Estimation with maximum likelihood remains similar: To estimate $p_\theta(b \mid a) = \theta_{a,b}$, we find *all* the cases where a happens, and then compute the proportion of them that are followed by b. In practice, we create a `counts` matrix, and normalize it such that each vector sums to 1[10]:

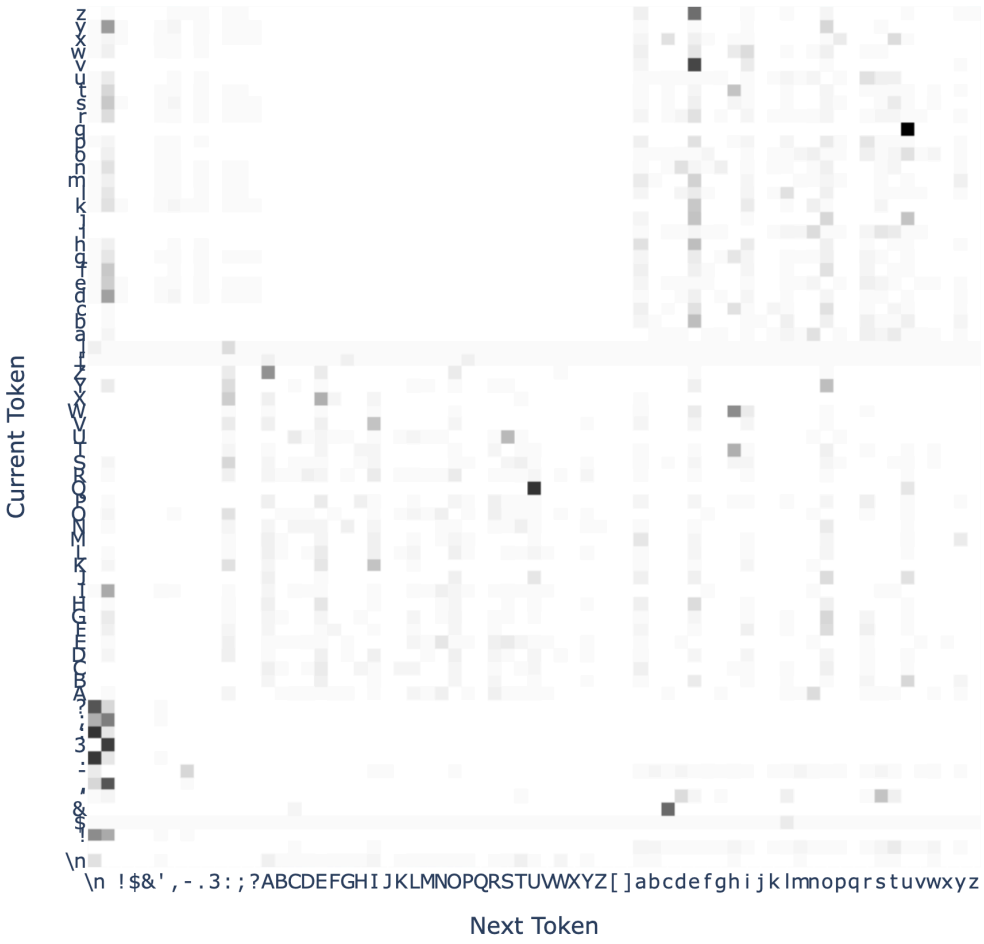
```
# Create "initial" counts
counts = np.zeros((len(vocab), len(vocab))) + 0.1

# Compute the counts matrix
for i in range(1, len(train_tokens)):
    prev_token = train_tokens[i-1]
    current_token = train_tokens[i]
    counts[prev_token, current_token] += 1

# Normalize to get proportions that sum to 1.
params = counts / counts.sum(axis=-1, keepdims=True)
```

When visualized, the matrix of estimated conditional probabilities is as follows. Note that each row sums to 1.

[9] That is, if $x_a = x_b$, and $x_{a-1} = x_{b-1}$, then $p(x_a \mid x_{a-1}) = p(x_b \mid x_{b-1})$.

This is a sensible assumption, as $p(\text{'e'} \mid \text{'h'})$ should be the same regardless if `e` is the 2nd or 285th token: all the information needed to predict it is (assumedly) in the prior token `h`.

[10] Note that we add a 0.1 count to each "transition" pair. Without it, if a pair (a, b) doesn't appear in the training sequence it would have $p_\theta(b \mid a) = 0$. If it subsequently appeared in the validation sequence, it would also have $p_\theta(b \mid a) = 0 \Rightarrow -\ln p_\theta(b \mid a) = \infty$ (and in turn, a sequence perplexity of ∞). Adding it ensures a small probability is assigned to every possible transition.



We can glean patterns from this matrix: for instance, the estimated probability of a newline following a newline $p_\theta(\text{'\textbackslash n'} \mid \text{'\textbackslash n'})$ is `0.187`.

1.5.2 Evaluation

Evaluating, we have `train_ppl=11.87`, and `valid_ppl=11.95`, which more than halves the perplexity values of the unigram model. This makes sense: our estimates for the next token *should* be better when we account for the previous token.

1.5.3 Generation

Sampling is also straightforward: given a starting token w_{i-1}, we sample the next token using the conditional probability vector corresponding to $p(\cdot \mid W_{i-1} = w_{i-1})$, as follows:

```
sequence = 'e'

for i in range(1, 30):
    prior_token = char2idx[sequence[i-1]]
    conditional_prob = params[prior_token, :]
    sequence += np.random.choice(vocab, p=conditional_prob)

sequence
```

```
'e:\nTERI:\nsidrexthes is ache gr'
```

The text now feels a bit more plausible, but not by much: it's challenging to form full words when you're constrained to ignore all tokens other than the one just before.

1.6 Bigrams: Neural Networks

The **computation** we derived above was to represent the joint probability $p(w_1, ..., w_{1024})$ as the product of conditional probabilities $p(w_i \mid w_{i-1})$. The **representation** we chose was to have each $p(b \mid a) = \theta_{a,b}$[11]. This resulted in a 67×67 matrix[12].

But we only *need* an individual parameter for each conditional probability in p_θ, if we want to be able to update a probability without altering the others. If we're okay with a constrained parametrization, we can use just about anything: a Fourier series, a sequence of piecewise linear functions, etc. Since neural networks are universal function approximators[13] (Hornik 1991), we could represent p_θ using a neural network; that is $p_\theta(b \mid a) = f_\theta(a)[b]$. In detail:

- **Input**: The neural network takes in the token a as input, and returns a vector of probabilities corresponding to $p_\theta(\cdot \mid a)$.
- **Output**: We then index into this vector with b, getting the probability $p_\theta(b \mid a)$.

There's a problem here: while the output (a probability) is continuous, the input a (a token) is a discrete integer. We can't use backpropagation here, since we can't differentiate through a. Or can we?

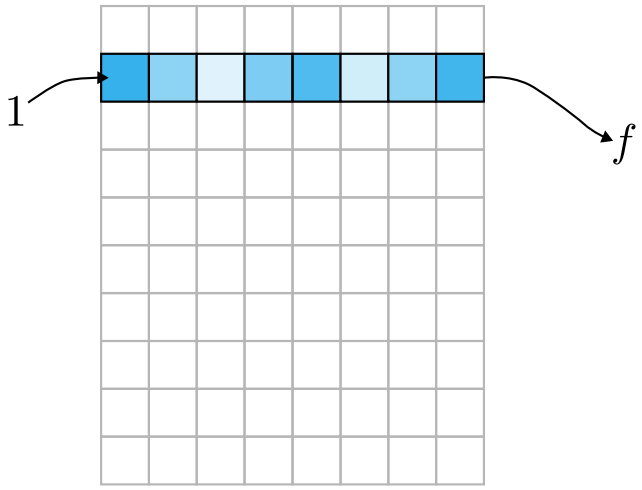
[11] Subject to $\sum_b \theta_{a,b} = 1$

[12] We could have *technically* used a 67×66 matrix. For each of the 67 conditional probability vectors, once we know the first 66 probabilities, we can compute the 67th by subtracting the sum from 1. Here it's just easier to have the extra column.

[13] In the case of infinite width; but even in finite cases, given sufficient width and depth they're effective.

1.6.0.1 Embeddings

One approach is to use embedding vectors: we associate a real valued vector with each token in the vocabulary, and use *that* as an input to the neural network:



This converts the continuous nature of neural networks from a *problem* to a *feature*. Recall that with continuous functions, small changes in the input result in small changes in output.

If the next token probabilities (the outputs) for any two tokens p and q are similar (that is, $p(\cdot \mid p) \approx p(\cdot \mid q)$, then during training, the embedding vectors for p and q (the inputs) can be optimized to be close to each other (again, similar inputs $\rightarrow$ similar outputs). Instead of a human defining a hard constraint (such as $\theta_{p,\cdot} = \theta_{q,\cdot}$ if we were using explicit conditional probability vectors), the network can *learn* these associations directly from data.

Note that our input pipeline now looks like this: in doing so, we convert a raw string into a sequence of vectors.

$$\text{raw text} \rightarrow \text{tokens} \rightarrow \text{integers} \rightarrow \text{embedding vectors}$$

1.6.0.2 Training

Training (or learning) really is just a concise way of saying "solving an optimization problem". Specifically, the following problem:

$$\min_{\theta}[-\ln L(\theta)]$$

That is, we wish to:

- find the values of the parameters θ of the neural network

- that minimize the negative log-likelihood of our observed training sequence[14]
- or equivalently, *maximize* the probability of our observed training sequence

Unlike previously, where we "knew" the closed-form, optimal solution (calculate the proportion of the total), this is a considerably more complex parametrization with no closed form solution[15]. After all, instead of having each $p_\theta(b \mid a)$ represented by a separate parameter $\theta_{a,b}$, we now have *every* parameter being used to compute *every* conditional probability.

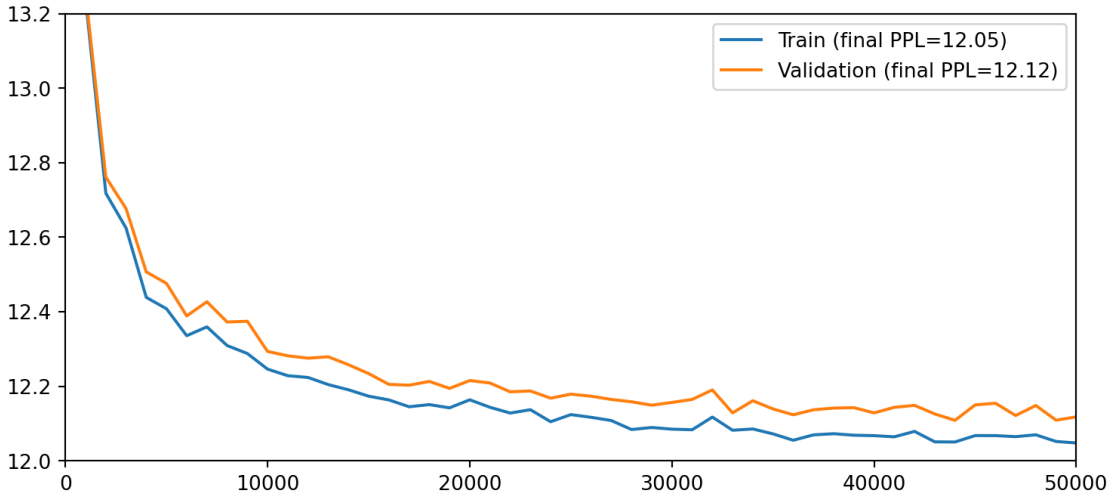
We instead use gradient descent to iteratively estimate values of θ with lower negative log-likelihood. We set up a simple neural network with 2 hidden layers as follows:

```
class LanguageModel(nn.Module):
    num_embeddings: int = 67
    features: int = 16

    @nn.compact
    def __call__(self, x):
        embed = nn.Embed(self.num_embeddings, self.features)
        # Get the embedding vectors for each input token
        x = embed(x)
        batch_dim, hist_size, features = x.shape
        x = x.reshape(batch_dim, hist_size * features)
        # Apply two hidden layers
        x = nn.Dense(self.features * 4)(x)
        x = nn.gelu(x)
        x = nn.Dense(self.features * 4)(x)
        x = nn.gelu(x)
        # Get logits for next token prediction
        x = nn.Dense(self.features)(x)
        x = embed.attend(x)

        return x
```

We optimize for 50,000 steps, and compute the perplexity (over the entire train and validation sequences) every 1000 steps. The training notebook is here:

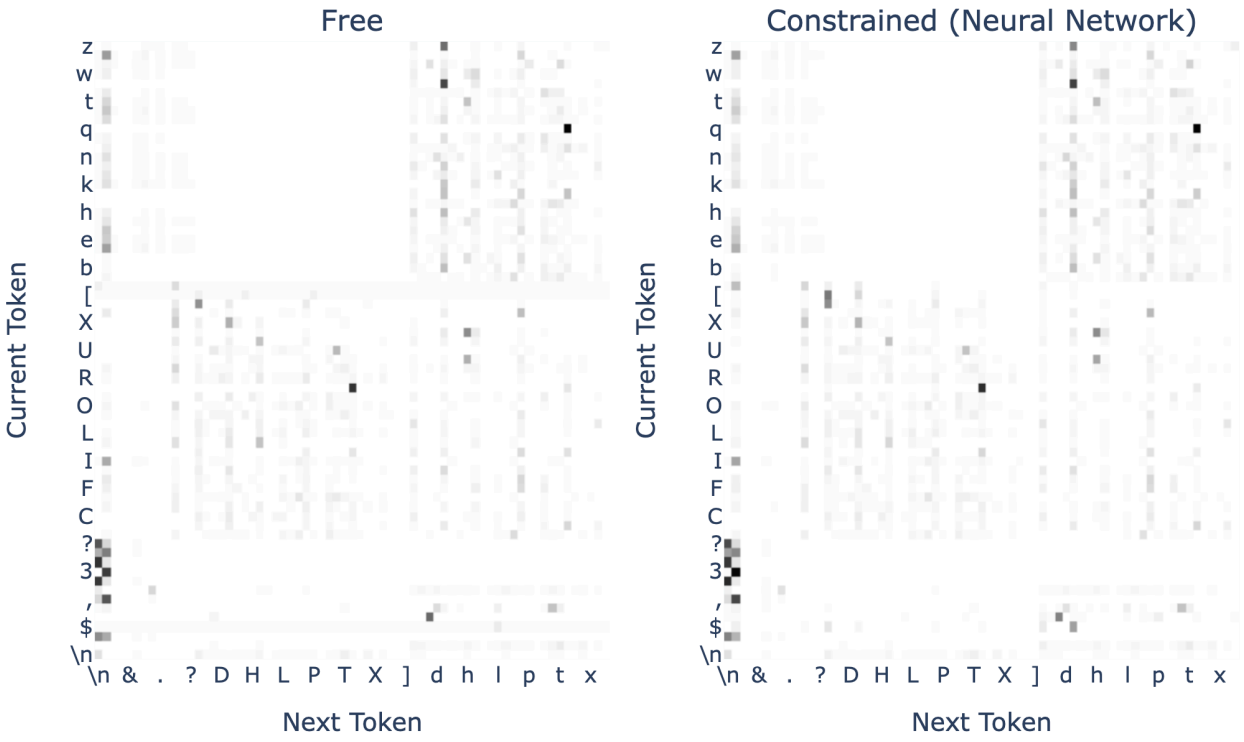
[14] Note that in practice, we're solving for $\min_\theta[-\ln L(\theta) + R(\theta)]$, where $R(\theta)$ is some regularization term (such as the squared sum of the weights $\lambda \sum_i \theta_i^2$).

[15] And many local optima, not just one global optima.



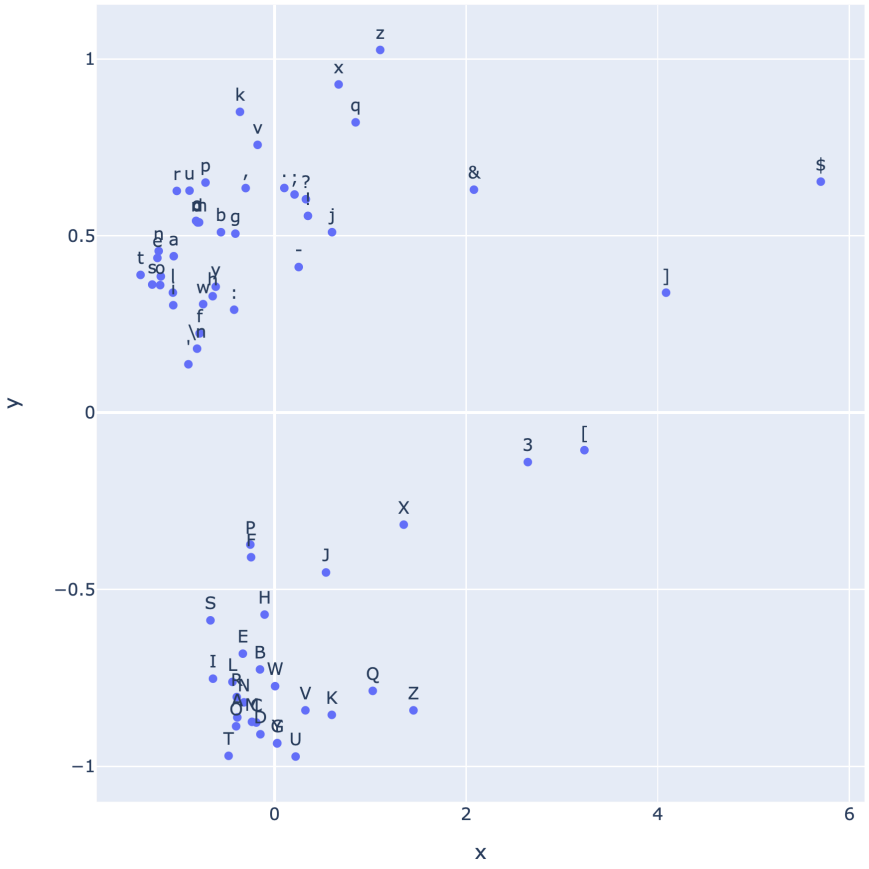
1.6.0.3 Analysis

First, let's look at the outputs. Recall that the network parametrizes $p_\theta(b \mid a) = f_\theta(a)[b]$. We can compute all the conditional probability vectors $p_\theta(\cdot \mid a) = f_\theta(a)$ by passing in all 67 unique values of a. Doing so, and concatenating these vectors into a matrix, we have a matrix quite similar to that from the previous section:



The similarity arises because, despite the different parametrizations, both of these models have the same objective: produce values of $p_\theta(b \mid a)$ for pairs of tokens a, b, that minimizes the negative log-likelihood of the training sequence. If for 100 appearances of `e`, 10 are followed by `o`, then the maximum likelihood estimate is $p_\theta(\text{'o'} \mid \text{'e'}) = \frac{10}{100}$, regardless if its parametrized with individual parameters or a neural network.

Let's also look at the embedding vectors associated with each token, after training is completed. Since these are 16-dimensional vectors, we project them down to 2 using PCA:



There's substantial structure here: the embeddings for the capital letters and lowercase letters form distinct groupings. Conceptually, this makes sense: the next token probabilities (the output) for all capital letter tokens (the input) are likely to have higher probability placed on lowercase letters (than just a lowercase → lowercase transition). Since their outputs share that similarity, their input embeddings should be similar (but *not* same) too.

1.6.1 n-grams

The neural network in the previous section had 7,360 free parameters; whereas the matrix of conditional probabilities only had 4,422. The neural net appears to consume more memory *and* has a

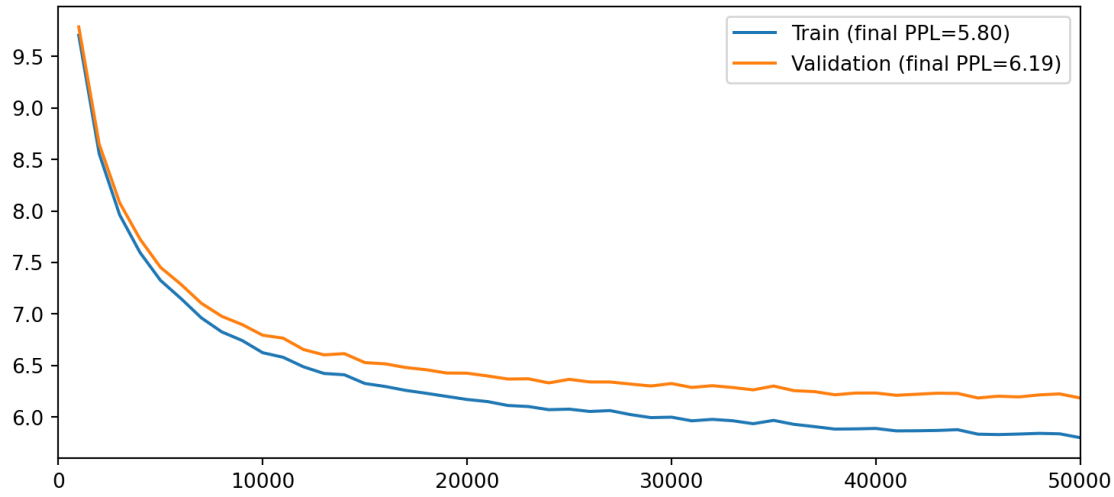
higher validation perplexity than the explicit parametrization. But now, instead of conditioning on the previous token, let's condition on the previous *seven*:

$$\begin{aligned} p(x_1, x_2, ..., x_{1024}) &= p(x_{1024} \mid x_{1023}, ..., x_1)p(x_{1023} \mid x_{1022}, ..., x_1)... \\ &\approx p(x_{1024} \mid x_{1023}, ..., x_{1017})p(x_{1023} \mid x_{1022}, ..., x_{1016})... \end{aligned}$$

We continue the "shared across timesteps" assumption; that is, the next token probability only depends on the previous 7, and not what numbered token it is in the sequence. Even then, conditioning on 7 tokens we'd have $67^7(67-1) \approx 10^{14}$ parameters. Since *most* combinations of the previous 7 tokens are invalid, we could use a sparse representation, but we'd still need a *huge* number of samples to accurately estimate next token probabilities.

But we could also just parametrize this with a neural network, as $p_\theta(x_i \mid x_{i-1}, ..., x_{i-7}) = f_\theta(x_{i-1}, ..., x_{i-7})[x_i]$. And neural networks do *not* have their number of parameters exponential in m. In fact, RNNs and Transformers have a *constant* number of parameters, independent of m[16].

Training the network from the previous section (but modified to accept 7 inputs vs 1 input, notebook here), and optimizing, we have:



The train perplexity shows a substantial improvement, at `5.80` for this constrained 8-gram model vs `11.87` for the "free" bigram model[17]. The learned function stores the next-token conditional probability vectors for all 67^7 possible sequences of prior tokens. Most of this space of 67^7 sequences are nonsensical (like "gsdaksx"). But of the ones in its training set, it is able to exploit shared structure between similar sequences[18] to predict similar conditional probabilities.

[16]The naive feedforward network we use here has its number of parameters increase linearly with m. Here, moving from 1 to 7 inputs, we have an increase from 7,360 to 13,504 params.

[17]And 11.87 is *the* lower bound on training perplexity possible with the bigram assumption, since the probability estimates for each conditional probability were at the global minima.

[18]Just as the bigram model did by placing all capital letters close to each other in input space.

1.6.1.1 Constraints as Generalization

Earlier in the widgets example, we saw that having $p_\theta(a) = \theta$ and $p_\theta(b) = \theta^2$ effectively *tied* their probability estimates together. This has a key consequence: If in a larger sample we saw a lower proportion of outcomes a, we couldn't decrease $p_\theta(a)$ without *also* decreasing $p_\theta(b)$.

This behavior carries over when we use a neural network. Consider the following:

- We train a bigram model (parametrized by a neural network), with sequences tokenized at the *word*-level.
- During training, the model learns an embedding vector for `"dog"` that is *very* close to the one for `"cat"`.
- A training minibatch contains the token pair `("dog", "sad")`, and the optimization process attempts to increase $p_\theta(\text{'sad'} \mid \text{'dog'})$.

Then, $p_\theta(\text{'sad'} \mid \text{'cat'})$ will *also* increase as `"dog"` and `"cat"` have quite close embedding vectors. In a way, this is a feature: the network is able to exploit the fact both `"dog"` and `"cat"` are "similar", and the former being followed by `"sad"` means the latter should also likely be followed by `"sad"`; it is able to generalize this update to another animal.

This generalization behavior is pointed out by Bengio et al. (2003). In their example, a well-trained model should assign the same probability to the sentences "The cat is walking in the bedroom" and "A dog was running in a room". Even if it sees only one during training, at test time the other will have similar embedding vectors, and in turn similar outputs.

But this generalization only works holds for a well-trained model, with a large enough corpus such that the embeddings have the correct separation. Suppose that in the "true" corpus, our dogs are sad and our cats are happy. The optimization process will only separate the embeddings sufficiently if we have **both** pairs `("dog", "sad")` and `("cat", "happy")` in our dataset. Without a requirement to predict different conditional probabilities $p_\theta(\text{'sad'} \mid \cdot)$ for each, we might have *unintended* generalization. And this is just one pair of embeddings.

1.6.1.1.1 An Emergent Property

Zooming in between the **computational**[19] and **representational**[20] levels here, we see a *macro-level* property arises: inputs with similar "meaning" have similar next-token predictions. At no point do we hand-specify the representation of each token; simply tuning the embeddings + weights over a sufficiently large corpus results in the "clustering" of tokens[21]. Referring back to the bigram model, at the *micro* scale we do not explicitly optimize the embeddings for uppercase letters for "grouping with other uppercase letters"; it emerges organically.

And this emergent property, of token sequences with similar "meaning" having similar inputs, allows a neural network to effectively *compress* the conditional probability matrix into its weights, taking up less memory *and* achieving near transfer.

[19] minimize negative log-likelihood
[20] the embedding vectors + network weights
[21] And this property was what drove key earlier methods such as Word2Vec (Mikolov et al. 2013).

1.7 Conclusions

1.7.1 Scaling and Generalization

At their core, large language models (LLMs) are not fundamentally different from the models we look at here: they too are functions that produce a conditional probability for the next token, conditioned on tokens already observed. Their functions have considerably more capacity: the largest neural network we look at has 13,504 params and has a context window of exactly 7 tokens. The largest GPT-3 model from Brown et al. (2020) has 175B parameters, and a context window of *up to* 2048 tokens.

But in keeping with the spirit of emergent properties (Anderson 1972), it's not *merely* that these language models are larger; they're also different. While they obey the "near transfer" properties described here (similar input embeddings $\rightarrow$ similar outputs), they also exhibit a *different* kind of generalization called in-context learning (ICL) (Min and Xie 2022). With ICL, the prompt itself can be composed of a few "examples" (e.g. a few translation input-output pairs), that dramatically improve the model's next-token prediction capabilities. This is hard to explain with just the "similar embedding vectors" property alone.

Our current understanding is that this behavior arises from "induction heads" (Olsson et al. 2022) that emerge organically during training in the Transformer networks (Vaswani et al. 2017) that parametrize these language models. One way to view this is a higher-order emergent property[22], that exists at the level of substructures in the network parameters (and not just the input embeddings as previously). ICL also appears to be dependent on *both* the use of Transformers, and the data distribution of natural language (Chan et al. 2022). In summary, our understanding of language models paramterized by neural networks is necessary, but *not sufficient* to understand LLMs, as there are emergent properties specific to the use of the Transformer architecture.

1.7.2 Data Distribution

In language modeling, we assume our data is i.i.d. drawn from a *true* distribution p. That is, the proportion of a snippet of text in an infinite-sized corpus should be $p(\text{text})$. But in our corpus, what we have can be better described as samples from $p(\text{text} \mid \text{time})$, with different values of time.

This has consequences: If the context is `The top song on the Billboard Hot 100 is`, then $p_\theta(\cdot \mid \text{context})$ will only be able to return conditional probabilities which minimized negative log-likelihood on the *training data*. The next token to this context snippet, in the real world changes on a weekly basis!

This problem is particularly exacerbated with fact-heavy completions. Recall that language modeling doesn't distinguish between the core structure of language, and one-off facts: they're all just tokens. The "facts" are stored in the weights of the model, no different than the conditional probabilities for the next token of any input sequence. Two potential ways to improve on this:

[22] Existing in between the computation and representation levels.

- **Edit model weights:** The recent work ROME (Meng, Bau, et al. 2022) introduced a method called "Causal Tracing" to find the subset of model weights that most influence the "next token" conditional probabilities for the "fact tokens" in a sample sentence, and an efficient editing mechanism for those weights. Follow up work then scales this to editing thousands of "stored facts" at once (Meng, Sen Sharma, et al. 2022).
- **Retrieval-based LMs**: Recent models such as RETRO (Borgeaud et al. 2022) and ATLAS (Izacard et al. 2022) *augment* a large transformer model with access to a database. Other works such as Lazaridou et al. (2022) directly use tools like search engines to add tokens to their input prompt. The overall effect is a model, which can learn to produce factual evidence by *copying* from retrieved data than by storing facts in its weights. Then, if the fact source is updated, the model simply copies the *new* information into its generated output.

1.7.3 Non-autoregressive models

The autoregressive, "generate a single token at a time, left to right" method we explore here is just *one* way to build a language model. A few recent papers that explore other strategies are[23]:

- The SUNDAE model introduced in Savinov et al. (2022) trains an autoencoder that iteratively denoises a snippet of text. Of note here is Table 4, where they show results on code-generation experiments, and how SUNDAE can account for context tokens both *before* and *after* the token to be generated.
- The SED model introduced in Strudel et al. (2022) uses diffusion directly on the embedding vectors of tokens. Instead of generating left-to-right, this allows them to iteratively refine an entire snippet of text, at each step.
- Fill-in-the-middle models explored in Bavarian et al. (2022) restructure the *training data* from (prefix, middle, suffix) $\rightarrow$ (prefix, suffix, middle). This allows a standard left-to-right architecture to be repurposed to "infill" text.

Overall, by leveraging shared structure, language models can store conditional probability tables for an exponential number of possible inputs. Many active lines of research continue to explore how these models work, and how they can be made better. And even today, because so many tasks can be rephrased as token prediction problems, they're beginning to power a large range of practical products, and their impact will only increase as they improve.

[23] And I note this is hardly exhaustive.

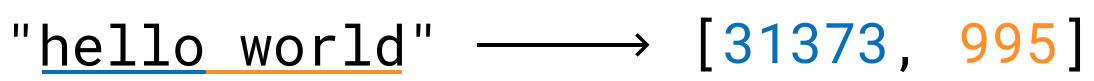
2 The Quirks of Tokenization

A lot of focus on improving neural-network based systems goes into the model architecture: the weights of the system, how they interact, etc. But unlike image classifiers, language models do not take in raw numbers as input: language is stored as "strings" after all. Tokenization (as first seen in Section 1.2.1) is the name of the process where a stream of language is converted into numbers that *can* be processed by a neural network. In this chapter, we'll take a deeper look at this process, and analyze how even modern tokenizers (such as GPT-4's) have surprising quirks that can be improved on.

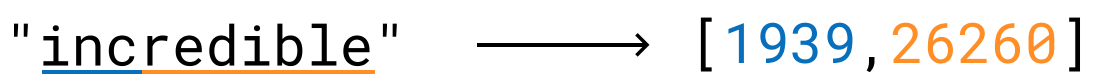
2.1 Tokenization

When we pass `'hello world'` into a system like GPT-4, we can't just directly send it to the neural network: such networks operate on continuous representations, not raw text. To transform raw text into a continuous representation, we use a two step process:

1. **Tokenization**: The string is broken into a sequence of integers, where each integer represents a "fragment" of the input. A natural way to do this is at the word level: if only two words "hello" and "world" were possible, we could assign each 0 and 1, and our representation would be `[0, 1]`. Indeed, using GPT-2's actual tokenizer, we'd have:

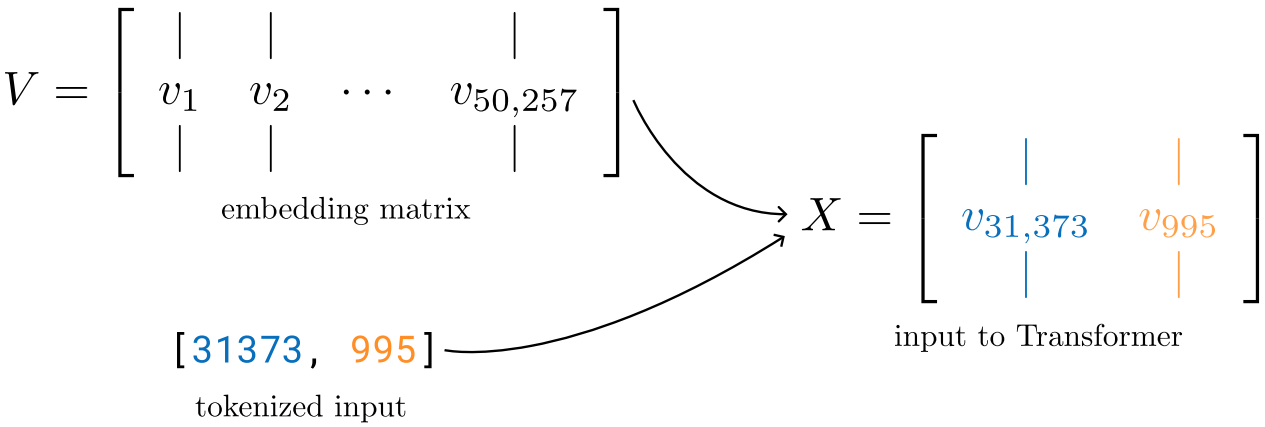


It isn't feasible to assign each unique word its own integer: English has hundreds of thousands of base words, and many of those words can be modified with prefixes/suffixes to create new words. In practice, we use subword tokenization: for less common words, we represent them with *multiple* tokens[1]:



2. **Embedding**: Once we've converted the input text into a sequence of tokens (each uniquely identified by an integer), we "pull out" the vector for each token (stored in the embedding matrix), and create a new seqeunce with the token integers replaced by their corresponding, continuous-valued vectors.

[1]Note here that "in", "credible" would be a more morphologically aligned breakdown. Tokenizers are trained with a statistics-based process (without any knowledge of morphology specifically), and it leads to such imperfections.



In this way, a text string is converted into a continuous, 2D matrix that can be subsequently used as input to a neural network.

2.2 Quirks

Although straightforward, the way typical tokenizers handle punctuation elements (such as capitalization and spaces) introduces a range of behaviors that to humans would seem surprising. There are three "big" ones:

2.2.1 Spacing

Tokenization often "absorbs" the space between words into the tokens using a "special" character; for GPT-2 this is Ġ. For instance, `'hello world'` as previously seen is encoded into `[31373, 1573]`. The individual tokens corresponding to those integers are `['hello', 'Ġworld']`.

This behavior is nice: it means tokens encoding the start of words have a Ġ prefixed to them; tokens that are always used as a suffix won't have them. *But*, there are plenty of words in the training data that aren't prefixed with a space either: two cases where you'd see this behavior are:

- At the start of sentences, such as `'hello'` in `'hello world'` above.
- Words inside brackets, such as `'(cucumber)'`

This means there's a *lot* of duplicate tokens for the same word, with and without Ġ. For instance, looking at just the two tokens here:

	With Ġ	Without Ġ
`hello`	23748	31373
`world`	995	6894

Scanning the entire GPT-2 vocabulary, there are 4,710 lowercased, unspaced tokens with a spaced variant, which is **~9.3%** of the vocabulary.

More importantly, so far we've examined pairs where *both* the spaced and non-spaced versions are single tokens each: subword tokenization complicates this further, by producing *different* tokenizations in general between both versions, such as:

- `' cucumber'` is encoded into `['Ġcuc', 'umber']`
- `'cucumber'` is encoded into `['c', 'uc', 'umber']`

The reason this is problematic as, in the embedding stage each unique token is assigned a unique vector: this means a language model needs to discover the same semantic meaning (`'hello'`) for both $v_{23,748}$ and $v_{31,373}$, instead of it being encoded from scratch.

2.2.2 Capitalization

Our issues aren't limited to spacing: since capitalization information is part of the tokens, `'Hello'` and `'hello'` are also encoded to two entirely different tokens

- Without spacing, `'hello'`, as previously is encoded into `31373`.
- Likewise but with capitalization, `'Hello'` is encoded into `15496`.

This issue then combines with the spacing issue, to produce *four* different tokenizations for the same word!

	With Ġ	Without Ġ
`hello`	23748	31373
`Hello`	18435	15496

Scanning the GPT-2 vocabulary again, this time for *capitalized* spaced, and *capitalized* unspaced versions of lowercased, unspaced tokens, we find 4,385 and 2,355 duplicates respectively. This means at least 11,450 tokens (~22.8% of the 50,257 token vocabulary) are redundant! And so far we've only looked at single token duplicates; more generally we'll have *four* different tokenizations with any number of subword tokens, such as:

	With Ġ	Without Ġ
`cucumber`	`['Ġcuc', 'umber']`	`['c', 'uc', 'umber']`
`Cucumber`	`['ĠC', 'uc', 'umber']`	`['C', 'uc', 'umber']`

2.2.2.1 All Capitals

To illustrate an even gnarlier case, if you have text that's in all capitals, that has its own representation seperate from either a first letter capitalized, or all lowercase. For example:

	With Ġ	Without Ġ
`hello`	`['Ġhello]`	`['hello]`
`Hello`	`['ĠHello]`	`['Hello]`
`HELLO`	`['ĠHELL', 'O']`	`['HE', 'LL', 'O']`

2.3 Fix: modeling a keyboard

As we observe, basic syntactic changes can lead to *dramatically* different representations of text for input to a language model. Such models have extremely large capacities, and can then "learn" to recognize the input in different forms, which is why they work well in practice. However, there's also potential in exploring ways to minimize this behavior from the start.

We can take inspiration from the special keys on a keyboard: for instance, to write "Hello", the actual keystrokes are `'<shift>hello'`. These don't exist on the final text in the training data, but we can reinsert them during preprocessing. Hence, to fix the pathologies described previously, we design the following three-stage pipeline:

1. Spacing: For any first letter in a word *not* preceded by a space, we prefix a `'<bksp> '`. That is, `'(hello)'` becomes `'(<bksp> hello)'`. Then, during tokenization we can reuse the `' hello'` token directly, without having to use an alternate tokenization for `'hello'`.
2. All Capitals: For continuous spans of all capitalized text `' LOREM IPSUM'`, we replace them with lowercased variants with special tags: `'<capss> lorem ipsum<capse>'`.
3. Start of word: For capitalized words, such as `'_Hello'`, we insert a `<shift>` at the front and replace with lowercase, such as: `'<shift> hello'`.

We implement all three of these stages using regexes, which are elaborated upon in Appendix B.

2.3.1 Worked example

To see this preprocessor in action, here's a worked example using one of the actual passages used during development: the first paragraph of the Wikipedia entry for the ARM big.LITTLE architecture.

> ARM big.LITTLE is a heterogeneous computing architecture developed by ARM Holdings, coupling relatively battery-saving and slower processor cores (LITTLE) with relatively more powerful and power-hungry ones (big). The intention is to create a multi-core processor that can adjust better to dynamic computing needs and use less power than clock scaling alone. ARM's marketing material promises up to a 75% savings in

power usage for some activities.[1] Most commonly, ARM big.LITTLE architectures are used to create a multi-processor system-on-chip (MPSoC).

Once processed the same passage now becomes the following:

```
<bksp><capss> arm<capse> big.<bksp><capss> little<capse> is a heterogeneous
computing architecture developed by<capss> arm<capse><shift> holdings,
coupling relatively battery-<bksp> saving and slower processor cores
(<bksp><capss> little<capse>) with relatively more powerful and power-<bksp>
hungry ones (<bksp> big).<shift> the intention is to create a multi-<bksp>
core processor that can adjust better to dynamic computing needs and
use less power than clock scaling alone.<capss> arm<capse>'<bksp> s
marketing material promises up to a 75% savings in power usage for some
activities.[1]<shift> most commonly,<capss> arm<capse> big.<bksp><capss>
little<capse> architectures are used to create a multi-<bksp> processor
system-<bksp> on-<bksp> chip (<bksp><shift> m<shift>p<shift>so<shift>c).
```

There are some interesting nuanced edge cases we correctly transform here:

- `' big.LITTLE'` is transformed into `big.<bksp><capss> little<capse>`. This means later during the tokenization process, we create the opportunity to reuse the general `' big'` and `' little'` tokens, instead of breaking `'LITTLE'` into a potentially multiple subword sequence as seen in Section 2.2.2.1.
- `'(big)'` is transformed into `'(<bksp> big)'`. This allows the reuse of the spaced, `' big'` token here, instead of needing two seperate `' big'` and `'big'` tokens.
- In the particularly special case of `'(MPSoC)'`, we transform it into `(<bksp><shift> m<shift>p<shift>so<shift>c)`. We program the preprocessor to only use `<capss>` and `<capse>` if the entire word is capitalized; if not (which would indicate a special acronym, as in here), it switches to inserting individual `<shift>`'s prior to each character.

2.4 Experimental Setup

To compare how the preprocessing affects both what tokens are created, and downstream performance, we train two tightly matched tokenizers, identical in all regards except that one has the preprocessing described above applied to its training data. Specifically:

- **Data**: We sample 75 million words from the March 2022 dump of English Wikipedia (Foundation, n.d.). For articles smaller than 4,096 words, we use the whole article; for those longer, we randomly select a single, continuous span of 4,096 words. This is the only dataset used: normally, to train a full LLM this data would be mixed in with open-web data, but since our needs are small we can constrict ourselves to this high-quality data only.

- **Model**: We use the SentencePiece tokenizer library (Kudo and Richardson 2018) to train the tokenizers. We train a Unigram language model, owing to the finding of Bostrom and Durrett (2020) that the Unigram approach is better capable of creating subword units aligned with morphology [2]. We use a vocabulary of 16,384, and ensure the four special tags[3] and punctuation characters are segmented *before* the actual words.

2.5 Results

2.5.1 Classification Performance

We can quickly test the two tokenizers ability to preserve relevant word-level semantics by using them in a linear classification task. Specifically, we use four categories of 20 from the 20 Newsgroups dataset (Lang 1995) [4]. A standard baseline is to use a tf-idf vectorizer to convert the text into vectors, and then use a Ridge regression model on top.

We make a simple modification: in the tf-idf vectorizer, instead of breaking at the whole word level, we use our tokenizers to break the text into subwords. This is a straightforward change, achieved with a 4-line function that defines a custom way to convert a single string into a list of (token-level) strings:

```
def analyzer(text):
    if preprocess:
        text = replace_caps(text)
    return s.encode_as_pieces(text)
```

The rest of the pipeline follows as per usual, and we end up with the following metrics:

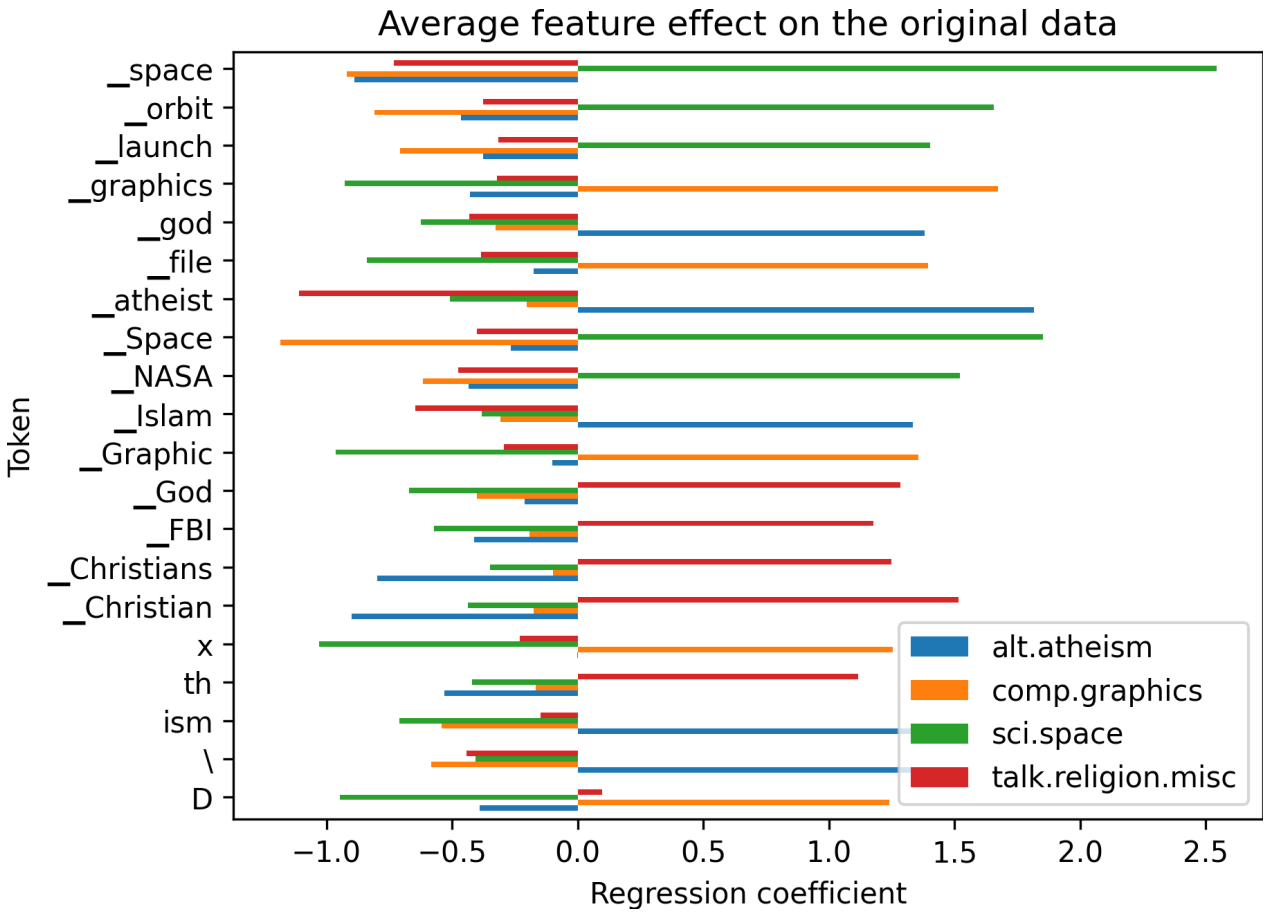
	f1 score
With preprocessing	75.7%
Without preprocessing	74.4%

As we see, our preprocessing creates tokens that produce a 1.3% lift in performance, everything else held constant.

2.5.1.1 Intepretability

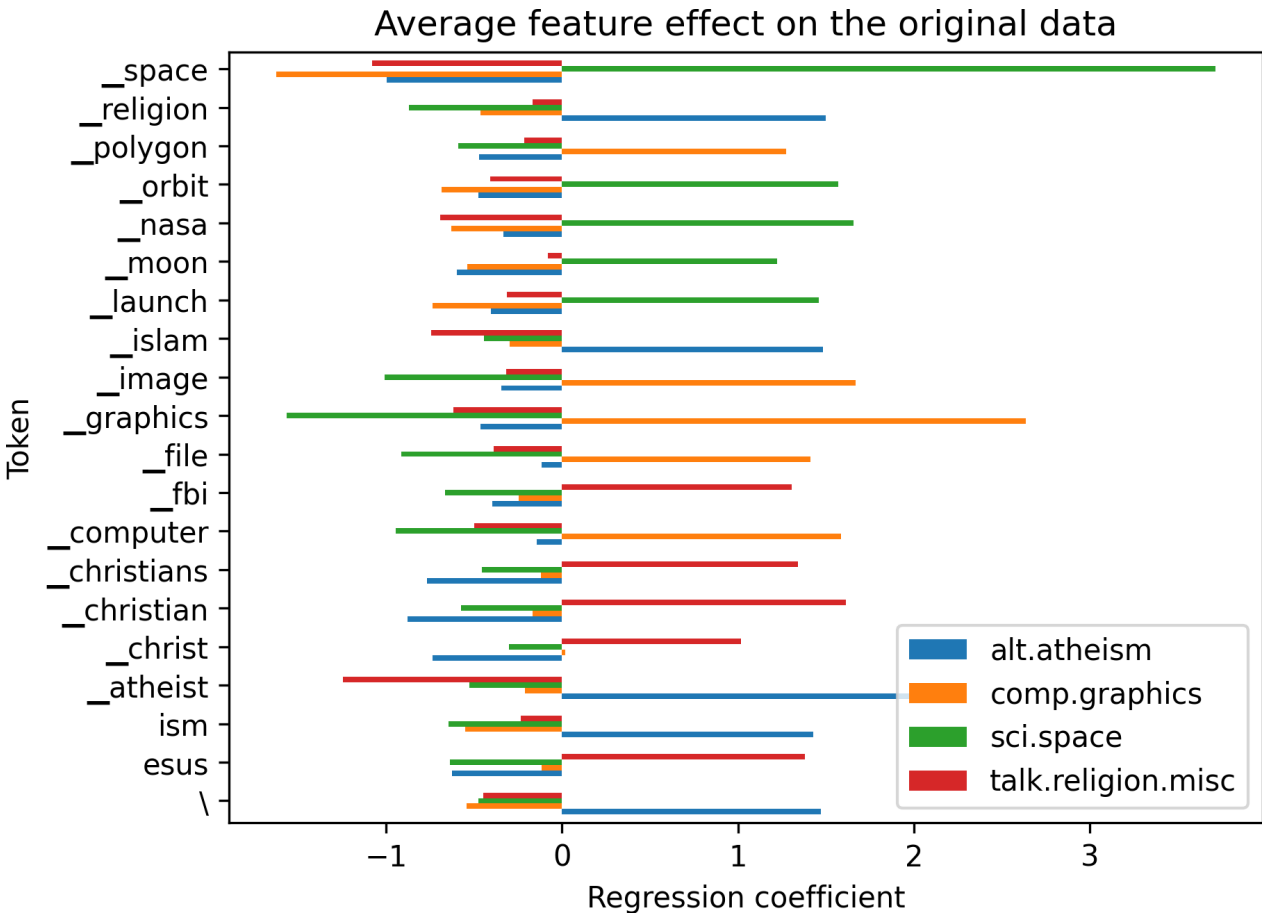
Since we're using a linear model, we can directly examine the coefficients for interpretability. Here are 5 tokens, for each of the 4 categories that have the highest coefficient (for that category), plotted with their coefficient for the other categories. For the version without preprocessing, we have:

[2] As opposed to the other major technique, Byte Pair Encoding (Sennrich, Haddow, and Birch 2016)
[3] These are `'<shift>'`, `'<capss>'`, `'<capse>'`, `'<bksp>'`
[4] We also filter to only keep datapoints more than 100 characters long, to preserve distinguishability between classes.



As we see, our regular tokenizer dedicates a number of single tokens to common acronyms (such as NASA and FBI), and the regression algorithm is able to use them as signal. Most interestingly, since `'_god'` and `'_God'` are distinct tokens, the regression algorithm learns to identify the former with the atheism forum, and the latter with the religion forum.

Our tokenizer using processed text also has similar results (such as dedicating tokens for now lower-cased versions of acronyms, like `'_nasa'` and `'_fbi'`). `'_god'` is now missing from the top tokens, since it is associated with at least two categories strongly.



From interpreting coefficients, it appears that the signals being leveraged are broadly the same as the unprocessed, cased text; the 1.3% increase in performance likely is from leveraging more general features for the harder examples.

2.5.2 Summary Statistics

For each spaced, lowercase token (such as `'_hello'`), we can identify if they have a variant (such as `'_Hello'`) and count the totals. For the two tokenizers, we have:

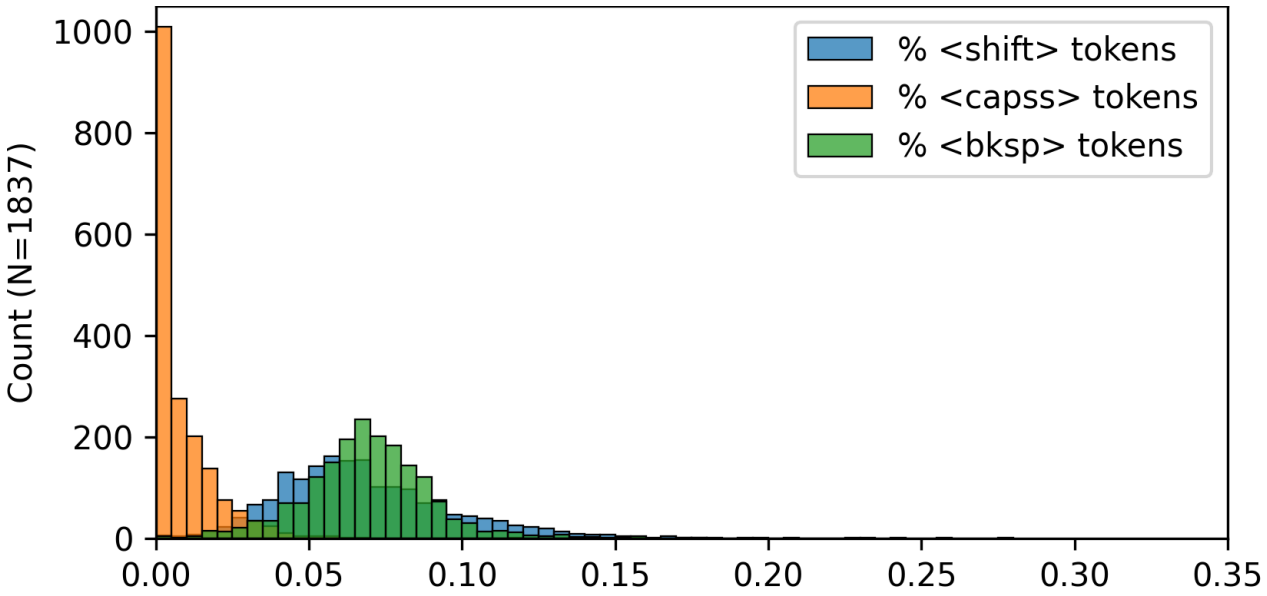
	Unspaced	Unspaced + Capitalized	Spaced + Capitalized
With preprocessing	341	0	0
Without preprocessing	680	1000	247

As we see, the capitalized duplicates are eliminated entirely, and the number of unspaced duplicates[5] are cut in half. In total, this means going from a tokenizer that is 11% duplicates to 2% duplicates, a sharp improvement.

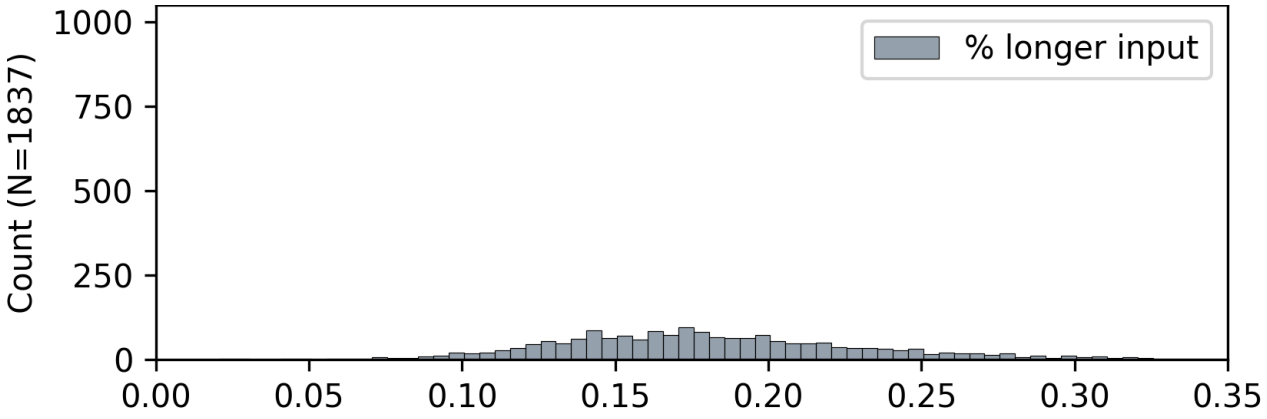
[5]We still expect unspaced duplicates to be non-zero, due to suffixes. For instance, we may use both `_ian` and `ian` tokens, since the first is a name, whereas the latter is the suffix to many words.

2.5.3 Distributional Statistics

Adding the special tokens results in our inputs being longer; since LLMs are linear in cost to the number of tokens generated, we should calculate just how much more expensive our inputs become. Using the 20 newsgroups dataset, we can empirically calculate the percentage of tokens that are one of the special tokens:



Over this input distribution, the median percentage of `<shift>`, `<capss>` and `<bksp>` tokens are 6.5%, 0.4%, and 6.9%. We can also calculate the distribution of increase in length:



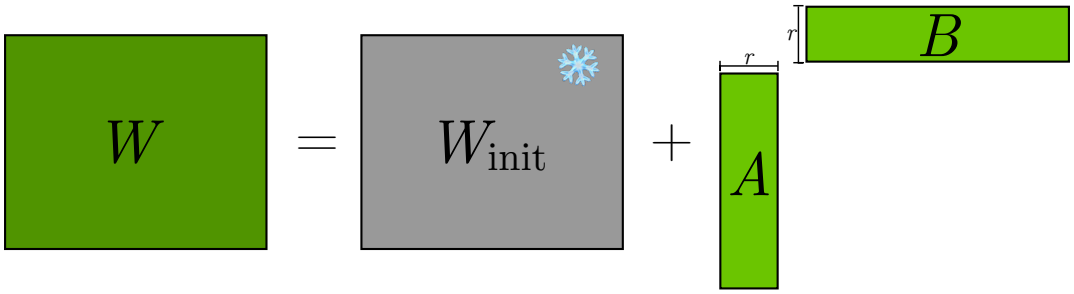
As we see, the median increase is 17%; in other words, our inputs will require 17% more compute to process (and have a corresponding increase in cost).

2.6 Conclusion

This chapter presents an alternate way of doing tokenization, where we see a language model as a rather literal model of keyboard button presses, and "extract" special tokens for punctuation to more closely align final text (on the open web) to the actual inputs on a keyboard. We see that it substantially trims redundant tokens (with the aim of promoting better generalization), and establishes improved performance on a baseline model. The 17% cost is steep, but can likely be alleviated by processing the special tokens differently (such as adding them to the previous token), since the information content in them is rather low, which we leave for future work.

3 LoRA and Weight Decay

An increasingly popular way to use LLMs in industry settings is to "finetune" a pretrained model for improved performance on a specific task. LoRA (Hu et al. 2021) is amongst the most popular ways to efficiently do this: instead of tuning the billions of weights of the full model (as would be normally necessary), we add small "adapter" weight matrices that *modify* the original weight matrices, and tune those instead.



This chapter dives deeper into a curious behavior: although LoRA is commonly seen an drop-in for full finetuning, its interaction with weight decay means it solves a *different* optimization problem than full finetuning. Namely, one where the solution weights are regularized towards the frozen base model ($W \to W_{\text{init}}$), instead of $W \to 0$ as in full finetuning.

This means, given increasingly more resources (even equalling that of full finetuning), LoRA does *not* increasingly better approximate full finetuning, because its objective function is implicitly different to that of full finetuning. This, depending on use case can either be seen as a **bug or a feature**, but is something practitioners should explicitly account for.

3.1 Recap: Finetuning

With LLMs, we typically finetune an initial model (that is "good" on a wide range of text-to-text tasks) to boost performance on a *specific* task of interest (e.g. generating database queries from natural language). We do this in a two-step process:

- First, creating a finetuning training dataset $(x_i, y_i)_n$, which contain pairs of inputs x and targets y.[1]

[1] In the database query example, the x's can be strings in English, and the y's are then strings corresponding to the query translated from English into the query schema.

- Optimize the weights of the initial model such that our finetuning training dataset $(x_i, y_i)_n$ becomes more "probable". The idea here is that a model that is more likely to generate the correct answers y on x's from our training set, will generalize and also be more likely to generate y's on *new* x's.

3.1.1 Full Finetuning

Full finetuning means we tune *all* the weights of the model. For a model such as GPT-3 175B (Brown et al. 2020), this means giving our optimization algorithm 175 billion numbers it can "dial" up and down as needed to make our finetuning training data more "probable". Let's dig a bit deeper, and more concretely define what we mean by weights here.

Each layer in a Transformer is primarily made of two components: a multihead attention network, followed by a feedforward network. This means the bulk of the "weights" that make up each layer are stored in six matrices[2], as shown. θ then, is used as shorthand refer to *all* the weights, stored in all the matrices across all the layers of the model.

$$\theta = \left[W_q^1, W_k^1, W_v^1, W_o^1, W_{\text{in}}^1, W_{\text{out}}^1, W_q^2, \ldots\right]$$

multihead attention — feedforward network

In full finetuning, every single weight in θ is opened up for updating. Our aim is to produce updated weights that minimize the negative log likelihood (NLL) as shown on the left[3]. There's no closed form way to get the "optimal" weights, so we solve the optimization problem by repeatedly applying many steps of gradient descent, as shown on the right.

$$\min_{\theta} \underbrace{\left[-\log P_\theta(y \mid x)\right]}_{L} \quad \rightarrow \quad w \leftarrow w - \alpha \frac{\partial L}{\partial w}$$

PROBLEM SOLUTION

Now, directly doing gradient descent this way would quickly lead to overfitting[4], so we usually regularize the problem. With LLMs, the regularization tool of choice is usually weight decay. Specifically, when using vanilla SGD[5], weight decay is equivalent to having a term in the loss equal to the squared sum of the weights:

[2] Note that if you use a GLU-variant activation (Shazeer 2020), then you add in a 7th "gating" weight matrix.
[3] Possible meaning "non-zero probability"
[4] In that the weights would all be optimized to perfectly repeat y_i for any x_i in the finetuning training set, at the *expense* of performing *much* worse on any x_i not in the training set.
[5] Stochastic Gradient Descent, a.k.a. the core workhorse of deep learning. In practice we use more sophisticated momentum-based methods, whose impact is described in Appendix A.

$$R(\theta) = \sum_i \sum_j [W_q^1]_{ij}^2 + \cdots$$

Hence, the overall objective now is as follows (where λ is a hyperparameter controlling the strength of the weight decay):

$$\min_\theta \left[\underbrace{-\log P_\theta(y \mid x)}_{L} + \frac{\lambda}{2} R(\theta) \right]$$

Differentiating this to objective to get the gradient, we notice the gradient update has two distinct terms[6]: the first corresponding to the minimizing the negative log likelihood as before, and a new second term $-\alpha\lambda w$ that pushes the weight towards the origin 0.

$$\begin{aligned} w &\leftarrow w - \alpha \left(\frac{\partial L}{\partial w} + \frac{\lambda}{2} \frac{\partial R}{\partial w} \right) \\ \Rightarrow w &\leftarrow w - \alpha \left(\frac{\partial L}{\partial w} + \lambda w \right) \\ \Rightarrow w &\leftarrow w - \alpha \frac{\partial L}{\partial w} - \alpha \lambda w \end{aligned}$$

Which means the regularized problem now looks like:

$$\underbrace{\min_\theta \left[\underbrace{-\log P_\theta(y \mid x)}_{L} + \frac{\lambda}{2} R(\theta) \right]}_{\text{PROBLEM}} \longrightarrow \underbrace{w \leftarrow w - \alpha \frac{\partial L}{\partial w} - \alpha \lambda w}_{\text{SOLUTION}}$$

In summary, adding a squared sum of weights loss is equivalent to subtracting a scaled version of each weight at each gradient descent step. This shifts the minima towards where the weights are closer to 0[7]; i.e. no one weight can have extremely large effects on the predictions of the model.

Full finetuning is highly flexible, but also *extremely* memory intensive: you generally need at least 3x the memory[8] required for the model itself, to account for its gradients and optimizer state. This was not an issue when models were $O(100M)$ params, but is certainly so today where they're regularly $O(10B)$ to $O(100B)$ params. Moreover, if you have 10 sub-tasks in your application (where you're tuning the model for each task), full finetuning requires you to host 10 versions of the model (as if hosting 1 isn't expensive as is!).

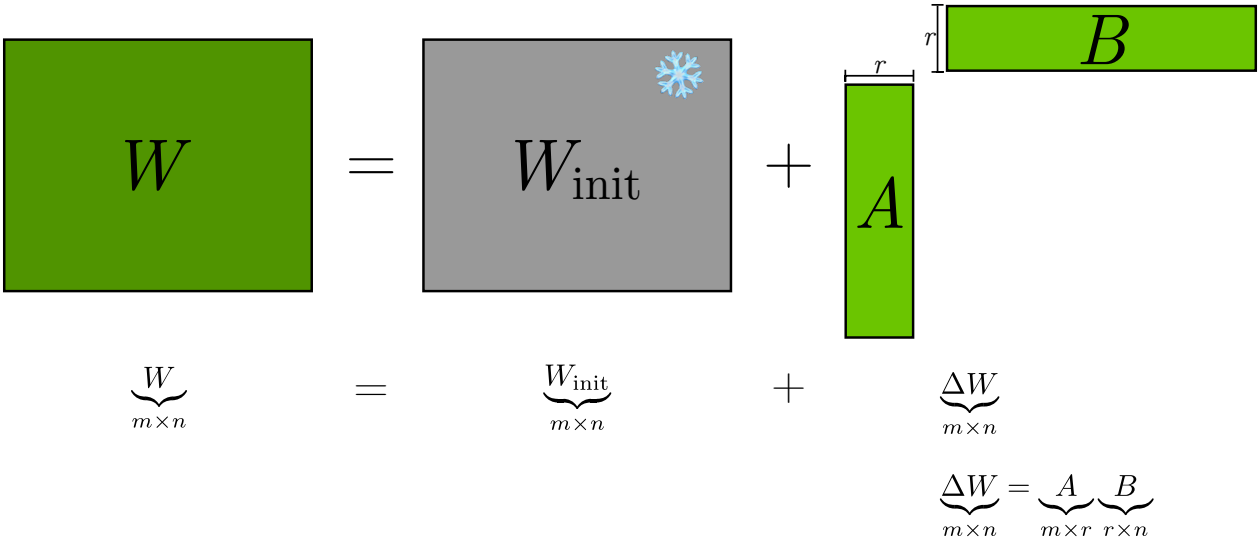
[6] This directly stems from the fact that the gradient of a sum (here, the two terms are NLL and regularization) equals the sum of the gradients (of each term).

[7] To reason why this is true, notice that the larger the weight, the "more" of it is subtracted away from itself.

[8] Assuming you're using an optimizer with some form of momentum (vanilla SGD doesn't need an optimizer state). It goes up to 4x for Adam, as it has two states: an exponential moving average of both the gradient means, and the gradient-squared means.

3.1.2 LoRA finetuning

LoRA (Low Rank Adapter) finetuning takes a different approach: instead of tuning the massive weight matrices of an LLM directly, we use a pair of small adapter matrices for each weight matrix we want to tune, of the following form:



That is, for each initial, frozen weight W_{init}, we have adapter matrices A and B. These two matrices are multiplied together to form ΔW, which is a low rank "adjustment" matrix for W_{init}, forming the adapted, tuned matrix W. This cuts the number of free parameters significantly: assume the original matrix W_{init} is $4,096 \times 16,384$. In the original, we'd have 67 million parameters to tune just for this one weight matrix, as follows:

$$4,096 \times 16,384 = 67,108,864 \approx 67 \text{ million}$$

With LoRA with rank $r = 4$, we only have:

$$4,096 \times 4 + 4 \times 16,384 = 81,920$$

This is less than 0.1% of the original number of parameters; the added overhead of storing 3 variants of these values (weights, gradients and optimizer states) is negligible compared to the memory used by the model itself.

Moreover, since the initial weights are "shared" across all the finetuning runs, at inference time we only need to load one copy of the initial model to be shared across many finetuned versions, with inference for each task using their own per-task adapter matrices. This makes having a "per-task" tuned LLM in an application not only viable, but easy.

3.2 The Interaction

Now that we've covered what LoRA is, we can begin to discuss how it interacts with weight decay to produce a feature/bug. Since A and B are the "actual" matrices we're performing gradient descent on, the weight decay term in the objective looks like this, in that we're moving the minima towards where the adapter matrices are closer to 0:

$$R(\theta) = \sum_i \sum_j [A_q^1]_{ij}^2 + \sum_i \sum_j [B_q^1]_{ij}^2 + \cdots$$

Let's contrast this with the formulation in full finetuning:

Weight decay (full finetuning)	Weight decay (LoRA default)
$W \to 0$	$A \to 0, B \to 0$ $\Rightarrow AB \to 0$ $\Rightarrow W = W_{\text{init}} + AB \to W_{\text{init}}$

- In full finetuning, we have $W \to 0$, in that the weight decays to 0 directly.
- However, in LoRA, because A and B decay to 0, in effect we have $W \to W_{\text{init}}$ instead.

This means LoRA solutions are biased towards the original frozen weight matrices, unlike in full finetuning, where they're biased towards zero. And this behavior does not go away with increasing the LoRA rank r - one could increase it all the way to infinity(!), and the optimization process would still be biased towards the original frozen weights instead of zero. That is, even in the limit, LoRA does not approximate full finetuning, but a different objective.

3.2.1 A fix

If we wanted the full adapted matrix to go towards zero (as would happen in full finetuning), we'd need a regularization term where the *entire adapted* weight matrix goes to zero, as follows:

$$\begin{aligned} R(\theta) &= \sum_i \sum_j [W_q^1]_{ij}^2 + \cdots \\ &= \sum_i \sum_j [W_{q,\text{init}}^1 + A_q^1 B_q^1]_{ij}^2 + \cdots \end{aligned}$$

This is actually straightforward to derive, and yields a pair of update equations that can be implemented much like standard weight decay. First, start at the core definition of weight decay, which involves calculating the gradient of the weight w.r.t. the regularization term:

$$w \leftarrow w - \alpha \left(\frac{\partial L}{\partial w} + \frac{\lambda}{2} \frac{\partial R}{\partial w} \right)$$

Second, compute the gradient of A and B[9] w.r.t. the "corrected" $R(\theta)$ above. This yields:

$$\frac{\partial R}{\partial A} = 2(W_{\text{init}} + AB)B^T$$
$$\frac{\partial R}{\partial B} = 2A^T(W_{\text{init}} + AB)$$

Inserting back into the definition of weight decay, we get the following concrete update equations for A and B:

$$A \leftarrow A - \alpha \frac{\partial L}{\partial A} - \alpha\lambda(W_{\text{init}} + AB)B^T$$
$$B \leftarrow B - \alpha \frac{\partial L}{\partial B} - \alpha\lambda A^T(W_{\text{init}} + AB)$$

3.2.1.1 In code

This is what the standard formulation of weight decay in the Optax (Babuschkin et al. 2020) library looks like. It's quite clean: add a `weight_decay` (λ) scaled version of the parameter `p` to its current update `g`[10].

```
# from https://github.com/google-deepmind/optax/blob/master/optax/_src/transform.py#L766
def update_fn(updates, state, params):
    if params is None:
        raise ValueError(base.NO_PARAMS_MSG)
    updates = jax.tree_util.tree_map(
        lambda g, p: g + weight_decay * p, updates, params)
return updates, state
```

To modify this to implement the math we just described above takes some of extra code, mostly in extracting the `W_init`, `A` and `B` matrices[11]. The core logic is just the two lines 18 and 20.

[9] We're dropping the q, k, … subscripts as the derivation is identical for all the weight matrices.

[10] We add terms to the update, as the *subtraction* of the update happens at the very end.

[11] This exact formulation assumes the adapters are defined inside the same layer as the original matrix; that is the params dict looks like `{'params': {'kernel': ..., 'kernelA': ..., 'kernelB': ...}}`. The actual implementation will depend on how the LoRA adapters have been defined (even though the underlying math will remain the same).

```
def update_fn(updates, state, params):
    def per_param_update_fn(path, update, param):
        # Get the params dict for the layer as a whole.
        param_name = path[-1].key
        # If current parameter is an adapter matrix.
        if param_name in ['kernelA', 'kernelB']:
            layer_params = params
            for dict_key in path[:-1]:
                layer_params = layer_params[dict_key.key]

            # Extract the initial weight matrix and adapter matrices.
            W_init = layer_params['kernel']
            A = layer_params['kernelA']
            B = layer_params['kernelB']

            # Compute the corrected decay term.
            if param_name == 'kernelA':
                decay_term = (W_init + A@B)@B.T
            else:
                decay_term = A.T@(W_init + A@B)

        # If current parameter is *not* an adapter matrix, use
        # default version of weight decay.
        else:
            decay_term = param

        return update + weight_decay * decay_term

    if params is None:
      raise ValueError(base.NO_PARAMS_MSG)
    updates = jax.tree_util.tree_map_with_path(
        per_param_update_fn, updates, params)
    return updates, state
```

3.3 Extension: Momentum and Weight Decay

You've likely noticed we spent a substantial amount of time explicitly working out the gradient for the regularizer term $R(\theta)$, instead of just directly absorbing it into L and letting autodiff take care of all this for me. That's because the equivalency (weight decay of gradients = adding an L_2 regularization term to the loss) is only true for non-momentum based optimizers like vanilla SGD, *not* modern momentum based optimizers such as Adam (Kingma and Ba 2015) or AdamW (Loshchilov and Hutter 2019).

The AdamW paper[12] is a solid, in-depth read to understand why this is the case, but in brief: to do weight decay we want to subtract away a scaled version of the parameter's value *at the current timestep*. However, adding an L_2 regularization term to the loss directly means the regularization gradient is added to the *momentum* state of the optimizer: the *past* value of the parameter now influence its weight decay, not just the current value. The overall effect here is parameters which had "large" values early in training are regularized *less*, defeating the point of weight decay!

The way modern optimization libraries such as Optax implement AdamW is by first implementing Adam's transformation of the gradient as a seperate subroutine[13] adam, that:

- takes in the NLL loss gradient $\frac{\partial L}{\partial w} = g$
- as well as past optimizer states (m, v)
- return a "transformed" gradient, an update u_t, that is, adam$(g, m, v) \to u$.

From there on out, the weight decay looks just like it did before, but swapping in u.

$$w \leftarrow w - \alpha \left(u + \frac{\lambda}{2} \frac{\partial R}{\partial w} \right)$$

Which means, a version of our corrected (decays to 0) LoRA update that is compatible with AdamW looks like[14]:

$$A \leftarrow A - \alpha u_A - \alpha\lambda(W_{\text{init}} + AB)B^T$$
$$B \leftarrow B - \alpha u_B - \alpha\lambda A^T(W_{\text{init}} + AB)$$

The code snippet above (implementing the decay to 0 LoRA) is actually already compatible with AdamW in Optax. This very nice behavior comes mostly from free because of the fact AdamW in Optax is *already* a decomposed chain of three operators (adam, weight decay, and then scaling by the learning rate); Optax's actual implementation is as follows:

```
# from https://github.com/google-deepmind/optax/blob/master/optax/_src/alias.py#L298
return combine.chain(
      transform.scale_by_adam(
          b1=b1, b2=b2, eps=eps, eps_root=eps_root, mu_dtype=mu_dtype),
      transform.add_decayed_weights(weight_decay, mask),
      _scale_by_learning_rate(learning_rate),
  )
```

All we'd need to do is create a new optimizer, where we swap in the `transform.add_decayed_weights` with our custom version, and we'd be set.

[12] Which pointed out this non-equivalency, and produced a version of Adam that "decoupled" weight decay.

[13] Note that this is *not* a simplification, the Optax library has an actual function called `scale_by_adam` here that does exactly this.

[14] An alternate, equivalent way to look at this is that we use Adam on the NLL loss, and pure SGD on the $R(\theta)$ loss.

3.4 Conclusion

In summary, LoRA has a different implicit objective than full finetuning, but it's also easy to correct if desired. That's it, really!

To my knowledge, there isn't literature documenting this interaction of LoRA with weight decay in depth. Conjecturing purely from first principles, I'd argue the default behavior is both a feature *and* a bug, depending on the amount of data - when there's a very few number of training points, it is a feature *because* it regularizes the updated model to stay close to the initial, "generally-capable" one. However, it's a *bug* when given large amounts of data, as the optimization process is less capable of straying too far from the base weights, *even if* it would aid end-task performance.

That said, as neat as the math is, empirical results are the only truth here. With so many free parameters, it may well turn out to be in practice there are solutions just as good (when regularized to be close to W_{init}) as full finetuning (regularized close to 0) given enough capacity.

4 Tensor Parallelism with `jax.pjit`

One of the challenges of training large language models (LLMs) is that their size can easily exceed 10B+ parameters (and the largest at 100B+ parameters). Pure data parallel strategies are no longer viable as the *model itself* no longer fits on single devices. In this chapter, we'll explore the mathematical underpinnings of tensor parallelism, which allows us to train a very large model while keeping it "split" across the many devices of a GPU/TPU supercomputer.

4.1 Intro: Parallelism

4.1.1 Data Parallelism

Until recently, large scale training of deep learning models have primarily used data parallelism:

- Each device stores a full copy of the model, and receives a "shard" of the batch (if the full batch is 8 training examples, split along 2 devices each device receives 4 examples).
- Each device independently computes the loss, and its gradient (w.r.t. the parameters) using its data shard.
- Only *once* during each step, they synchronize their gradients and update their own copy of the model.

As long as a full copy of the model[1] fits on device, this general strategy can scale gracefully to the typical maximum of 8 GPUs on a single host, and was the likely strategy used to train the "big" (213 million params) Transformer with in the original Attention is All You Need (Vaswani et al. 2017, 7) paper [2]. Properly optimized, data parallelism scales to even hundreds of GPUs on multiple hosts.

However, data parallelism isn't enough when the model itself can no longer fit on a single device. This is where model parallelism comes in.

4.1.2 Tensor Parallelism

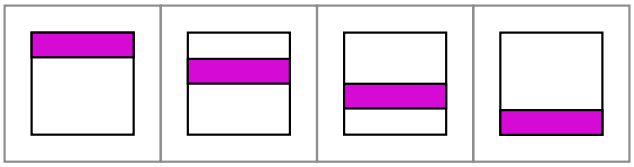
Model parallelism is when we split the model itself across multiple devices. Tensor Parallelism ("sharding") is one of two ways to do this; the other is Pipeline Parallelism ("pipelining"). The latter is briefly discussed at the end, but the focus in this chapter really is on the former.

[1] *and* a copy of the optimizer state and gradients when we're training, so potential total memory use of upto 3x the size of the model itself.

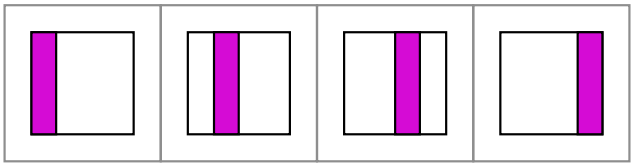
[2] Specifically, in section 5.2 the authors note "We trained our models on one machine with 8 NVIDIA P100 GPUs."

Tensor parallelism is the answer to this question: *what if* we could compute the activations of *every* layer of our model, *distributed* across all our devices?

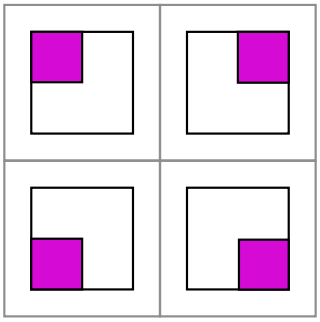
Suppose we have 4 devices: with standard data parallelism we make each device compute *all* the embedding dimensions for 1/4th of the batch:



But perhaps we could make each device compute 1/4th the embedding dimensions for the entire batch, like this:



Even more: instead of sharding on one axis, we could shard both axes. What if we arranged these 4 devices in a 2×2 mesh, such that the first (top left) device computed 1/2 the embedding dimensions for 1/2 the batch?



This is the big idea behind tensor parallelism: arranging our devices into a 2D mesh, and then sharding *both* our weights and activations on *both* axes, for all the layers. That is, **each** device holds a single "shard" of **every** layer in the model. When done properly, it is possible to run calculations with only *one* full copy of the model distributed across all the devices.

We'll start ground up: at the level of the dot products themselves, and see how sharding allows us to do very large matrix multiplies, by trading off increased communication for reduced memory use. Then, we'll scale it up to a full model in JAX, implementing sharding with `pjit` on a 15B language model for inference, with focus on the exact code changes, keeping them minimal.

4.2 Intro: Dot products

Let's start with an observation: any dot product between two vectors can be broken down into the sum of multiple smaller dot products. Suppose:

$$a = \begin{bmatrix} 1 \\ 0 \\ 2 \\ -1 \end{bmatrix}, b = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 2 \end{bmatrix}$$

Then, the dot product of these two vectors of length 4 is

$$a \cdot b = (1 \times -1) + (0 \times 2) + (2 \times 0) + (-1 \times 2) = -3$$

But we could easily re-write that expanded calculation as

$$\underbrace{[(1 \times -1) + (0 \times 2)]}_{-1} + \underbrace{[(2 \times 0) + (-1 \times 2)]}_{-2}$$

Each of these two terms individually is also a dot product of two vectors of length 2. Recoloring the original vectors, we can imagine them as composed of two "partitioned"-vectors:

$$a = \begin{bmatrix} 1 \\ 0 \\ 2 \\ -1 \end{bmatrix} \quad b = \begin{bmatrix} -1 \\ 2 \\ 0 \\ 2 \end{bmatrix}$$

Now, say I wanted my friend to help out with this tedious calculation. If I calculated the dot product with the first partition of each vector (getting back -1), they'd only need to return the *result* (-2) of their partition (and not their entire sub-vectors) for me to calculate the full dot product, $(-1) + (-2) = -3$.

4.3 Intro: Matrix multiplies

Let's build on this with another observation: In a matrix multiply $AB = C$, C is simply a storage mechanism for the pairwise dot-products of all the (row) vectors of A and (column) vectors of B[3]. Specifically, let:

$$A = \begin{bmatrix} 1 & 0 & 2 & -1 \\ 2 & 1 & 0 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 0 & -1 \\ 1 & 2 \\ 2 & 0 \\ 0 & 2 \end{bmatrix} \quad AB = C = \begin{bmatrix} 4 & -3 \\ 1 & -4 \end{bmatrix}$$

[3]Why the *row* vectors of A and the *column* vectors of B? Why not just the column vectors of both? This is mostly due to convention, as confusing as it can be for newcomers. I like the Mnemonic RAC-B ("rack b"), rows of A, columns of B.

A's first row vector and B's second column vector should seem familiar: we *just* took their dot products. And as expected, the element of the *first row, second column* of C is that dot product -3. This perspective also neatly explains two facts about matrix multiplication:

- Why C is a 2×2 matrix: A has two row vectors, and B has two column vectors, resulting in a 2×2 matrix to capture all the pairwise dot products. (Likewise, if A had 3 row vectors, C would be of shape 3×2).
- Why the "inner axes" (A being 2×4, B being 4×2) have to match: we can't take dot products of vectors of different lengths. Take note of this "inner axes" terminology, we're about to build on this right now!

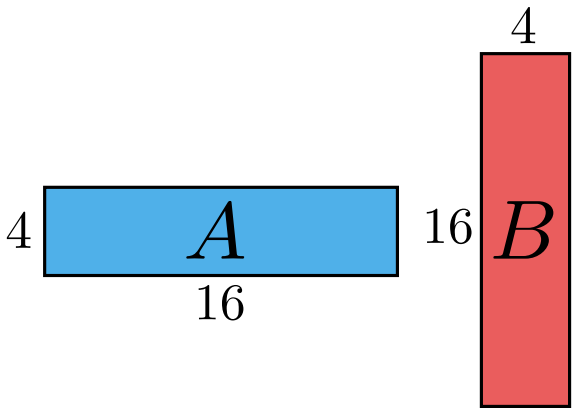
Both combined, we have the general rule for the shapes: $\underbrace{A}_{n \times d} \underbrace{B}_{d \times m} = \underbrace{C}_{n \times m}$

4.4 Sharding: A Matrix Multiply

The *vast* majority of compute time in deep neural networks is spent on the matrix multiplies (matmuls) between data (activation) matrices and weight matrices. We focus on a single matmul in this section, building up two cases (1A, 1B and 2), that together can explain *every* possible sharding pattern.

Let's start:

- Suppose we now have a new A matrix of shape 4×16, and a B matrix of shape 16×4. That is, each has 4 vectors of length 16 each. We want to compute $C = AB$, where C will be a 4×4 matrix.



- Also suppose we have 8 GPUs/TPUs, which are much slower at *multiplying* matrices than adding matrices[4]. We arrange these 8 devices in a 2×4 mesh.

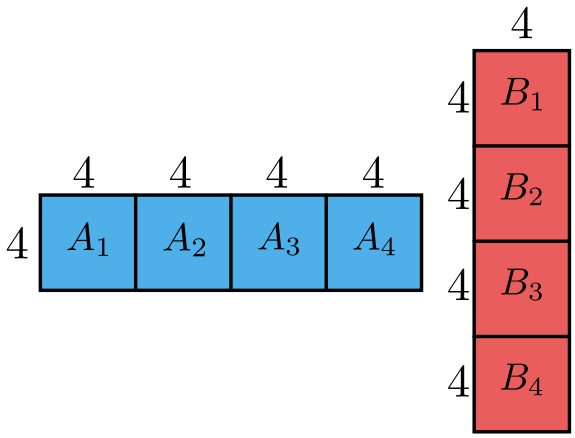
[4]For an $N \times N$ matrix, matrix multiplication has time complexity $O(N^3)$ while matrix addition is $O(N^2)$. For a large enough matrix, the speedup from the parallel matrix multiplies can outweigh the cost of communicating then adding afterwards.

Our goal here is to do as little compute as possible here: in the **final version**, we should split up this large matrix multiply such that we only compute *one* full copy of $C = AB$ split across all 8 devices.

4.4.1 Case 1: Inner Axes

For now, let's try a version of the problem that may be easier to reason about: split up AB such that *each device* has one full copy of $C = AB$.

To get started, we can partition both matrices, just as we previously partitioned vectors. Note that unlike vectors, with matrices we have multiple axes we can "slice" on. For now, let's use this pattern, where each "sub-matrix" is a 4×4 matrix:



Since A is the first matrix (row vectors) and B is the second matrix (column vectors) in the matmul AB, we're effectively partitioning along the "inner axes"; cutting each vector into multiple sub-vectors, just as in the dot product example. To make this clearer, we might write down this "sharding configuration" in pseudocode as:

- A: `(full, sharded)`
- B: `(sharded, full)`

Continuing this reasoning, just as we decomposed the large dot product into the sum of multiple smaller dot products, we can decompose this larger matrix multiply into the sum of smaller matrix multiplies. Specifically, $AB = A_1B_1 + A_2B_2 + A_3B_3 + A_4B_4 = C_1 + C_2 + C_3 + C_4$.

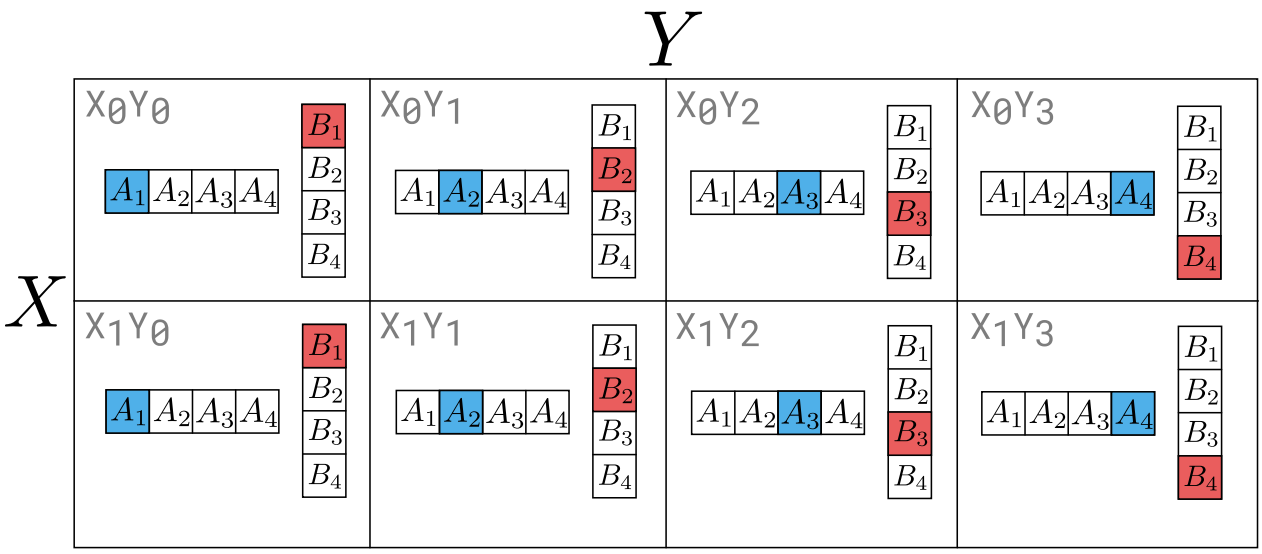
Each of these terms is the pairwise dot product of a **subset** of A and B's vectors' feature dimensions; $C_1 = A_1B_1$ is the matrix holding the pairwise dot products of the vectors' first four dimensions, $C_2 = A_2B_2$ is the second four, and so on. Summing the four C_i matrices, we have the pairwise dot products computed along *all* the vector dimensions.

This result implies something curious: to multiply a 4×16 and 16×4 matrix, we don't *need* to do it in one go; we can just do 4×4 multiplies and add the results. If we did these multiplies on different devices, that's a neat way of speeding up the multiplication, and that's the idea at the

heart of tensor parallelism. There's two possible cases here: when the sharding axes "match", and when they "mismatch", both of which will appear in the full neural network we'll examine.

4.4.1.1 Case 1A: Mesh-axes match

Since we have a 2×4 device mesh, one way to partition A and B onto this mesh is the following way:



In technical terms, A's column dimensions and B's row dimensions are *sharded* along the `Y` axis. We could write our sharding config now as:

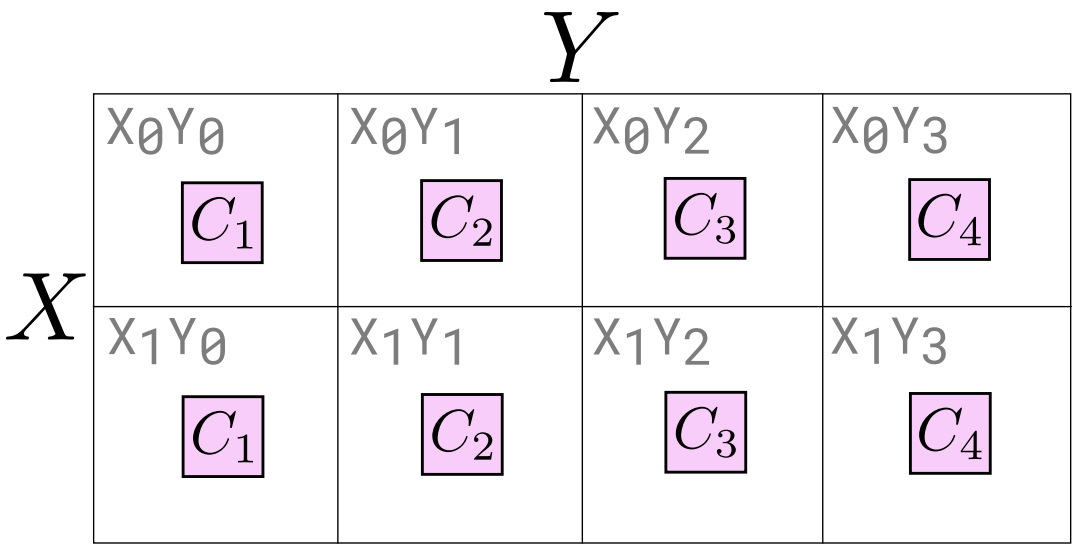
- A: `(full, sharded_Y)`
- B: `(sharded_Y, full)`

This config concisely captures how the matrices are divided across this mesh. For instance, on the four devices where `X=0` (the top row), we have all of A's rows (first axis), and the second shard of A's columns (the second axis; here, the second 4 dimensions). A full copy of A then is *distributed* across `Y=0,1,2,3`.

However, there's duplication: since we're *not* sharding on `X`, it means there are two full groupings with `Y=0,1,2,3`, *each* with a full copy of A. The same reasoning applies to B (as we see in the diagram), but with rows and columns reversed.

4.4.1.1.1 Multiply

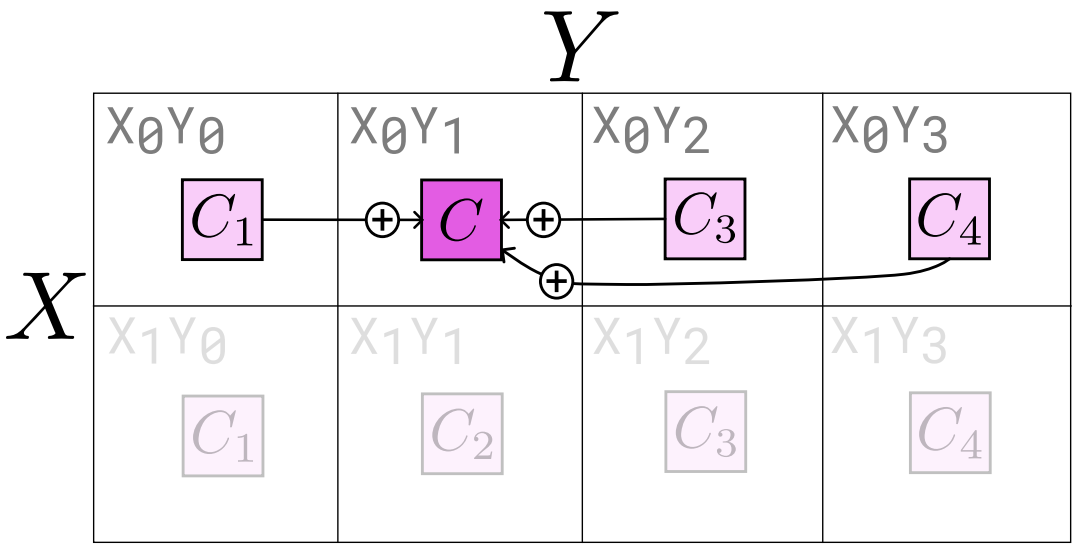
Now, each device multiplies the shard of A and B, to produce part of the multiplied value C. For example, the device at `X=0, Y=1` on the mesh produces $C_2 = A_2 B_2$.



Note that the duplication over the `X` axis results in both duplicate memory use *and* computation: for instance, both `X=0, Y=0` and `X=1, Y=0` are performing the same calculation.

4.4.1.1.2 AllReduce

All these matmuls are only fragments of C: they have the same *shape* as C, but each only contains pairwise dot products of *parts* of A and B's vectors. To calculate the full C, the devices can *"pull"* the other necessary values from its neighbors and *then add them* to its own calculation. These collective ops are well known, and this specific one is called AllReduce[5]. Let's visualize this from `X=0, Y=1`'s perspective (but remember, for now and the rest of this post, in actuality collective ops are performed by **all devices** *at the same time*):



[5]You may also recognize this as just `jax.lax.psum`, which it is!

To recap, our starting scenario was one where each device had to perform this $(4 \times 16) \times (16 \times 4)$ matrix multiply to get a copy of the final result C. With sharding, each device only had to do one $(4 \times 4) \times (4 \times 4)$ matrix multiply, and sum together the partial results from its neighbors.

And here's the real lesson: this "shard $\rightarrow$ multiply $\rightarrow$ allreduce" strategy scales to far larger matrices! If in the self-attention block of a Transformer we had Q and K matrices of shape 512×4096 each (512 vectors, each of length 4096), then getting the "match" between every pair of queries and keys QK^T is a $(512 \times 4096) \times (4096 \times 512)$ matrix multiply.

With this strategy, by splitting the inner axes along the mesh `Y` axis's 4 devices, each device in the group of 4 only has to do a $(512 \times 1024) \times (1024 \times 512)$ multiply (that is, pairwise dot products with just 1024 of 4096 dimensions each), which is about 4x faster[6].

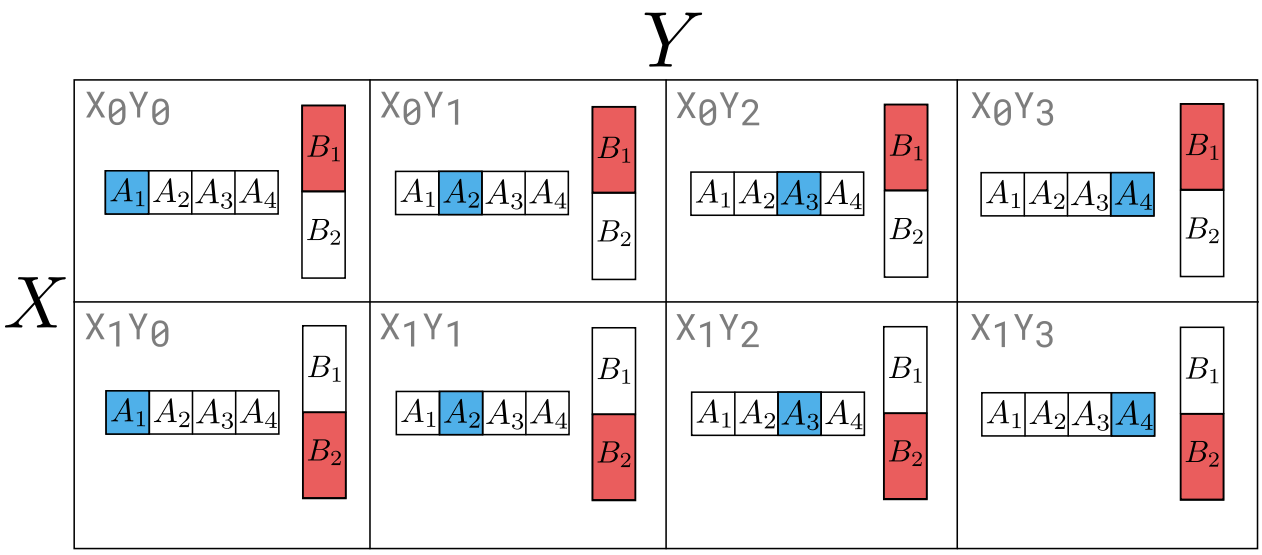
But we're still duplicating work along the `X` axis; we'll see shortly how with Case 2, this can be this 8x faster, leveraging all 8 devices.

4.4.1.2 Case 1B: Mesh-axes mismatch

Let's try the above problem (each device obtains a full copy of C), with a modified sharding config:

- A: `(full, sharded_Y)`
- B: `(sharded_X, full)`

That is, we now shard B's first axis on `X`, *not* `Y`. Visualized, the mesh now looks like this:

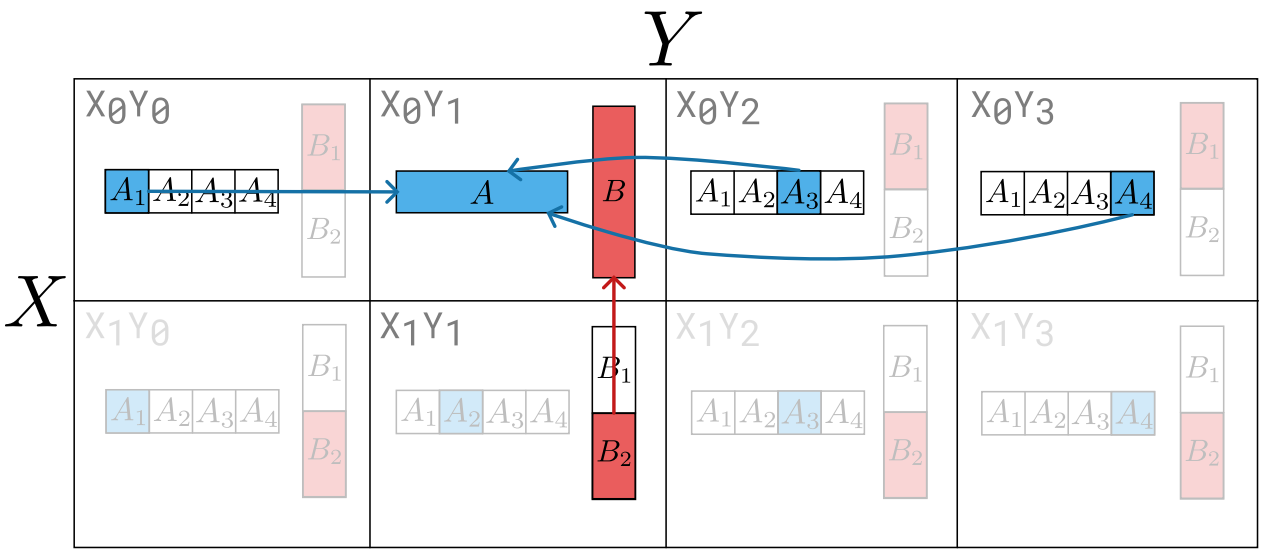


[6]Of course, *not* accounting for the communication overhead of the AllReduce. For systems that are compute-bound, splitting the actual multiplication across multiple devices and syncing them together (due to the $O(N^3)$ compute costs of a matrix multiply vs the $O(N^2)$ memory transfer) is still often worth it.

This doesn't appear to be a very useful mesh: we now have 4 copies of B over the mesh, compared to the previous two. Worse, because the sharding axes mismatch, we can't actually multiply: A_1 is of shape 4×4 (since 16 divided on 4 devices), and B_1 is of shape 8×4 (16 divided on 2 devices). That is, their number of inner dimensions don't match: each device has twice as many dimensions of B's (column) vectors than they have of A's (row) vectors.

This inability to multiply will be true in general: assume we start off with a valid matmul (inner axes match in number of dimensions), we're dividing this inner axis length by the number of devices on the corresponding mesh axis, and that may not be the same. Here, A does $\frac{16}{4} = 4$ due to the four devices on `Y`, while B does $\frac{16}{2} = 8$ due to the two devices on `X`.

But this is fine: we can use another collective op called AllGather, to restore the full matrices on each device. Visualizing this on `X=0, Y=1` (and remembering that in reality all devices are doing this *at the same time*):



Once each device restores a full copy of A and B, they multiply it as normal to directly get C. Now, this may seem odd: why'd we shard the matrix, only to gather it (then multiply) in full on every single device? (that is, no compute benefit, since *every* device does the same calculation).

Because it turns out when we combine Case 1B with Case 2, we can quite cleanly eliminate this duplicate computation. Let's see how.

4.4.2 Case 2: Outer Axes

We've focused on sharding on the "inner axes" of a matrix multiply so far, but there's nothing stopping us from sharding the outer axes; if anything, it's easier! Building on prior examples, let's try this sharding pattern:

- A: `(sharded_X, sharded_Y)`

- B: `(sharded_X, full)`

Visualizing, we have the following:

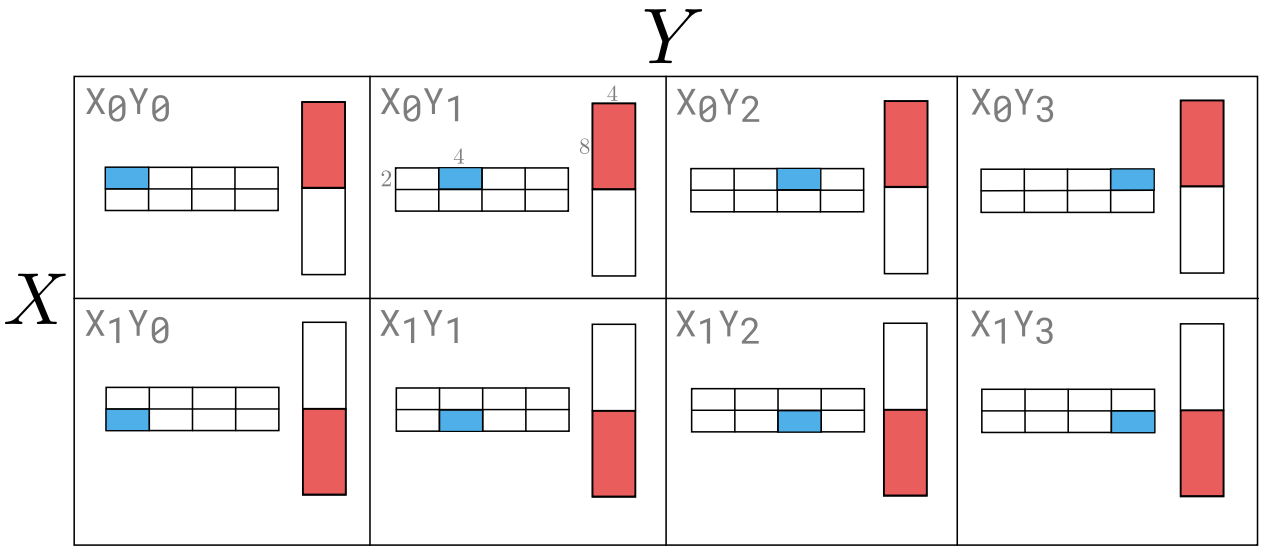
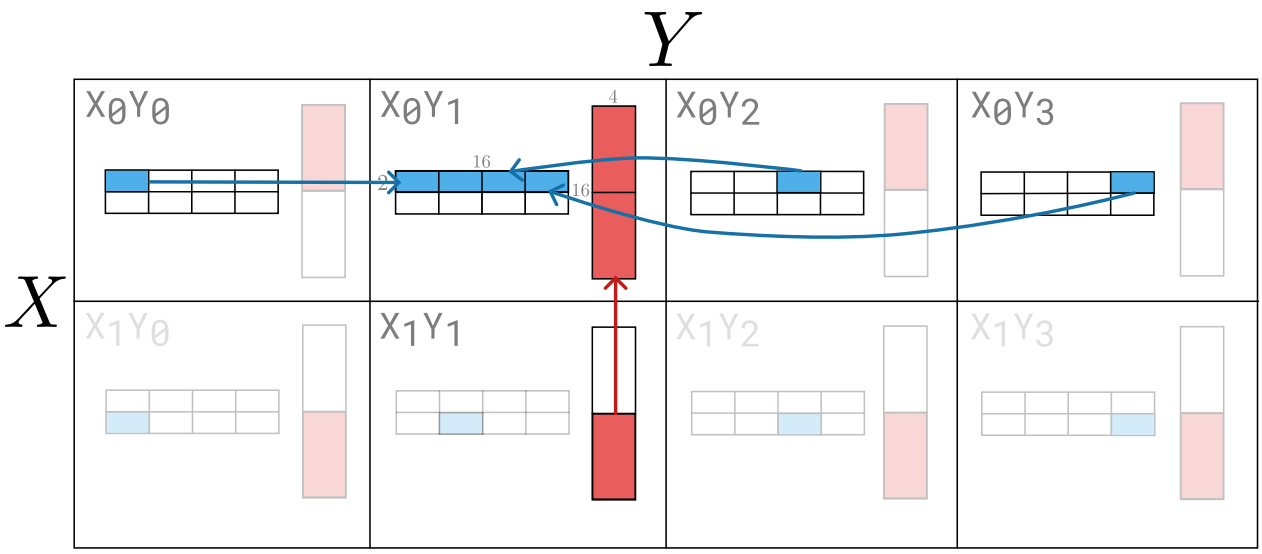


Figure 4.1: Since there's a *lot* of partitions this point on, it's easier to keep track of them by keeping track of their "colored in" areas than naming them individually.

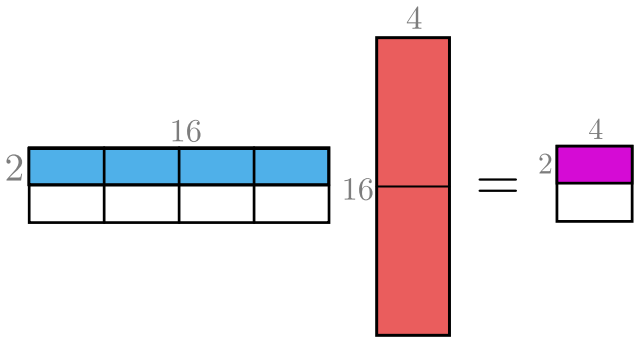
This is still the same as case 1B, with one change: `A`'s outer axis is now sharded on `X`. This means that devices with `X=0` contain shards of the first 2 (of 4) row vectors, whereas devices where `X=1` contain shards of the second 2 (of 4) vectors. The *feature dimensions* of these vectors remain sharded over `Y`.[7] We now only have *one* full copy of A, fully distributed over the entire 2×4 mesh.

We now proceed as previously with case 1B, allgathering A and B on their inner axes. Visualizing from the perspective of `X=0,Y=1`:

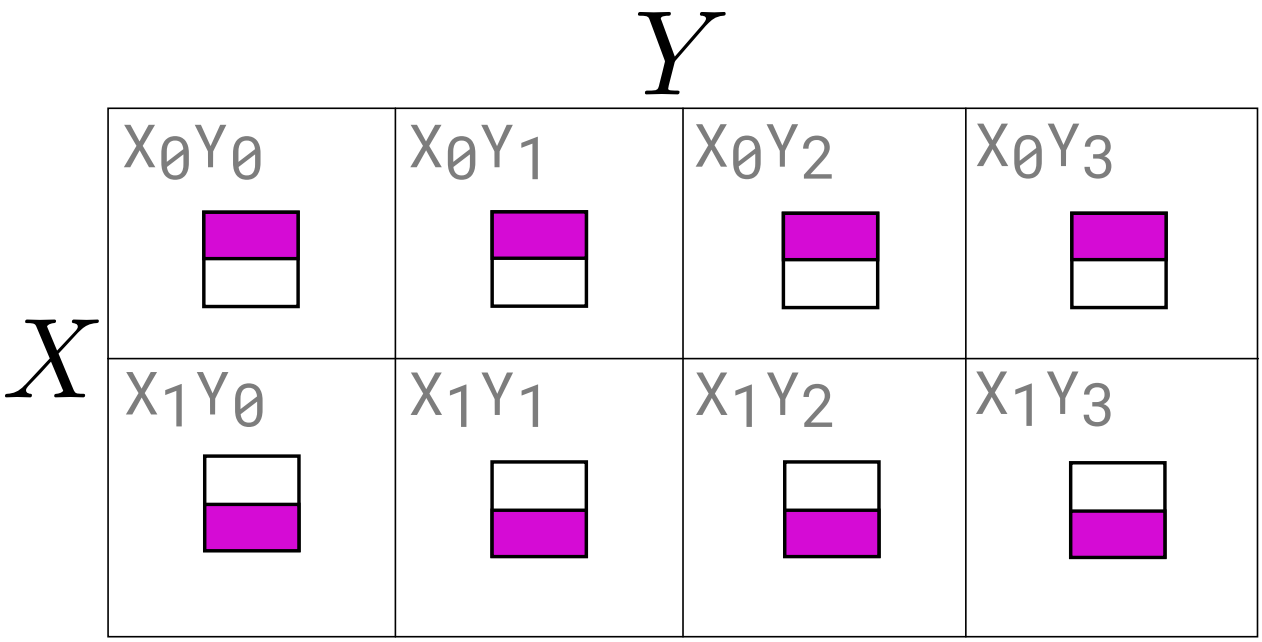
[7] Again, the *outer* axes in a matrix multiply are individual vectors; the *inner* axes are the feature dimensions of those vectors, as we saw in the intro. Feel free to scroll back if you need to recap!



Now we multiply the *first* 2 (of 4) row vectors of A with the 4 column vectors of B. To store these pairwise dot products, we end up creating a 2×4 matrix, that is, the top half of the (final) values of C:



Once all 8 devices do their respective allgathers and multiplies, we have:



This should feel familiar! If A is a batch of data (what we'd typically call X), with individual training examples as row vectors, then this is just batch parallelism: the two "submeshes" (corresponding to `X=0` and `X=1`) are each processing *half* of this batch, and then store the half of C that corresponds to the shard of A it processed.

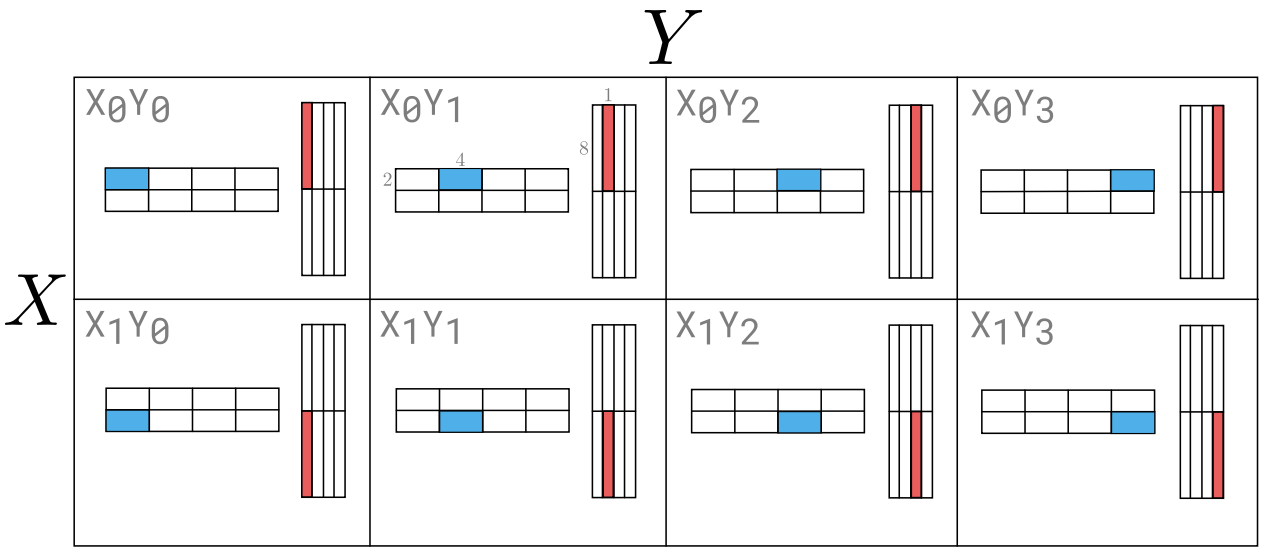
But there's *still* duplicate computation going on: all four devices in the submesh (`X=0` or `X=1`) are doing the same work!

4.4.3 Full Sharding

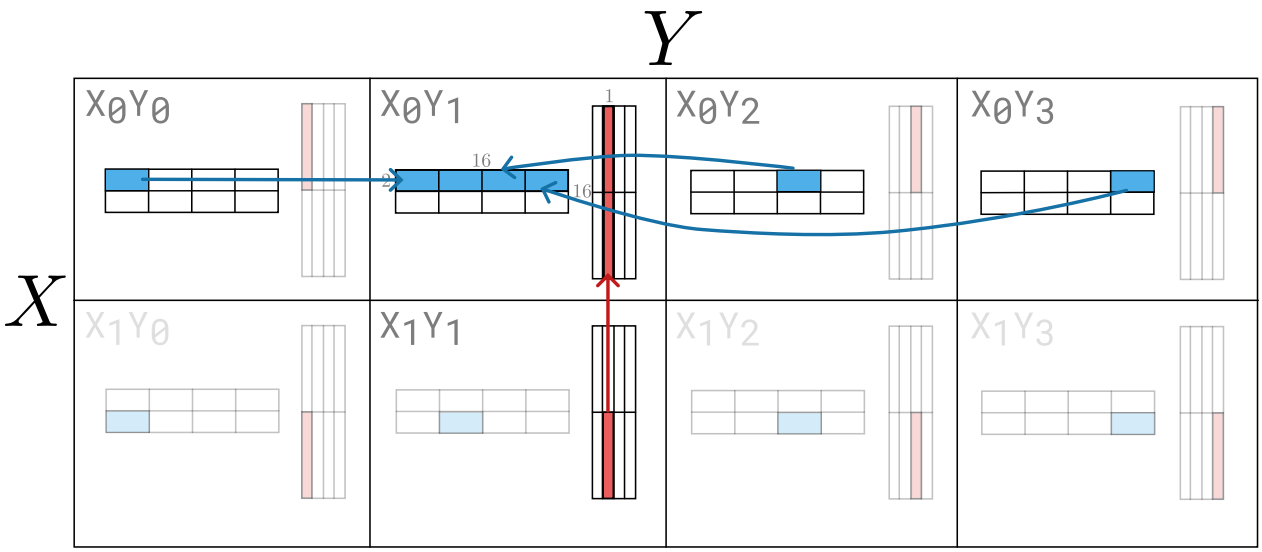
What if we also sharded B along its outer axis? That is, we shard both X and Y over `X` and `Y`, with the following sharding pattern:

- A: `(sharded_X, sharded_Y)`
- B: `(sharded_X, sharded_Y)`

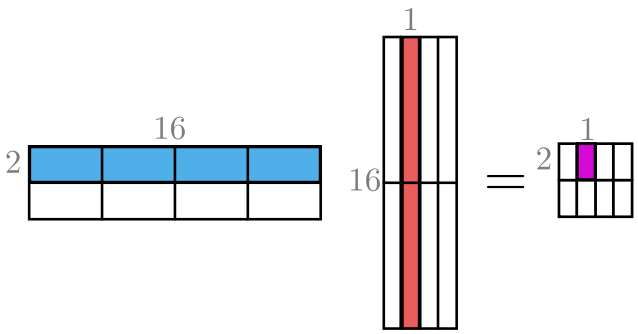
Visualized, we see that we only have one copy of A and B distributed over the whole mesh:



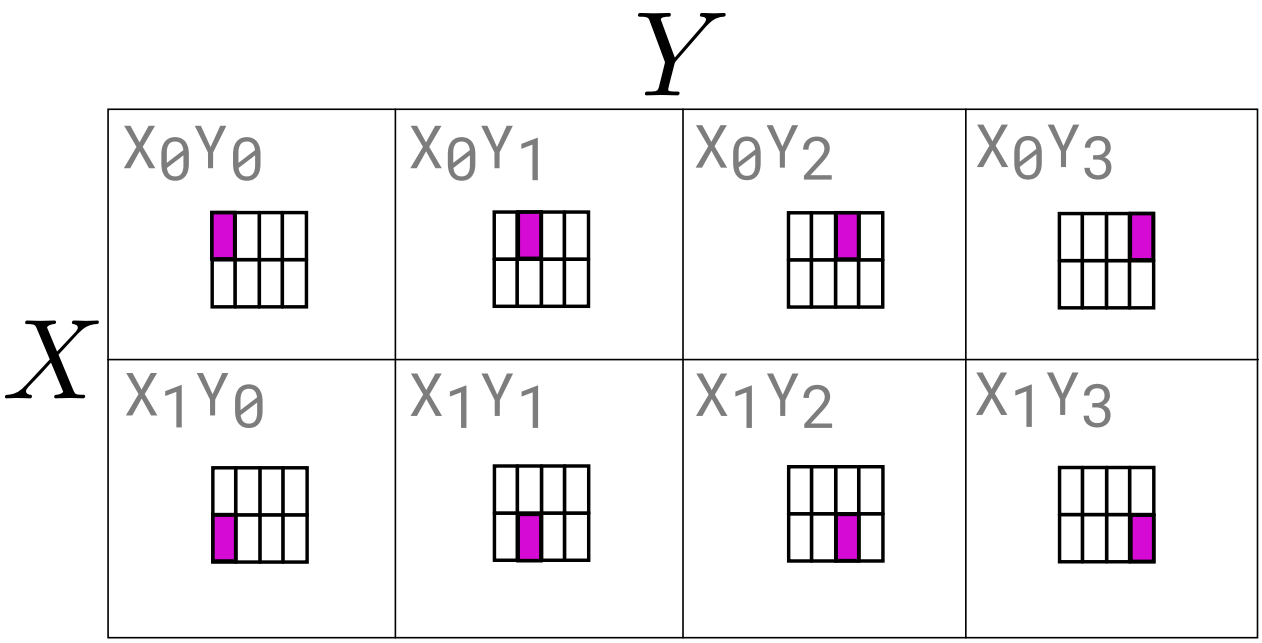
Since this sharding spec produces a mismatch along the inner axes, using case 1B, we do an all gather. However, this time as we've also sharded *both* the outer axes, `X=0,Y=1` will only be processing the full pairwise dot products of the first shard (first 2 row vectors) of A, with the second shard (the 2nd column vector) of B.



And then, a multiply, resulting in a 2×1 shard of the final output:



Zooming out, we see all 8 devices have computed a unique part of the output. Even more wonderfully, C has the same sharding pattern as A and B, `(sharded_X, sharded_Y)`:



4.5 Sharding: GSPMD-style

The GSPMD[8] paper (Xu et al. 2021) introduced an optimal sharding pattern for Transformers, subsequently used to train PaLM (Chowdhery et al. 2022). To reinforce our understanding of the two cases above, we'll now look at the sharding spec for the feedforward network in the Transformer layer. But before that, one final detour:

[8]GSPMD is the parallelization system that enables `pjit`, implemented as an extension to XLA (JAX's compiler).

4.5.1 What *is* XW?

Deep learning models at their core are straightforward: a number of layers stacked on top of each other. The core building block is the fully connected layer, often written as $\sigma(XW + b)$. It's really three operations:

- XW[9] is a matrix multiply, where X is a matrix of datapoints as row vectors, and W is a weight matrix.
- b is a vector of biases we add to XW.
- σ is a non-linearity of our choice, often `ReLU` but more recently enhanced variants such as `GeLU` in Transformer models.

We know what it *means* to shard the data X along its outer axis over `X`: it's just splitting the batch into yet smaller batches, standard data parallelism. But what does it mean to shard W along its outer axis, over `Y`? To understand that, let's dive deeper: what *is* XW?

Well, it's just a storage mechanism for the dot products of X's row vectors, and W's column vectors. We can directly reuse the math at the start of this post, replacing A and B with X and W:

$$X = \begin{bmatrix} 1 & 0 & 2 & -1 \\ 2 & 1 & 0 & -2 \end{bmatrix} \quad W = \begin{bmatrix} 0 & -1 \\ 1 & 2 \\ 2 & 0 \\ 0 & 2 \end{bmatrix} \quad XW = \begin{bmatrix} 4 & -3 \\ 1 & -4 \end{bmatrix}$$

The first row of the output matrix, stores the dot products of the first row vector of X with *all* of the column vectors of W. Since the dot product calculates how much two vectors "match" [10], the *new* output row vector (of the XW) stores how much the *original* row vector (of X) matches with each of the W weight vectors. Since we have two weight vectors, the output row vectors have two feature dimensions; if we have 20,000 weight vectors, they'd have 20,000 feature dimensions!

This output, once passed through a nonlinearity are the *new* data vectors (the "activations"), which will then be multiplied with the next layer's weight matrix to produce yet *newer* activations. This really is what we mean by stacking layers on top of each other: using the feature matching scores of one layer as the data to the next layer.

Answering the question, to shard W's outer axis along `Y` is to ask each device with a given `Y` value to compute the dot products with a *subset* of W's column vectors. We can do this since the dot product of a data vector with a weight vector *doesn't* depend on any other weight vectors. This means on a 2×4 mesh with `Y=0,1,2,3`, devices with `Y=0` can calculate dot products with the first $1/4$th of the weight vectors, devices with `Y=1` the second $1/4$th, and so on. This is exactly what we see in the calculation of C in the previous section (again, replace A and B with X and W)

[9] Note that in math contexts this is often written as Wx, where x is a *column* vector. In Python, data points are stored as *row* vectors, which means X *has* to be on the left hand side (because the feature dimension needs to be the inner dimension, the one we take dot products over!).

[10] Roughly speaking. The dot product is sensitive to the *magnitude* of the vectors too, not just their direction. If a particular weight vector is *very* large, it will have large dot products even if the data vector isn't that similar. However, with proper regularization, most weight vectors should be within a "reasonable" range, enabling comparison.

4.5.2 GSPMD's sharding spec

To recap, in the original Transformer (and most variants since), "Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network." (A. Huang et al. 2022) Let's look at the GSPMD paper's proposed sharding spec for the second sub-layer, the feedforward network (FFN):[11]

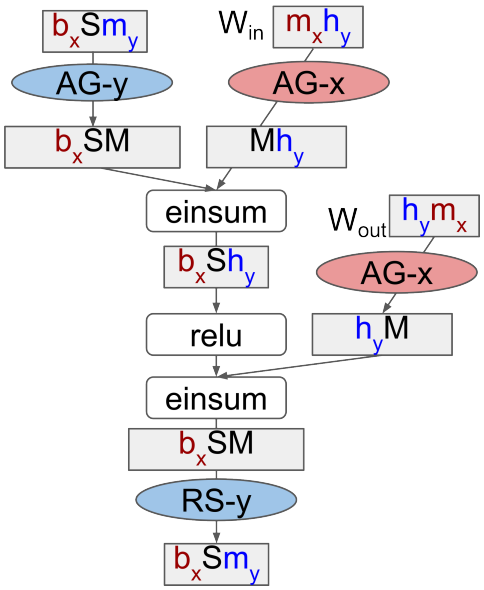


Figure 4.2: Fig. 7 from the GSPMD paper, modified with data X on the left (weight W_{in} on the right) to emphasize the XW_{in} matmul

The FFN is made of two fully-connected layers (the $\sigma(XW + b)$ we just discussed). Breaking this down:

- The first multiplies a weight (W_{in}, sharded as $m_x h_y$) to transform the input features (the *original* embeddings, sharded as $b_x S m_y$) into the features of the "hidden layer" (sharded as $b_x S h_y$). In most implementations, the hidden layer has 4x the number of features of the input/outputs of the FFN.
- The second multiplies a weight (W_{out}, sharded as $h_y m_x$) to transform the features of the "hidden layer" (sharded as $b_x S h_y$) into the output features (the *new* embeddings, sharded as $b_x S m_y$). The *number* of output features are the same as the input features, but the features *themselves* are different!

Let's zoom in on this, looking at each of these multiplies (the `einsums` in the diagram) separately:

[11] This is specifically the "2D finalized" sharding pattern presented in Table 1.

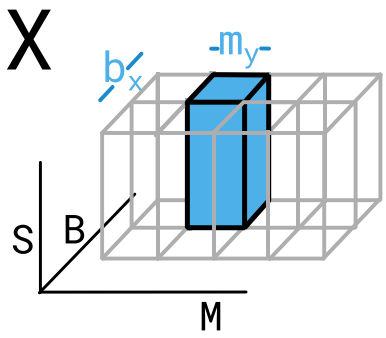
4.5.2.1 Embed → Hidden

To make things concrete, let's say we have 8 sequences, each of length 512, and an embedding dimension of 5120. That is, the shape of X is $(B \times S \times M) = (8 \times 512 \times 5120)$. Since we want 4x the number of features for the hidden layer, we'll need 20480 weight vectors of length equalling the embedding dimension 5120; hence W_{in} is of shape $(M \times H) = (5120 \times 20480)$.

The sharding spec for the data X and weights W, in the concise form in the diagram are $b_x S m_y$, and $m_x h_y$. We can rewrite this more verbosely as:

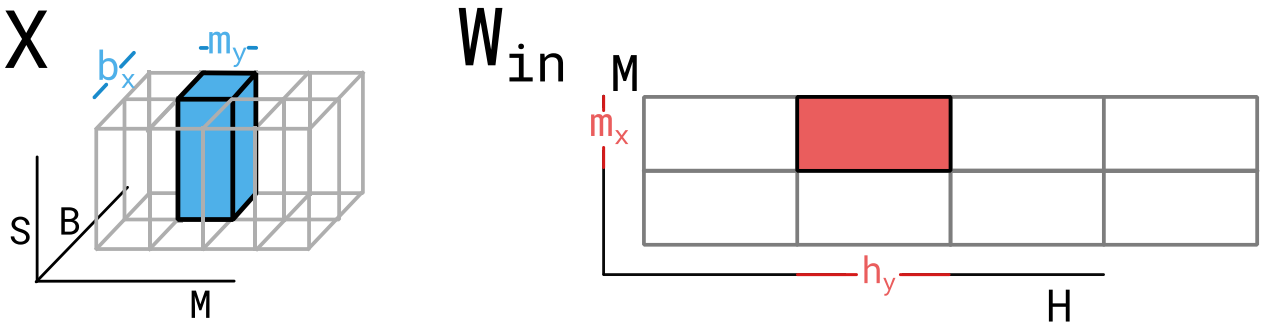
- X: `(shard_X, full, shard_Y)  # (batch, seq_len, embed)`
- W: `(shard_X, shard_Y)  # (embed, hidden)`

Note that we *don't* shard along the sequence length axis. This means, for instance, device `X=0, Y=1` has:

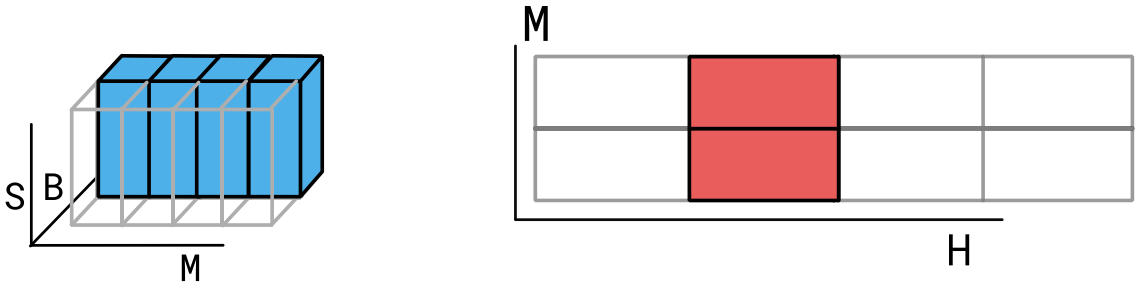


- the second of four slices (the second 1280 of 5120) along the embedding dimension
- of *all* the timesteps
- of the first of two slices (the first 4 of 8) along the batch dimension.

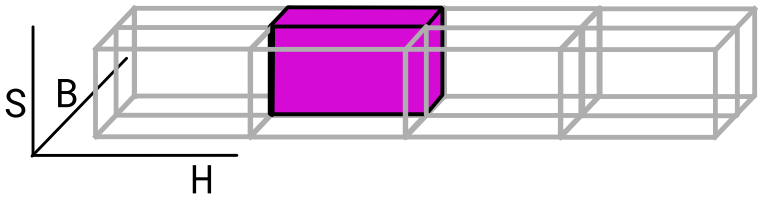
This sharding pattern should look familiar: this is Case 2 (since the outer axes are sharded) combined with Case 1B (since the inner axes are sharded on mismatched mesh axes), again! Visualizing again from the perspective of `X=0, Y=1`:



As previously in Case 1B, we allgather the first matrix (here, X) along `Y`, and the second matrix (here, W_{in}) along `X`. Once allgathered, both matrices have their inner axes = $M = 5120$ dimensions.



We leave the outer axes sharded: this means we're only multiplying the embeddings of the first 4 (of 8) sequences in the batch. We're multiplying *all* the input dimensions, but only computing the second (of four slices) of the hidden dimensions, that is, dimensions `(5120, 10239)` of 20480 dimensions.

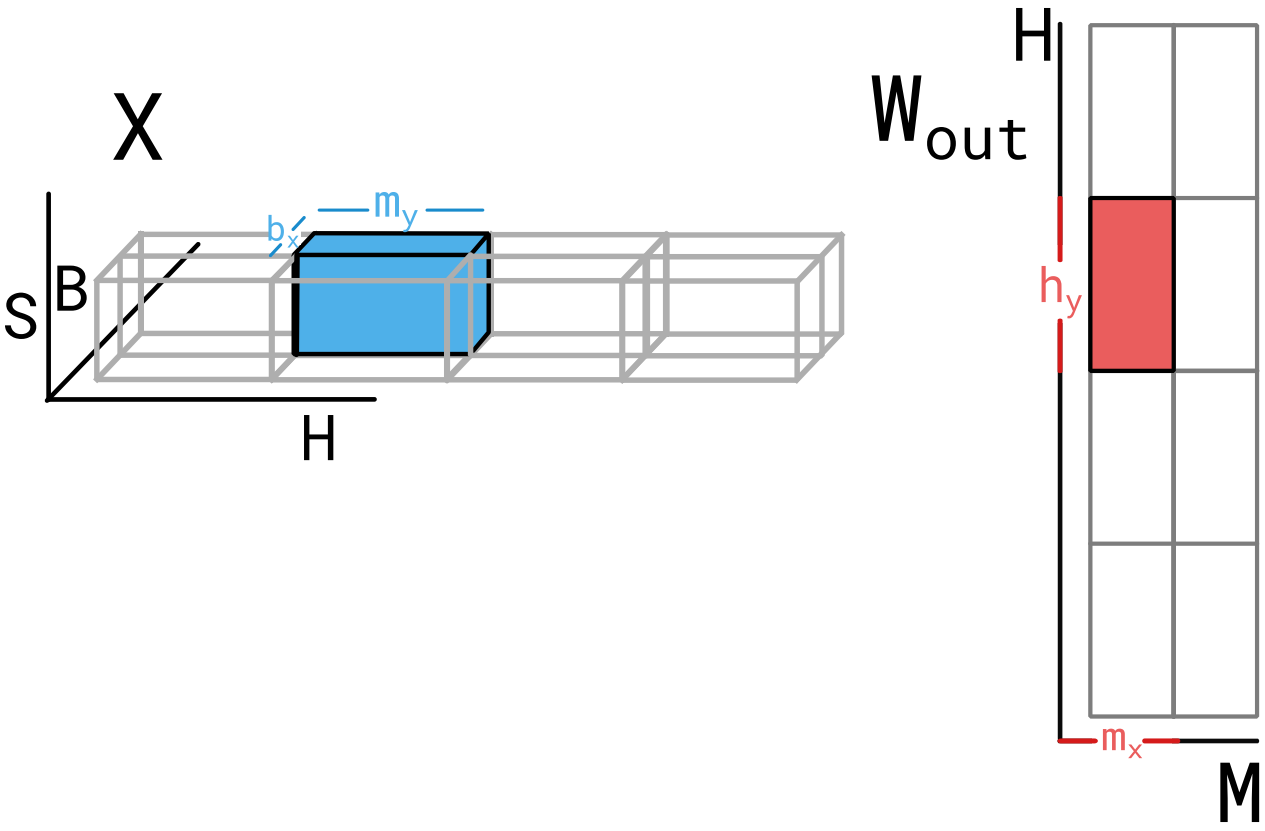


All 8 devices then compute a unique shard of the output (corresponding to a different batch slice along `X`, and hidden dimension slice along `Y`), successfully avoiding any duplications.

4.5.2.2 Hidden $\rightarrow$ Embed

So far, it seems like we've only been using Case 1B and Case 2. But here, the sharding spec for the data and weight, in the concise form are $b_x S h_y$ and $h_y m_x$. In verbose form:

- X: `(shard_X, full, shard_Y)  # (batch, seq_len, hidden)`
- W: `(shard_Y, shard_X)  # (hidden, embed)`



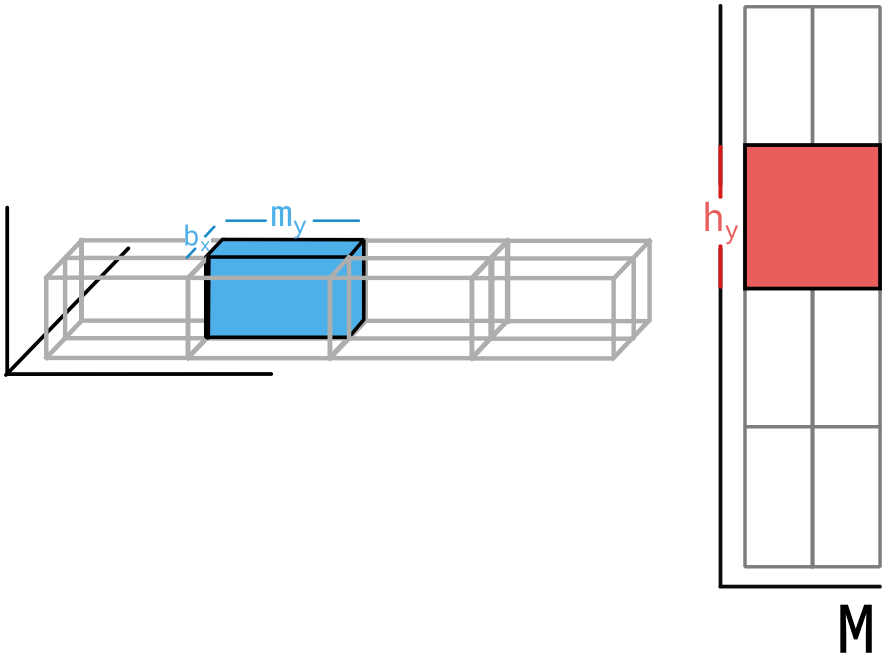
That is, the inner axes are sharded on the same mesh axis, which is case 1A. This means we can directly multiply on-device (without having to allgather over the inner axes), and then run allreduce to get the final output. There is one problem though: when *both* matrices are fully sharded over a 2D mesh, and the mesh axes of their *inner* axes match, the mesh axes of their *outer* axes also match.

This means if we multiply the shards directly, we'll have the opposite problem of duplicate work: incomplete work. Specifically, since both the outer axis of X (batch) and W_{out} (embed) are sharded on `X`, for the first batch slice on the `X=0` submesh we'll only compute the dot product with the *first half* of the weight vectors (on `X=1`, the dot products of the second batch slice with the *second half* of the weight vectors).

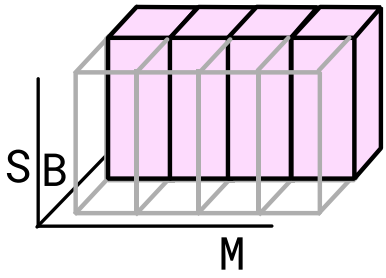
We don't have any devices computing the dot products of the first batch slice with the second half of weight vectors (and vice versa)!

To fix this, we need to allgather either X or W across their *outer* axis. Generally, we prefer to keep the batch axis sharded, so we allgather along W's outer axis. This means each device now computes:

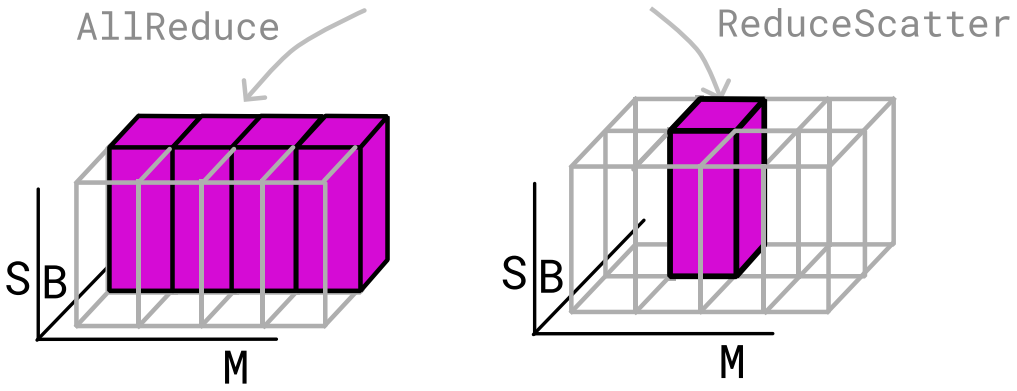
- the dot product over 1/4th of the hidden dimensions
- for 1/2 of the batch
- for *all* of the weight vectors.



Then multiplying, each device along `X=0` produces *part* of the dot products for *all* the embedding dimensions (case 1A), for the first batch shard (case 2):



At this point, we have two options: we can either proceed with an allreduce, as previously covered in case 1A. *Or*, we can notice what we *want* is to produce an output tensor with the same sharding pattern as the input to the entire FFN subnetwork. In this case, we can use a different collective op called reduce-scatter. Unlike allreduce, which sums up values across all devices for the entire shard, this sums up values only for the smaller part of the shard we want:



4.5.2.3 Wrapping it up

We did it! As we see, Case 2, combined with Case 1A or 1B as appropriate allowed us to stack two fully sharded, fully connected layers on top of each other, *and* produce an output with the exact same sharding as the input (no resharding needed!).

We won't cover the sharding of the self-attention sublayer here, but you should be able to combine the two cases here, with Table 1 in the GSPMD paper to work out how the attention heads are sharded over the `Y` axis[12].

Now that we have a deep understanding of how to shard a neural network, let's write some code! We're in luck here: I'm currently working on a port of the ESM2 (Lin et al. 2022) protein language model into Flax (Heek et al. 2020), so much of the examples will be directly pulled from the working codebase.

For context: the model built here is a BERT (Devlin et al. 2019) style, encoder-only Transformer. It is a 15B param model with 48 layers, and 5120 dimensional embeddings as seen in the previous section. Each encoder layer has two sublayers as previously described: a self-attention sublayer followed by a feedforward network sublayer. Let's start by understanding `pjit`'s programming model, then using it to progressively build the full model.

4.6 The `pjit` programming model

A good way to think of `pjit` is a supercharged `jax.pmap`. If you recall, `pmap` runs the same program on multiple devices, each with a different shard of *input data* over the *batch* axis. `pjit` is more flexible: it allows us to shard both the data and weights (and when training, even the optimizer states) in whatever configuration we please over a mesh. To do so, `pjit` requires three things from us:

- A mesh specification, mapping the "logical" devices on the 2D (or higher-D) mesh to the *physical* devices available.

[12] Or in the math textbook way, "this is left as an exercise to the reader". But really, I think working it out would be a neat exercise!

- The sharding spec of all tensors being passed as *input* to, and returned as *output* to from the function.
- Sharding constraints for select intermediate tensors inside the function. This isn't *strictly* necessary (XLA GSPMD will try to find a viable layout), but can lead to improved memory usage.

Note what isn't here: JAX doesn't need us to insert any of the collective ops we discussed. It uses a constraint based model, where we specify sharding constraints for the "big", memory intensive tensors, and it automatically determines the sharding pattern for all other intermediate tensors in the function, as well as any collective ops that need to be inserted to meet these constraints.

4.6.1 Sharding constraints

So how do we specify a sharding constraint? Well, we write our function as we normally would, and then call `with_sharding_constraint` on the tensor we'd like to place the constraint on:

```
from flax.linen import partitioning as nn_partitioning

def forward(x):
    # ...run some code here.
    # note x here has shape [batch x seq_len x embed]
    x = nn_partitioning.with_sharding_constraint(x, ("batch", None, "embed"))
    # ...continue more code here.
    return x
```

When *not* inside a `pjit`, this is a no-op, returning the original input unchanged. Inside a `pjit`, XLA will insert any collective ops needed to meet the constraints we specified in the second argument.

The second argument here specifies the mesh axis to shard each of the axes of this rank-3 tensor over. The `None` in the middle means "do not shard over the sequence length axis". But note that the other two don't have `X` or `Y`, but "names" (`batch` and `embed`). This is because at runtime, we'll use a set of rules *mapping* these from names *to* the mesh axes. For instance, here's the one I use for the ESM2 models on a 2×4 mesh:

```
# Create 2D TPU mesh
DEFAULT_TPU_RULES = [
    ("batch", "X"),
    ("hidden", "Y"),
    ("heads", "Y"),
    ("embed_kernel", "X"),
    ("embed", "Y"),
]
```

Under these rules, `("batch", None, "embed")` will become `("X", None, "Y")`. Writing our code this way keeps it flexible as we potentially try out different sharding configurations.

4.6.1.1 A first run

Then, at runtime there's three things we need to do:

1. Create the `mesh`: the `mesh` is the object that translates our abstract, 2D mesh to the actual physical hardware that will be running the computation. This is where we specify, say, which of the 8 physical devices becomes the mesh device `X=0,Y=1`.

```
from jax.experimental import maps

mesh_shape = (2, 4)
# We reshape the devices into a 2D mesh, and name the mesh axes.
devices = np.asarray(jax.devices()).reshape(*mesh_shape)
mesh = maps.Mesh(devices, ("X", "Y"))
```

2. `pjit` the function: This is similar to `jax.jit`, except we now also need to specify the sharding spec for the input and output.

- The input to this transformer is, say, a 8×512 matrix of integers, and we shard the first (batch) axis along the mesh `X` axis.
- The output is 5120 dimensional vectors, so the output is a matrix of shape $8 \times 512 \times 5120$. We keep the batch axis sharded over `X`, and the embedding axis sharded over `Y`, which we define as `P("X", None, "Y")`. Note that `pjit` is lower level than the `flax` methods, and needs the mesh axes directly inside a `PartitionSpec`; the translation rules we define above are only used for the constraints *inside* the `nn.Module`. You can read more about `pjit` at the JAX level here.

```
from jax.experimental import pjit, PartitionSpec as P

# Create fn for inference.
pjit_forward = pjit.pjit(
    forward,
    in_axis_resources=P("X", None),
    out_axis_resources=P("X", None, "Y"),
)
```

3. Call the function: we use two context managers, one to activate the mesh, and the other specifying the translation rules for all the sharding constraints inside the function.

```
with maps.Mesh(mesh.devices, mesh.axis_names), nn_partitioning.axis_rules(
    DEFAULT_TPU_RULES
):
    x = pjit_forward(x)
```

Let's now apply these constraints to the weights and activations of the feedforward network.

4.7 Applying constraints to a FFN

In a previous section, we looked at the sharding spec the GSPMD paper proposed for the FFN in a transformer layer. To summarize in a table[13]:

Tensor	**Sharding Spec `[shape]`**
activation: embedding	$X, \_, Y$ `[batch, seq_len, embed]`
W_{in}	X, Y `[embed, hidden]`
activation: hidden	$X, \_, Y$ `[batch, seq_len, hidden]`
W_{out}	Y, X `[hidden, embed]`

This sharding spec is for any generic, dense Transformer. The code below is the second sublayer (the FFN network) of an encoder layer in the ESM2 model. We apply this sharding spec to the weights on lines 11 and 21, and to the activations on lines 13 and 23:

```
1  # ... we apply a layer norm and multi-head attention before this.
2
3  # Create second residual block (LayerNorm + MLP)
4  residual = x
5  x = nn.LayerNorm(name="final_layer_norm", epsilon=1e-5)(x)
6
7  # Create + apply first MLP layer with weight + activation sharding constraints.
8  x = partitioning.Dense(
9      self.ffn_embed_dim,
10     name="fc1",
11     shard_axes={"kernel": ("embed_kernel", "hidden")},
12 )(x)
13 x = nn_partitioning.with_sharding_constraint(x, ("batch", None, "hidden"))
14 # Don't approximate gelu to avoid divergence with original PyTorch.
15 x = nn.gelu(x, approximate=False)
16
17 # Create + apply second MLP layer with weight + activation sharding constraints.
18 x = partitioning.Dense(
19     self.embed_dim,
20     name="fc2",
21     shard_axes={"kernel": ("hidden", "embed_kernel")},
22 )(x)
23 x = nn_partitioning.with_sharding_constraint(x, ("batch", None, "embed"))
24 x = residual + x
25
26 return x
```

[13]The full sharding spec for both the FFN and self-attention can be found in Table 1 in the GSPMD paper. PaLM uses this same spec, but over a much larger 256×12 mesh per pod. (Sec. 4)

The activation sharding specs are applied as in the initial example: we just `with_sharding_constraint`. But there's two new things:

- There's a new `shard_axes` argument being passed into the layer definition on lines 11 and 21.
- We're using the `partitioning.Dense` layer instead of the standard `nn.Dense`.

Let me elaborate on what's going on here.

4.7.1 Sharding constraints: Weights

Here's the goal: we want to apply the same `with_sharding_constraint` function we used on the activation tensors, to the weight tensors in these `Dense` layers. Problem is, it's defined as a local variable *inside* the `__call__` method of `nn.Dense`; and is not an attribute I can access from the outside. There's two options here:

- First is the `t5x` (Roberts et al. 2022) route of creating a new version of `Dense` directly with the constraint applied inside the `__call__` method. This works in a robust, well-tested library! However, we'd need to make a modified copy of *every* layer where we want to apply a constraint over a weight.
- The second approach is noticing that all `flax.linen` layers use the `nn.Module`'s (their parent class) `.param` method to create their params. Then, we can write a simple mix-in class that *overrides* the default param method to apply the sharding right as it is created:

```
@dataclasses.dataclass
class ShardMixIn:
    """Adds parameter sharding constraints for any flax.linen Module.
    This is a mix-in class that overrides the `param` method of the
    original Module, to selectively add sharding constraints as specified
    in `shard_axes`"""

    shard_axes: Optional[Mapping[str, Tuple[str, ...]]] = None

    # Modifies off
    # https://github.com/google/flax/blob/main/flax/linen/partitioning.py#L304
    def param(self, name: str, *init_args):
        # Initialize using the original Module's `param` method
        param = super().param(name, *init_args)

        # If `shard_axes` specified and param name in the dict, apply constraint
        if self.shard_axes and (name in self.shard_axes.keys()):
            axes = self.shard_axes[name]

            # Apply the sharding constraint (e.g. axes=('embedding', 'hidden'))
            param = nn_partitioning.with_sharding_constraint(param, axes)

```

```
            # Sow this, to have the AxisMetadata available at initialization.
            self.sow(
                "params_axes",
                f"{name}_axes",
                nn_partitioning.AxisMetadata(axes),
                reduce_fn=nn_partitioning._param_with_axes_sow_reduce_fn,
            )

        return param
```

There's only 10 lines of code here, which add the `shard_axes` argument to *any* Flax `nn.Module`. As we can see, `with_sharding_constraint` is applied on line 21, *only* when a given param has a constraint specified. No need to rewrite the original layer definition, we simply create a new version that inherits the original, and our mix-in:

```
class Dense(ShardMixIn, nn.Dense):
    pass
```

This is just *one* solution to be able to apply `with_sharding_constraint` to the weights, and I'm definitely quite open to feedback on whether this is a sensible strategy!

4.7.2 Putting it all together

We omitted a key detail in the opening example: in the real forward pass (the `.apply` method) we need to pass in *both* `esm_sharded_params`, and the data `batch`. Since the params are an input argument, they will also need a sharding spec. The `params` in Flax are a `PyTree` (specifically, a nested dict) and so the sharding spec is a nested dict with the same structure. There's some plumbing here, so let's go through it step by step:

Because the `ShardMixIn` `.sow`'s the sharding metadata into the module, this metadata is available at model initialization with the `.init` method. Let's initialize the 15B model, and inspect the shapes of the parameters of layer 42:

```
import functools
import flax.linen as nn
import jax
import jax.numpy as jnp

from esmjax.modules import modules
from esmjax.modules import partitioning


embed_dim = 5120
num_heads = 40
```

```
num_layers = 48

embedding = nn.Embed(33, embed_dim)
block_fn = functools.partial(modules.EncoderLayer, num_heads, embed_dim, embed_dim * 4)
esm2 = modules.ESM2(embedding, block_fn, num_layers)

key = jax.random.PRNGKey(0)
arr = jnp.array([[0, 1, 2]])
```

We can see that the 5120-dimensional embeddings are projected to produce embeddings for 40 heads, with 128 dims each.

```
# jax.eval_shape replaces all actual arrays with ShapeDtypeStruct
# This avoids memory use, *and* allows us to inspect the param shapes.
params = jax.eval_shape(esm2.init, key, arr)
params['params']['42']['self_attn']
```

We can also see the axis metadata generated when calling the `.init` method:

```
params['params_axes']['42']['self_attn']
```

Only the params that we've specified a sharding constraint over exist in this `PyTree`. To pass into `pjit`, we use a utility function to convert the names into mesh axes, and replicate the structure of the full params. The `AxisMetadata` are replaced with proper `PartitionSpecs`, and all other params have their sharding pattern set to `None`, meaning full replication.[14]

```
params, params_axes = params.pop("params_axes")
esm_axes = partitioning.get_params_axes(params,
        params_axes,
        rules=partitioning.DEFAULT_TPU_RULES)
esm_axes['params']['42']['self_attn']
```

We now pass this sharding spec (`esm_axes`) into the `pjit` definition. Then, we have a fully sharded inference method, distributing the computation work of this 15B model across all 8 cores of a TPU. You can find a fully runnable notebook here.

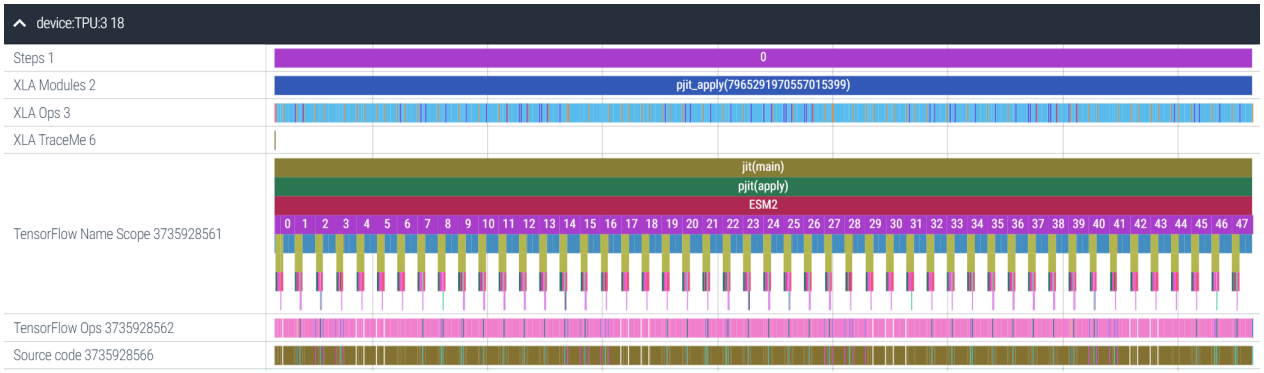
```
apply_fn = pjit.pjit(
    esm.apply,
    in_axis_resources=(esm_axes, P("X", None)),
    out_axis_resources=P("X", None, "Y"),
)
```

[14] On a larger model, we'd shard even the biases and layer norms, but on this scale it's fine not to. They're a *lot* smaller than the weights.

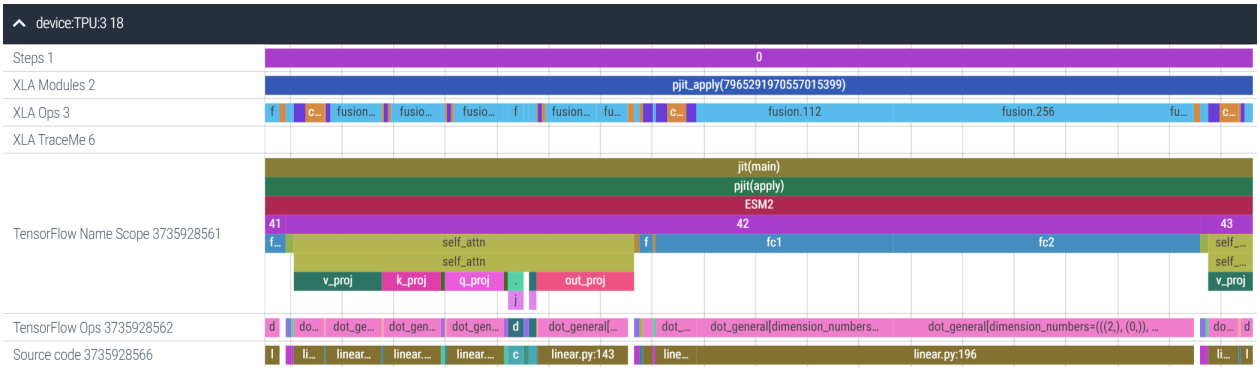
```
with maps.Mesh(mesh.devices, mesh.axis_names), nn_partitioning.axis_rules(
    partitioning.DEFAULT_TPU_RULES
):
    embeds = apply_fn(esm_sharded_params, batch)
```

4.8 Program Trace

Upto now, we've handwaved the fact that there's a *lot* of communication going on in this forward pass. How much time on the forward pass are we spending on these collective communication ops? The short answer: on a TPUv2-8, about 20%[15]. The way to answer this is a program trace, and JAX makes this easy: here's the full trace of all 48 layers of the ESM2 15B model on TPU3 (of 0 to 7) of a TPUv2-8, taking about 4.25s to complete inference with a batch size of 32:

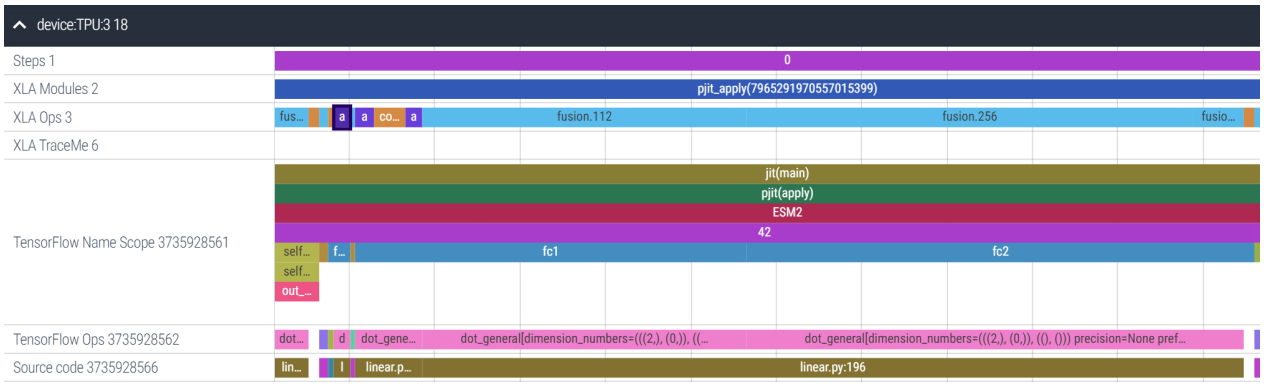


Let's zoom in on layer 42. We can tell from this trace that the FFN sublayer (fc1 and fc2) takes more time to execute than the self-attention sublayer. On the top bar we have the `XLA Ops`, the direct, device-level ops being executed. Most of these are `fusion` ops, a combination of fused multiplies and elementwise ops (e.g. addition, subtraction):



[15] A quick estimate counting *both* the communication ops (e.g. allgather, fused reduce-scatters) as well as data formatting ops.

Let's zoom in more into the FFN sublayer. Amusingly, XLA has decided to allgather `fc2`'s weight matrix (the selected purple box) *before* the matmuls of `fc1` and `fc2`. This is the power of JIT-compilation: XLA is able to re-order operations as needed for better performance. It's also inserted a reduce-scatter over the results (the rightmost, blue fusion op). Overall, the FFN sublayer takes 54ms, and 8ms are spent on collective communication and data reformatting ops, about ~15%[16]:



In summary, for a ~20% performance tradeoff, we can now run inference with only *one* copy of the model stored across all our devices! I want you to explore this trace yourself, so here's the link to Perfetto; just hit `Open trace file` and upload this trace, and go play around!

4.9 Conclusion: Beyond Tensor Parallelism

Tensor parallelism is powerful, allowing us to scale from 1 GPU/TPU to all 8 connected GPU/TPUs, and when using larger slices of a TPU pod, even further (PaLM was trained using just tensor parallelism, on two full TPUv4 pods with 3072 chips each, Sec. 4). There's three concluding thoughts I'd like to leave you with:

- **Pipeline parallelism:** Given the large volume of communication involved, tensor parallelism is only viable when there is fast (ideally 1TB/s+) interconnect between devices. This is true for TPUs all the way up to an entire pod; however, GPUs only have fast interconnect (e.g. NVLink) in groups of 8 on a single host. *Between* hosts, the interconnect is slower (e.g. commercial cloud is typically on the order of ~100GB/s [17]), meaning a different strategy is necessary.

  If tensor parallelism is slicing a model "horizontally" (each layer is sharded across all devices), pipeline parallelism is slicing it "vertically" (device 1 can hold layer 0,1 device 2 holds layers 2,3, and so on). The only communication is when activations move *between* layers, not inside a layer. The problem is that it leads to "bubbles" where devices are inactive (Y. Huang et al.

[16] It's worse in the self-attention sublayer (29%), which also takes less time overall, resulting in an average of 20%. Would be a better layer to focus more for improvement!

[17] Although, newer offerings such as the p4d.24xlarge or BM.GPU4.8 have *considerably* better inter-node bandwidth. At the same time, the A100 GPUs themselves are much faster, which means the inter-node bandwidth must keep up just to *avoid* becoming a bottleneck.

2019). Large GPU clusters tend to use tensor parallelism for all 8 GPUs connected on a single host, and pipeline parallelism between hosts to make the best of both strategies.

- **More automation:** `pjit` is incredibly flexible, capable of accommodating any sharding pattern we can come up with. The GSPMD paper covers even more "strange" cases such as sharding convolutions over spatial axes, across multiple devices. However, we still do need to specify a sharding pattern, which for non-experts can be challenging. There's a lot of exciting work going on in frameworks such as Alpa (Zheng et al. 2022), which automate this entirely, and I'm excited to see where this line of research is headed.
- **Larger models?** Scaling language models has been a strategy that continues to work with no clear signs of slowing down. But a substantial fraction of scaling up goes into learning *factual knowledge* about the world[18] than the semantics of language. Retrieval-augmented models such as RETRO (Borgeaud et al. 2022) and Atlas (Izacard et al. 2022) are *much* smaller (the largest RETRO model is only 7.5B params). However, they introduce a new axis (retrieval time) to the current trio of compute, memory use and intra/inter-host communication, and I'm curious to learn where the bottlenecks will arise as this strategy is scaled up.

4.9.1 Further Reading

If you'd like to keep learning more about parallelism at scale, here's a couple places to help you get started:

- *How to Train Really Large Models on Many GPUs?* (Weng 2021): This is a great blog post providing a "big picture" overview of the multiple types of parallelism possible on GPU clusters, as well as other memory saving strategies.
- *Scalable Training of Language Models using JAX pjit and TPUv4* (Yoo et al. 2022): A technical report from Cohere detailing how they use `pjit`, data and tensor parallelism to scale their training on TPUv4 pods (without needing pipeline parallelism)
- *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism* (Shoeybi et al. 2019): This paper explored sharding the data and weights *strictly* on the outer axes (Case 2 only), motivated by a need to minimize inter-device communication on large GPU clusters.
- *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM* (Narayanan et al. 2021): This is a follow-up paper, exploring large scale parallelism on GPU clusters with a fusion of tensor parallelism (Case 2 only) combined with pipeline parallelism.
- *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models* (Rajbhandari et al. 2020): This paper looked at the technical optimizations needed to store only one copy of a model across all devices (Zero-DP, Stage 3), finding this increases communication volume by only 1.5x over baseline data parallelism.
- *GSPMD: General and Scalable Parallelization for ML Computation Graphs* (Xu et al. 2021): We used the sharding spec introduced in this paper; the paper as whole discusses extensively about propagating user annotations across a computation graph, and the technical considerations involved.

[18] And factual knowledge can go stale: remember, a BERT model trained in 2019 would associate the word Corona more with beer than the pandemic!

5 Chinchilla's Implications

In a field where the "big ideas" seem to change on a weekly basis, Chinchilla (Hoffmann et al. 2022) is a standout paper: it came out a little over 18 months ago, and found then-LLMs to be massively undertrained compared to their model size, with the then dominant scaling laws (Kaplan et al. 2020) suggesting that on log-log scales, model size N be scaled ~3x (2.7x) faster than the dataset size D[1].

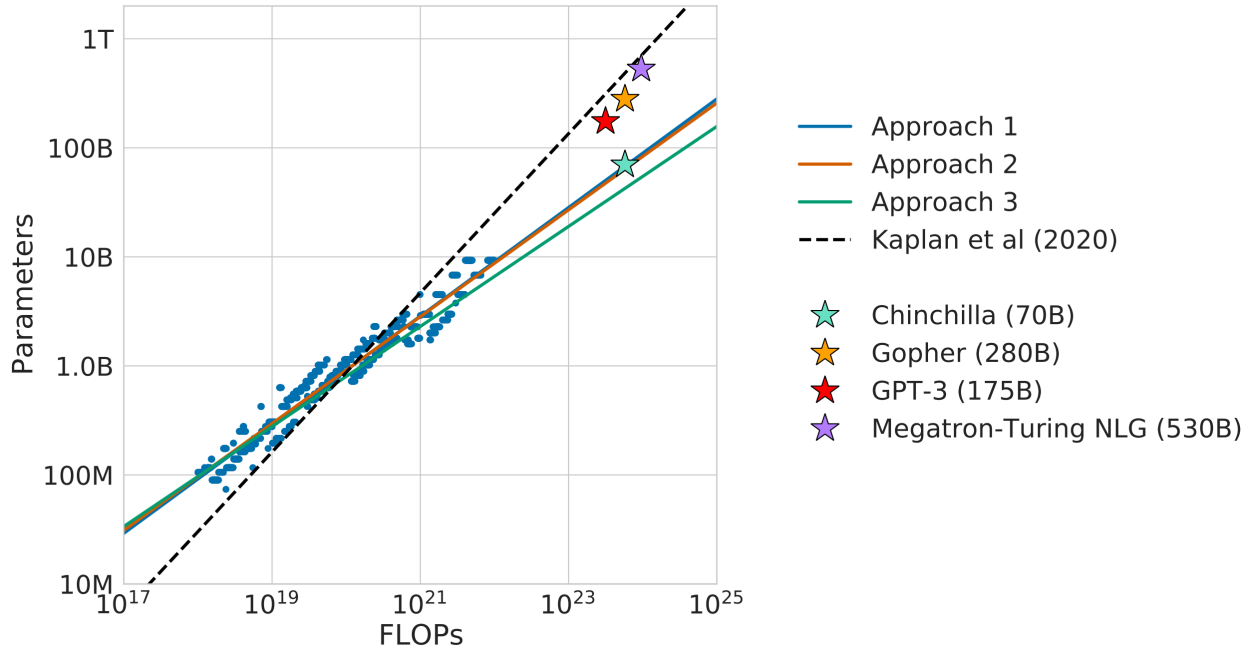


Figure 5.1: Pre-Chinchilla models (such as Gopher (Rae et al. 2022)) tended to use the 3:1 scaling implied by Kaplan et al. (2020), whereas post-Chinchilla models use 1:1 scaling. Fig 1 from Hoffmann et al. (2022).

Chinchilla, by accounting for the effect of the learning rate scheduler, proposed that model size and dataset size should in fact be scaled in a 1:1 ratio. This meant that the (then) largest 500B+ parameter models (such as PaLM 540B (Chowdhery et al. 2022) or MT-NLG 530B (Smith et al. 2022)) were substantially undertrained[2], and would need several orders of magnitude more compute than

[1]That is, $\log N = 2.7 \log D + K$, where K is some constant. Equivalently, $N \propto D^{2.7}$. In concrete terms, for every 2.7 orders of magnitude increase in model size N, we only need to increase dataset size D by one order of magnitude.
[2]That is, they needed to be trained on much more data for that size to be compute optimal.

was available for that size to be "optimal"[3]. This triggered an industry-wide shift towards smaller models trained for longer (for a given compute budget).

Let's dive deeper into the scaling laws, how they were derived, their implications for LLM training runs, how to (and not to) interpret them.

5.1 What's compute optimal?

Chinchilla's focus is on training models that are "compute optimal": in this context, creating a model with the lowest loss for a given, fixed amount of compute C. There's two words in the phrase "compute optimal", so let's examine them both.

5.1.1 Compute

Before beginning a training run, we can estimate the total number of FLOPs[4] (Floating Point Operations) a run will have available from just four factors:

- The FLOPs/sec[5] per chip.
- The number of chips.
- How long (in hours/days) we plan the run to be.
- The MFU (Model FLOPs/sec Utilization) of the run. MFU is the % of maximum FLOPs/sec your chips can actually use towards training your model.[6]

So, suppose I have access to 64 H100s for 1 week, and initial tests reveal my MFU is 50%. Then, the total FLOPs available during this run is:

$$\frac{989 \times 10^{12} \text{ FLOPs}}{\text{second} \cdot \text{H100}} \times 604800 \text{ seconds} \times 64 \text{ H100} \times 50\% = 1.91 \times 10^{22} \text{ FLOPs}$$

Assuming the chips cost 4$/hour, this is a 43K$ training run! That is a lot of money to be spending, even on a run quite small by LLM standards. You'd want this money spent optimally (under some definition of optimal), and this is where Chinchilla comes in.

[3] Said differently, for the total compute available to the teams then, they could've obtained a lower validation loss with a smaller model trained on more data.

[4] FLOPs measure the number of individual arithmetic operations on floating point numbers a larger computation (such as a matrix multiply) will use.

[5] FLOPs/sec is often written as FLOPS (with a capital S for second), but this can be confusing, so I write it out explicitly here.

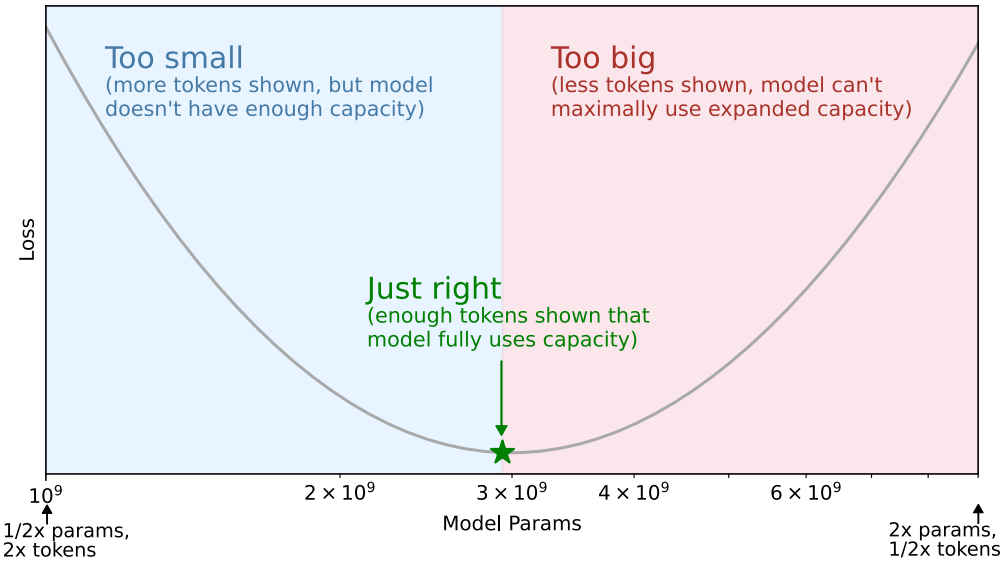
[6] In more depth, this is the FLOPs/sec used towards the computation of the training job itself (disregarding any re-computations such as activation checkpointing) divided by the peak FLOPs/sec of the hardware. Chowdhery et al. (2022) first introduced this, and this can be quickly calculated by running your job on the cluster for a few minutes (vs days/weeks for the full run).

5.1.2 Optimal

There are two factors that influence the amount of FLOPs a run uses: The size of the model N, and the number of tokens D. A commonly used approximation (introduced in (Kaplan et al. 2020)) to the number of FLOPs a training run uses (the cost C) is

$$C \approx 6ND$$

A derivation of this approximation can be found in Section C.2. Note the directly inverse relationship between model size and tokens used here: for a fixed FLOPs budget C, doubling the model size N means it can only "see" half as many tokens D during training[7]. There's a sweet spot to strike here: a model not so small it doesn't have enough capacity to learn from too many tokens, but not so large it barely sees enough tokens to make use of the added capacity.



Which N and D is best? The answer is in the scaling law literature (Kaplan et al. 2020, Sec 6.1): the "optimal" $(N_{\text{opt}}, D_{\text{opt}})$ are the ones that produce a model that achieves the lowest loss on a validation set of the pretraining data, subject to the fixed cost constraint (the green star above)[8].

Chinchilla's general approach then, is to compute $(N_{\text{opt}}, D_{\text{opt}})$ for a few small values of C, and use them to extrapolate $(N_{\text{opt}}, D_{\text{opt}})$ for the C equivalent to the full training run.

[7] Again, you can always double the model size and use the same number of tokens as before, but you now need 2x the compute! Chinchilla's analyses are about how to maximally use a *fixed* amount of compute.

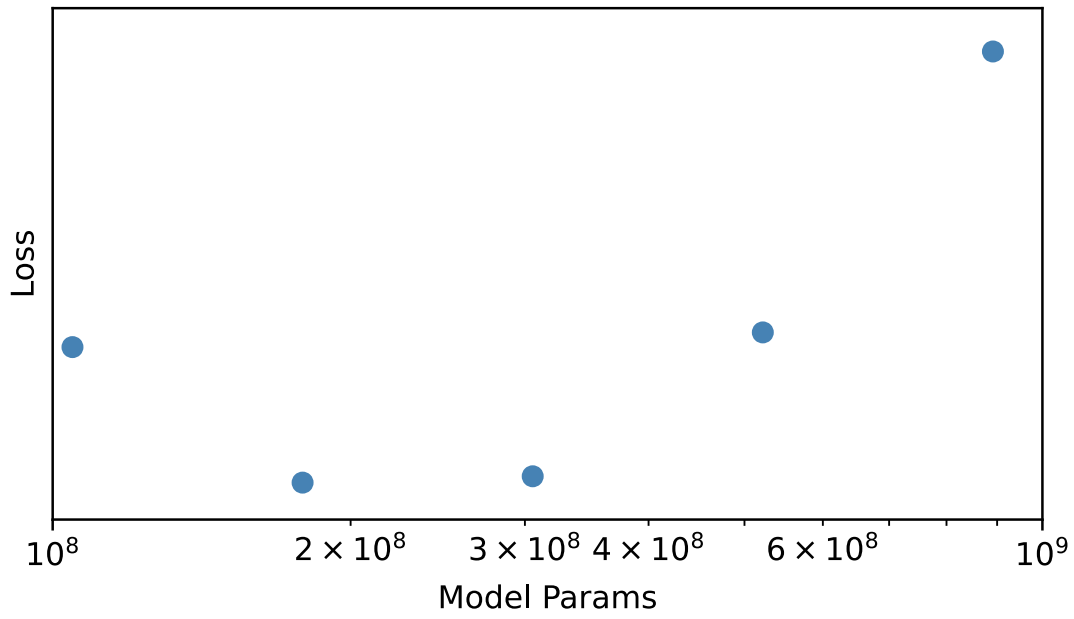
[8] A deeper dive on the findings of Kaplan et al. (2020), and how Chinchilla built on those findings can be found in Section C.1

5.2 Chinchilla Scaling

The full Chinchilla paper uses three different methods (fixed model size, fixed FLOPs and parametric fitting) to estimate the scaling behavior between model size and data, with similar results across all three. We focus on approach 2, where they use a set of 9 different FLOPs counts (from 6×10^{18} to 3×10^{21}). The method is as follows:

5.2.1 Calculating an IsoFLOP curve

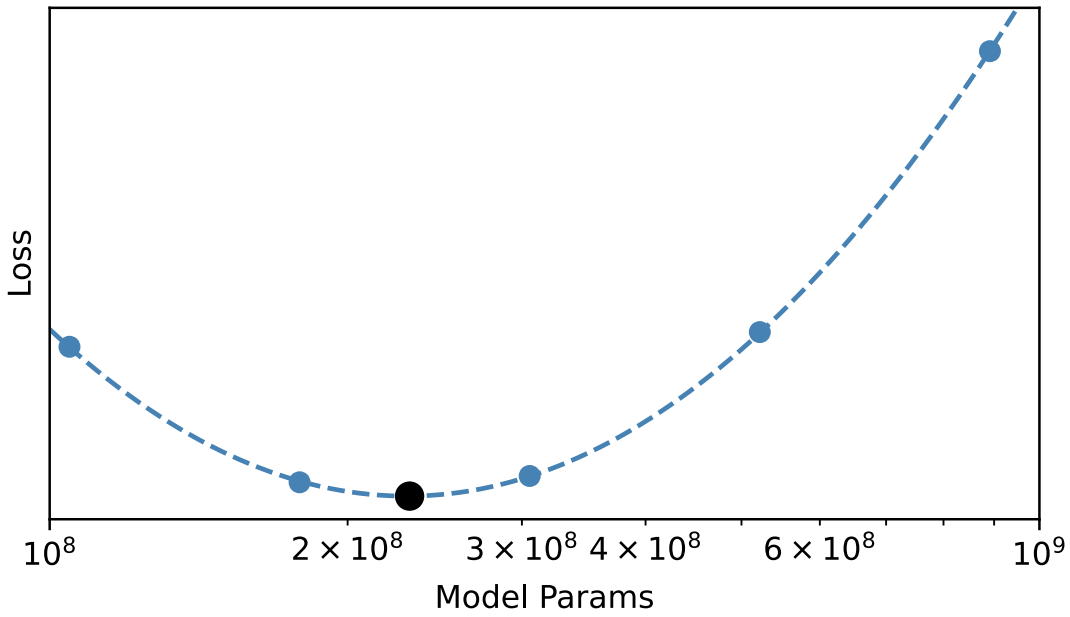
First, for a given C, train multiple models, varying N and D such that the FLOPs count remains C[9]. Compute the validation loss L of each model, producing a plot like this:



Then, fit a parabola to the points $(\log N, L)$. This allows us to predict the loss of each model of size N trained under the fixed amount of compute C. The authors call this an IsoFLOP curve[10], and it allows us to find N_{opt}, the model size with the lowest loss for that C.

[9] Note that the Chinchilla paper uses a more detailed approach to calculating C than Kaplan et al. (2020)'s $C \approx 6ND$ approximation, explained in Appendix F. The $C \approx 6ND$ approximation is within 10% across two orders of magnitude (Table A4), so it's still a good mental model!

[10] In that the FLOPs (C) is the quantity being held constant for each point on this curve.



This process is then repeated for each of the 9 values of C, resulting in the full IsoFLOP curves plot:

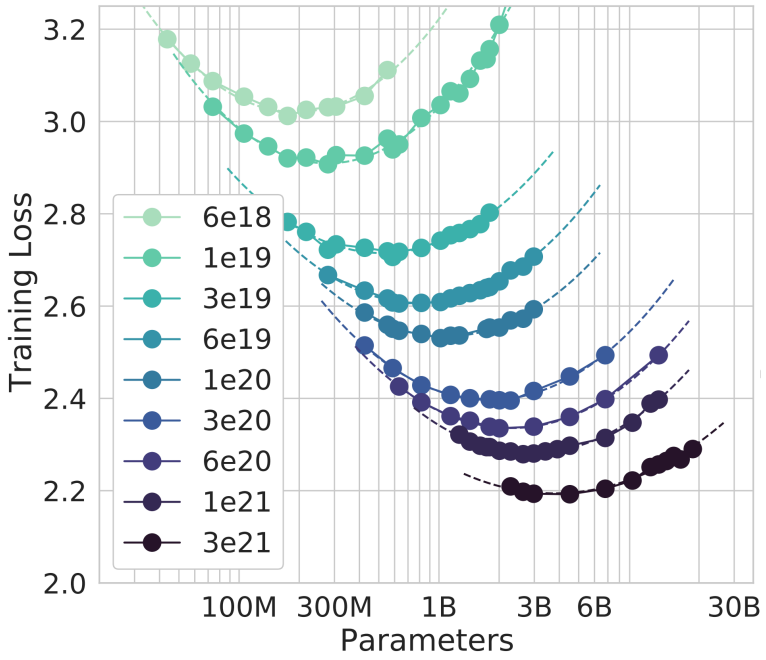


Figure 5.2: Figure 3 (left) from Hoffmann et al. (2022)

5.2.2 Model Scaling

Each of these 9 curves above have one value of N_{opt}. The authors then fit a power law $N_{\text{opt}} = AC^a$ to the points (C, N_{opt}). This is where the scaling law appears: when $a = 0.49 \approx 0.5$, they obtain a very tight fit to the empirically calculated N_{opt} values. This allows them to extrapolate the best model size (66B) for Chinchilla's full run of 5.76×10^{23} FLOPs, two orders of magnitude larger than the largest IsoFLOP curve of 3×10^{21} FLOPs.

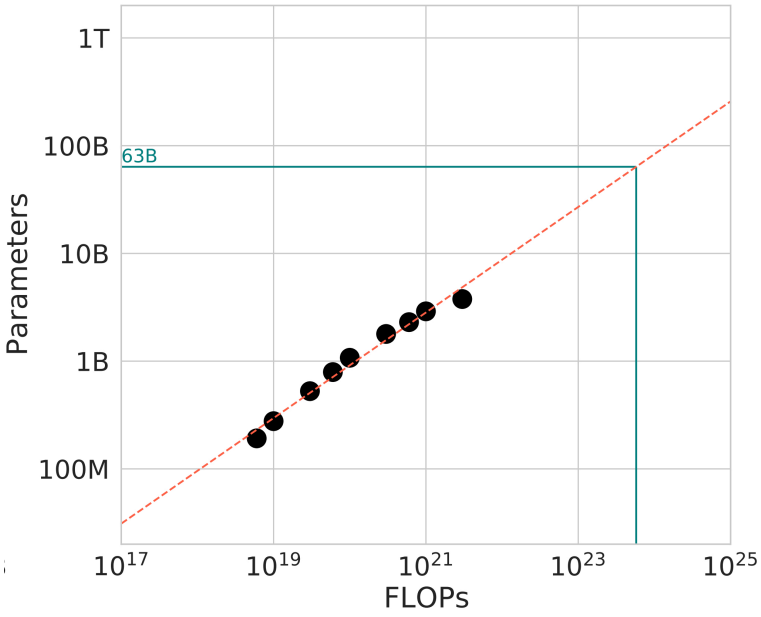


Figure 5.3: Figure 3 (center) from Hoffmann et al. (2022)

Extrapolating two orders of magnitude in FLOPs is quite the jump, but there's an exact reason $C = 5.76 \times 10^{23}$ is chosen for the full training run - the same amount of compute was used to train the preceding Gopher 280B (Rae et al. (2022)) model. According to the scaling laws calculated here, a compute optimal model for Gopher's compute budget should be 4x smaller, trained on 4x more tokens.

This prediction is tested empirically, and it holds, validating the scaling laws: Chinchilla 70B outperforms Gopher 280B on a suite of benchmarks, as detailed in Section 4.2 in the paper.

5.2.3 Data Scaling

With scaling, the discussion usually centers on how to scale the *model size* w.r.t. increasing compute. This is because N and D are not independent: for a fixed C, if you know N_{opt} you also know D_{opt}[11]. Fitting a similar power law, $D_{\text{opt}} = BC^b$, the authors obtain $b = 0.51 \approx 0.5$[12], which one should

[11] Intuitively, if you know the size of the model, and amount of compute used to train it, you can quickly reverse calculate how many tokens the model would need to be "trained on" to hit that compute budget.

[12] The actual 80% confidence intervals for a is $(0.462, 0.534)$, so it's not unreasonable to round it to 0.5 for simplicity.

see as an alternate view of the same finding above!

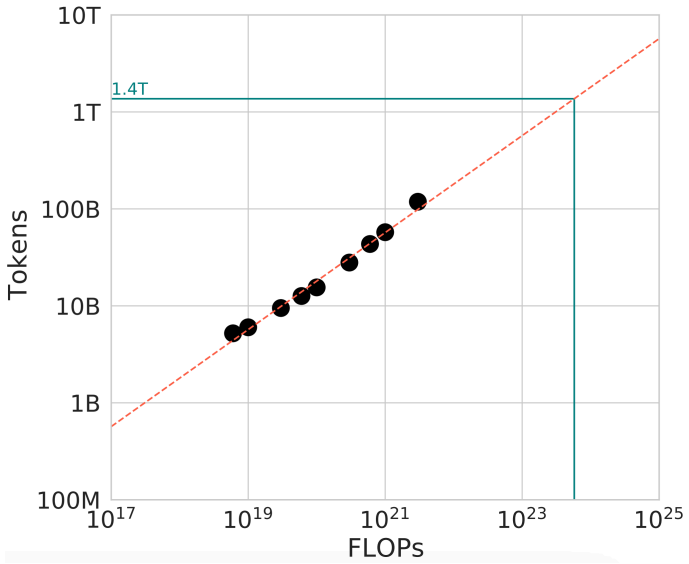


Figure 5.4: Figure 3 (right) from Hoffmann et al. (2022)

Doing it this way, and noticing both $a \approx 0.5$ and $b \approx 0.5$ does make the 1:1 ratio between model size scaling and data scaling clear.

5.2.4 Generality

The analysis in the core body of the paper takes place using the MassiveText dataset, a proprietary dataset also used to train Gopher. To validate the generality of these findings on other datasets, in Appendix C they reproduce this scaling behavior on two subsets of MassiveText, C4 (a public dataset first introduced in Raffel et al. (2020)) and GitHub code with 4 values of C. In both, they find the constant a linking C and N_{opt} to be ≈ 0.5:

Approach	Coef. a where $N_{opt} \propto C^a$	Coef. b where $D_{opt} \propto C^b$
C4	0.50	0.50
GitHub	0.53	0.47
Kaplan et al. (2020)	0.73	0.27

Figure 5.5: Table A2 from Hoffmann et al. (2022)

It is important to note that these estimates are not highly precise: the power-law is fitted on only

9 values of C in the main experiments[13] (and only 4 values of C for the GitHub/C4 experiments in Appendix C). The 80% confidence intervals for a using the IsoFLOP approach is $(0.462, 0.534)$, which is still rather wide! Moreover, while a power law fit works well, we don't know if the "true" functional form between C and N_{opt} *is* a power law[14].

Yet, despite not being highly precise, the 1:1 scaling suggested by the results do outperform the previous 3:1 scaling - Chinchilla 70B is 4x smaller than Gopher 280B, but trained on 4x more data it outperforms the larger model, highlighting the importance of data. That the measurements of a and b replicates across three approaches (Section 3), and that the IsoFLOP approach replicates on two more datasets (C4 and GitHub, producing estimates of a and b that are each closer to $(0.5, 0.5)$ than $(0.73, 0.27)$) provide further support that 1:1 scaling is an improvement generally.

5.2.4.1 Quadratic Compute

One intuitive (and important) conclusion from the 1:1 scaling of model size and data means, if you want a compute optimal model that's 2x large, you need to train it on 2x many tokens. And since FLOPs count is the product of both, this means you need 4x as much compute!

Continuing that reasoning, if you want a model 5x larger, you need 25x as much compute, and so on! $N_{\text{opt}} \propto C^{0.5}$ rewritten differently is $C \propto N_{\text{opt}}^2$, that is you need to scale compute *quadratically* with model size. This is enormously expensive, and is the core reason model sizes peaked around early-2022 (pre-Chinchilla): we're only just now doing training runs with C large enough that models of that size (500B+) are *compute optimal*, and future model size scaling will remain slower (compared to pre-Chinchilla) because of this quadratic factor.

5.3 Chinchilla in practice

Chinchilla, again, was an impactful paper that revealed just how wasteful training runs up till that point have been (see Timbers (2023) for historical overview). Its message is memorable and concise: scale data with model size in a 1:1 ratio.

This message does need to be interpreted with nuance! For instance, the paper comes with this table calculating the optimal number of params and tokens across a range of FLOPs values:

[13] Understandably so: computing even one of the IsoFLOP curves means training multiple models with a fair amount of compute each, which isn't cheap!

[14] On those lines, here's a neat GitHub implementation using PySR to directly regress scaling laws from data (*without* assuming a power law fit first)

Parameters	FLOPs	Tokens
400 Million	1.84e+19	7.7 Billion
1 Billion	1.20e+20	20.0 Billion
10 Billion	1.32e+22	219.5 Billion
67 Billion	6.88e+23	1.7 Trillion
175 Billion	4.54e+24	4.3 Trillion
280 Billion	1.18e+25	7.1 Trillion
520 Billion	4.19e+25	13.4 Trillion
1 Trillion	1.59e+26	26.5 Trillion
10 Trillion	1.75e+28	292.0 Trillion

Figure 5.6: Table A3 from Hoffmann et al. (2022)

But strictly speaking, this is only optimal for the exact dataset + model architecture used to compute the coefficients of the scaling law, that is, $N_{\text{opt}} = AC^a$ and $D_{\text{opt}} = BC^b$. The argument Chinchilla makes is that $a \approx 0.5$ and $b \approx 0.5$ generally, *across* datasets; it does not make any claims as to what general values of A and B are, and they can vary from dataset to dataset!

For instance, the final Chinchilla 70B model is trained on 1.4T tokens. If we had an aggressively deduplicated version of the MassiveText dataset (such as (Abbas et al. 2023)), it is possible to have a scaling law experiment that yields 1.0T tokens as optimal, while also staying consistent with $b \approx 0.5$, producing a plot that looks like the following:

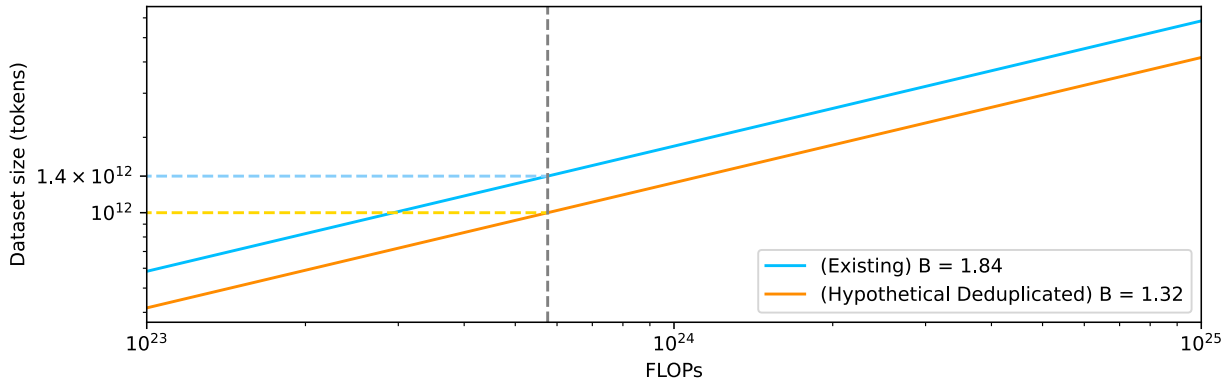


Figure 5.7: Notice that the slopes are the same (that is, $b = 0.5$) but the *intercepts* B are different

To interpret this, remember that the scaling law is fitted as $D_{\text{opt}} = BC^b$. On a log-log plot, B acts as the intercept, while b is the slope. **Chinchilla makes claims about the slope:** that $b \approx 0.5$ (and $a \approx 0.5$ on the model size side). This means *once* you've already found a value $(N_{\text{opt}}, D_{\text{opt}})$ at a small value of C, Chinchilla provides you the recipe to scale up your N on more tokens D from that exact data mixture.

But even a single $(N_{\text{opt}}, D_{\text{opt}})$ pair will be unknown to you for your exact data/architecture setup at the start of your experiments, so at the minimum you'll want to perform ~3-5 small training runs

to produce one IsoFLOP curve, to produce at least one $(N_{\text{opt}}, D_{\text{opt}})$ you can extrapolate from.

Subsequent work (such as Dey et al. (2023), Appendix D) replicated MassiveText-like dynamics in a, b[15] *and* in A, B (finding ~20 tokens per parameter to be optimal) on a different dataset, the Pile (Gao et al. 2020). This suggests a general rule of thumb (20 tokens/param) for decoder-only, natural language models[16], that have been recommended in other blogposts (such as Anthony, Biderman, and Schoelkopf (2023)).

That said, it is important to recall the assumptions this rule-of-thumb is built on[17], and to be willing to calculate a new IsoFLOP curve if any assumption is violated.

5.3.1 Compute optimal?

Chinchilla's scaling laws are concerned with optimality under *one* definition of cost: the amount of FLOPs used in training. This translates to real world, *monetary* cost only if you're paying per-unit of compute. In practice, if you're a big lab, you likely already have a contract reserving accelerator capacity with one of the large cloud providers; the compute is already *paid for*, and the real cost is *opportunity*.

Chinchilla only tells you how to produce the "best" (lowest validation loss) model given a compute budget; the meta-calculus of how valuable each model *is*[18], is an entirely other (and very real!) concern that it cannot answer. Moreover, there are practical instances where you may want to "overtrain" a smaller model with more data (= higher training costs), so that it is easier to serve to end users, as we see next.

5.3.1.1 Inference Costs

In industry applications, much of the cost will often not be in *training*, but in *inference*, serving the model to end users.

Since transformer inference cost is linear in model size, a model that's 3x smaller will take 3x less FLOPs for inference[19]. Suppose the compute optimal model for an initial 2.66×10^{21} FLOPs budget is $N = 2.8$B params trained on $D = 156$B tokens. We can always train a 1.5B parameter model for *longer* than compute-optimal to achieve the same performance[20] as the 2.8B model. This "extra compute" we call the compute overhead. Just how large is it?

Chinchilla's scaling laws also give us a way to quantify this! Instead of looking at IsoFLOP curves (where the FLOP is the quantity held constant on each curve) we can look at IsoLoss curves[21] (where the loss is the quantity held constant on each curve). This specific N and D produce a loss of 2.24; we can produce a full range of (N, D) values with that same loss, as shown on the left.

[15] 1:1 data:model scaling, as expected

[16] without needing to produce IsoFLOPs curves for each new dataset

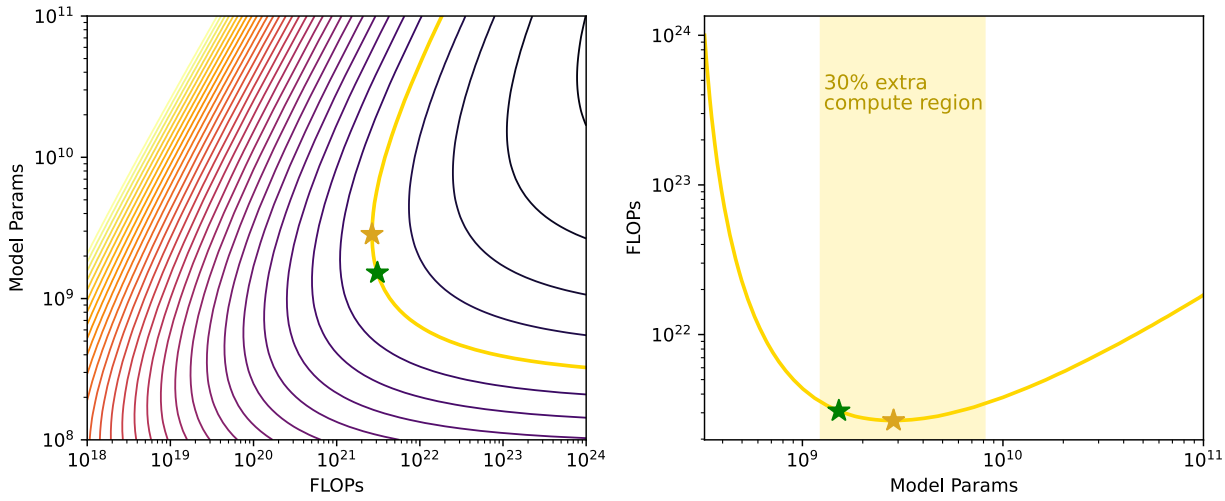
[17] That the model is decoder-only, natural language, with a mixture similar to MassiveText/the Pile.

[18] These are questions such as "Given our compute reservations, is a 3B model that can be deployable in 6 weeks more valuable than a 20B model deployable in 3 months?".

[19] And hence, properly optimized at scale, will be 3x cheaper.

[20] that is, the same validation loss

[21] For this analysis, we use the formula in equation 10, appendix D.3, which is $L(N, D) = 1.69 + \frac{406.4}{N^{0.34}} + \frac{410.7}{N^{0.28}}$



We can also plot the IsoLoss curve for a loss of 2.24, as on the right. As we see, the "cheapest" way to achieve that loss[22] is through the compute optimal 2.8B model (gold star). But we can also produce a model nearly half the size (1.5B, green star) if we're willing to spend 3.09×10^{21}, or 16% more instead. When is this worth it?

5.3.1.1.1 Payoff

Short answer: calculate the minimum number of tokens that, when inferred using the 1.5B "over-trained" vs 2.8B "optimal" model, will have saved us more in inference than the excess spent in training. Then:

$$\begin{aligned}
\text{overhead} &= (2N)D_{\text{inf}} \\
\text{2.8B FLOPs} - \text{1.5B FLOPs} &= (2 \times 1.5 \times 10^9)D_{\text{inf}} \\
3.09 \times 10^{21} - 2.66 \times 10^{21} &= (2 \times 1.5 \times 10^9)D_{\text{inf}} \\
D_{\text{inf}} &= \frac{3.09 \times 10^{21} - 2.66 \times 10^{21}}{2 \times 1.5 \times 10^9} \\
D_{\text{inf}} &= 140\text{B}
\end{aligned}$$

That is, if we're serving more than 140B tokens, the cost savings from this ~45% smaller model will become worth it. There's a couple things to note here:

- The *inference* cost of D_{inf} tokens passed through a model of size N is $\approx 2ND_{\text{inf}}$, not $\approx 6ND_{\text{inf}}$. This is because we only need the costs for the forward pass, *not* the forward + backward pass (where the backward pass is 2x more than the forward pass).

[22] Note that we're talking exclusively about how to produce a smaller model with the same validation loss. But as we'll see next, loss is not the same as downstream performance!

- This also means every training token is 3x more expensive than every inference token; we need to pass in at least 3 inference tokens for every extra training token for the cost to be worth it. Put more directly, overtraining only makes sense for models that will receive very high usage.
- To put the 140B inference tokens into context, the number of training tokens D needed for a model of size 1.5B to achieve a loss of 2.24 (based on the loss formula above) is $\approx$ 339B. This means we'll only need to serve a fraction of the tokens needed to train the model for the compute overhead to be worth it.

Generally speaking, for models intended to be used in production, a compute overhead of upto ~100% will often be worth paying to obtain a model ~30% the size (see De Vries (2023) for this analysis).

5.3.1.2 Latency

Compute isn't the only factor at play at inference time: latency is too! A model that's 50% the size not only uses 50% the compute, but could also reduce the computation time by upto 50%[23]. If that is the difference between your user waiting for the output vs. your service being not viable, that is a constraint that needs to take priority over training a compute-optimal model.

Moreover, Chinchilla's scaling laws tell you that you can *optimally* use an increased compute budget by scaling your model size and data 1:1, but you can *ignore* it and just increase data, keeping the model size fixed. If there *is* an upper bound to the size of your model (due to latency considerations or otherwise), you can *always* improve performance (subject to diminishing returns) by training on more tokens.

This, combined with trillion-token datasets and the ability to reuse data for upto 4 epochs (Muennighoff et al. 2023) means sub-100B parameter models (and likely even larger) are not data constrained during pre-training, and can be trained *far* past compute-optimal (as is the case with model families such as LLama 2 (Touvron et al. 2023)) for maximal performance under compute/latency constrained inference.

5.3.2 Loss $\neq$ Performance

One last thing to note is that Chinchilla is concerned exclusively with minimizing loss on a validation set: it makes no direct claims about the actual capabilities of a model. With language models, the loss used most commonly is perplexity (Huyen 2019). Intuitively, perplexity measures between how many "options" in its vocabulary, on average, a language model is "choosing between" when generating the next token (lower is better). But perplexity only *correlates with* the behaviors we want; it itself is not the objective we care about.

This can lead to counterintuitive behaviors: for instance in the PaLM2 report (Anil et al. 2023), although a 9.5B model achieves a lower loss on $C = 1 \times 10^{22}$ FLOPs, a 16.1B model trained with the same amount of compute (but higher loss) actually performs better on downstream evaluations.

[23] Assuming we still have the same amount of hardware, and are not too bottlenecked on the generation of the output tokens.

It is always critical to remember that language modeling is a proxy objective for natural language understanding (NLU) capabilities. We're still subject to Goodharting (Sohl-Dickstein 2022) on whether this proxy objective (perplexity) optimizes for what we really want!

5.4 Conclusion

Understanding Chinchilla's scaling laws as derived not only helps us better understand the assumptions made, but also enables us to recalculate them given a substantial change in dataset, model architecture, or domain. Moreover, understanding Chinchilla's definition of a compute-optimal model helps us decide when we might *not* want one, and might want to overtrain a smaller model instead.

Overall, Chinchilla is much more than just training compute optimal models: it's being able to make *quantifiable* tradeoffs between cost, model size and dataset size.

Bibliography

Abbas, Amro, Kushal Tirumala, Dániel Simig, Surya Ganguli, and Ari S. Morcos. 2023. "SemDeDup: Data-Efficient Learning at Web-Scale Through Semantic Deduplication." https://arxiv.org/abs/2303.09540.

Anderson, P. W. 1972. "More Is Different." *Science* 177 (4047): 393–96. https://doi.org/10.1126/science.177.4047.393.

Anil, Rohan, Andrew M. Dai, Orhan Firat, Melvin Johnson, Dmitry Lepikhin, Alexandre Passos, Siamak Shakeri, et al. 2023. "PaLM 2 Technical Report." https://arxiv.org/abs/2305.10403.

Anthony, Quentin, Stella Biderman, and Hailey Schoelkopf. 2023. "Transformer Math 101." https://blog.eleuther.ai/transformer-math/.

Babuschkin, Igor, Kate Baumli, Alison Bell, Surya Bhupatiraju, Jake Bruce, Peter Buchlovsky, David Budden, et al. 2020. "The DeepMind JAX Ecosystem." http://github.com/deepmind.

Bavarian, Mohammad, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. 2022. "Efficient Training of Language Models to Fill in the Middle." arXiv. https://doi.org/10.48550/ARXIV.2207.14255.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Janvin. 2003. "A Neural Probabilistic Language Model." *J. Mach. Learn. Res.* 3 (null): 1137–55.

Borgeaud, Sebastian, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, George Bm Van Den Driessche, et al. 2022. "Improving Language Models by Retrieving from Trillions of Tokens." In *Proceedings of the 39th International Conference on Machine Learning*, edited by Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato, 162:2206–40. Proceedings of Machine Learning Research. PMLR. https://proceedings.mlr.press/v162/borgeaud22a.html.

Bostrom, Kaj, and Greg Durrett. 2020. "Byte Pair Encoding Is Suboptimal for Language Model Pretraining." In *Findings of the Association for Computational Linguistics: EMNLP 2020*, edited by Trevor Cohn, Yulan He, and Yang Liu, 4617–24. Online: Association for Computational Linguistics. https://doi.org/10.18653/v1/2020.findings-emnlp.414.

Brown, Tom B., Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. 2020. "Language Models Are Few-Shot Learners." https://arxiv.org/abs/2005.14165.

Chan, Stephanie C. Y., Adam Santoro, Andrew K. Lampinen, Jane X. Wang, Aaditya Singh, Pierre H. Richemond, Jay McClelland, and Felix Hill. 2022. "Data Distributional Properties Drive Emergent in-Context Learning in Transformers." arXiv. https://doi.org/10.48550/ARXIV.2205.05055.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, et al. 2022. "PaLM: Scaling Language Modeling with Pathways." https://arxiv.org/abs/2204.02311.

De Vries, Harm. 2023. "Go Smol or Go Home." https://www.harmdevries.com/post/model-size-vs-compute-overhead/.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–86. Minneapolis, Minnesota: Association for Computational Linguistics. https://doi.org/10.18653/v1/N19-1423.

Dey, Nolan, Gurpreet Gosal, Zhiming, Chen, Hemant Khachane, William Marshall, Ribhu Pathria, Marvin Tom, and Joel Hestness. 2023. "Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster." https://arxiv.org/abs/2304.03208.

Foundation, Wikimedia. n.d. "Wikimedia Downloads." https://dumps.wikimedia.org.

Gao, Leo, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, et al. 2020. "The Pile: An 800GB Dataset of Diverse Text for Language Modeling." https://arxiv.org/abs/2101.00027.

Heek, Jonathan, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. 2020. "Flax: A Neural Network Library and Ecosystem for JAX." http://github.com/google/flax.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, et al. 2022. "Training Compute-Optimal Large Language Models." https://arxiv.org/abs/2203.15556.

Hornik, Kurt. 1991. "Approximation Capabilities of Multilayer Feedforward Networks." *Neural Networks* 4 (2): 251–57. https://doi.org/https://doi.org/10.1016/0893-6080(91)90009-T.

Hu, Edward J., Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. "LoRA: Low-Rank Adaptation of Large Language Models." https://arxiv.org/abs/2106.09685.

Huang, Austin, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman. 2022. "The Annotated Transformer." http://nlp.seas.harvard.edu/annotated-transformer/#encoder-and-decoder-stacks.

Huang, Yanping, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, et al. 2019. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." In *Advances in Neural Information Processing Systems*, edited by H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, and R. Garnett. Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2019/file/093f65e080a295f8076b1c5722a46aa2-Paper.pdf.

Huyen, Chip. 2019. "Evaluation Metrics for Language Modeling." *The Gradient*.

Izacard, Gautier, Patrick Lewis, Maria Lomeli, Lucas Hosseini, Fabio Petroni, Timo Schick, Jane Dwivedi-Yu, Armand Joulin, Sebastian Riedel, and Edouard Grave. 2022. "Few-Shot Learning with Retrieval Augmented Language Models." arXiv. https://doi.org/10.48550/ARXIV.2208.03299.

Kaplan, Jared, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. "Scaling Laws for Neural Language Models." https://arxiv.org/abs/2001.08361.

Karpathy, Andrej. 2015. "The Unreasonable Effectiveness of Recurrent Neural Networks." http://karpathy.github.io/2015/05/21/rnn-effectiveness/.

Kingma, Diederik P., and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, edited by Yoshua Bengio and Yann LeCun. http://arxiv.org/abs/1412.6980.

Kudo, Taku, and John Richardson. 2018. "SentencePiece: A Simple and Language Independent

Subword Tokenizer and Detokenizer for Neural Text Processing." In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, edited by Eduardo Blanco and Wei Lu, 66–71. Brussels, Belgium: Association for Computational Linguistics. https://doi.org/10.18653/v1/D18-2012.

Lang, Ken. 1995. "Newsweeder: Learning to Filter Netnews." In *Proceedings of the Twelfth International Conference on Machine Learning*, 331–39.

Lazaridou, Angeliki, Elena Gribovskaya, Wojciech Stokowiec, and Nikolai Grigorev. 2022. "Internet-Augmented Language Models Through Few-Shot Prompting for Open-Domain Question Answering." arXiv. https://doi.org/10.48550/ARXIV.2203.05115.

Lin, Zeming, Halil Akin, Roshan Rao, Brian Hie, Zhongkai Zhu, Wenting Lu, Allan dos Santos Costa, et al. 2022. "Language Models of Protein Sequences at the Scale of Evolution Enable Accurate Structure Prediction." *bioRxiv*. https://doi.org/10.1101/2022.07.20.500902.

Loshchilov, Ilya, and Frank Hutter. 2019. "Decoupled Weight Decay Regularization." In *International Conference on Learning Representations*. https://openreview.net/forum?id=Bkg6RiCqY7.

Marr, D., and T. Poggio. 1976. "From Understanding Computation to Understanding Neural Circuitry." USA: Massachusetts Institute of Technology.

McCandlish, Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. "An Empirical Model of Large-Batch Training." https://arxiv.org/abs/1812.06162.

Meng, Kevin, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. "Locating and Editing Factual Associations in GPT." *Advances in Neural Information Processing Systems* 35.

Meng, Kevin, Arnab Sen Sharma, Alex Andonian, Yonatan Belinkov, and David Bau. 2022. "Mass Editing Memory in a Transformer." *arXiv Preprint arXiv:2210.07229*.

Mikolov, Tomas, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. "Efficient Estimation of Word Representations in Vector Space." arXiv. https://doi.org/10.48550/ARXIV.1301.3781.

Min, Sewon, and Sang Michael Xie. 2022. "How Does in-Context Learning Work? A Framework for Understanding the Differences from Traditional Supervised Learning." http://ai.stanford.edu/blog/understanding-incontext/.

Muennighoff, Niklas, Alexander M. Rush, Boaz Barak, Teven Le Scao, Aleksandra Piktus, Nouamane Tazi, Sampo Pyysalo, Thomas Wolf, and Colin Raffel. 2023. "Scaling Data-Constrained Language Models." https://arxiv.org/abs/2305.16264.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Anand Korthikanti, Dmitri Vainbrand, et al. 2021. "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." arXiv. https://doi.org/10.48550/ARXIV.2104.04473.

Olsson, Catherine, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, et al. 2022. "In-Context Learning and Induction Heads." *Transformer Circuits Thread*.

Rae, Jack W., Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, et al. 2022. "Scaling Language Models: Methods, Analysis & Insights from Training Gopher." https://arxiv.org/abs/2112.11446.

Raffel, Colin, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2020. "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer." *Journal of Machine Learning Research* 21 (140): 1–67. http://jmlr.org/papers/v21/20-074.html.

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '20. Atlanta, Georgia:

IEEE Press.

Roberts, Adam, Hyung Won Chung, Anselm Levskaya, Gaurav Mishra, James Bradbury, Daniel Andor, Sharan Narang, et al. 2022. "Scaling up Models and Data with `t5x` and `seqio`." *arXiv Preprint arXiv:2203.17189*. https://arxiv.org/abs/2203.17189.

Savinov, Nikolay, Junyoung Chung, Mikolaj Binkowski, Erich Elsen, and Aaron van den Oord. 2022. "Step-Unrolled Denoising Autoencoders for Text Generation." In *International Conference on Learning Representations*. https://openreview.net/forum?id=T0GpzBQ1Fg6.

Schunk, Dale H. 2011. *Learning Theories*. 6th ed. Upper Saddle River, NJ: Pearson.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, edited by Katrin Erk and Noah A. Smith, 1715–25. Berlin, Germany: Association for Computational Linguistics. https://doi.org/10.18653/v1/P16-1162.

Shazeer, Noam. 2020. "GLU Variants Improve Transformer." https://arxiv.org/abs/2002.05202.

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." arXiv. https://doi.org/10.48550/ARXIV.1909.08053.

Smith, Shaden, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, et al. 2022. "Using DeepSpeed and Megatron to Train Megatron-Turing NLG 530B, a Large-Scale Generative Language Model." https://arxiv.org/abs/2201.11990.

Sohl-Dickstein, Jascha. 2022."Too much efficiency makes everything worse: overfitting and the strong version of Goodhart's law ." https://sohl-dickstein.github.io/2022/11/06/strong-Goodhart.html.

Strudel, Robin, Corentin Tallec, Florent Altché, Yilun Du, Yaroslav Ganin, Arthur Mensch, Will Grathwohl, et al. 2022. "Self-Conditioned Embedding Diffusion for Text Generation." arXiv. https://doi.org/10.48550/ARXIV.2211.04236.

Timbers, Finbarr. 2023. "Five Years of Progress in GPTs." *Five Years of Progress in GPTs - by Finbarr Timbers*. Artificial Fintelligence. https://finbarrtimbers.substack.com/p/five-years-of-progress-in-gpts.

Touvron, Hugo, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, et al. 2023. "Llama 2: Open Foundation and Fine-Tuned Chat Models." https://arxiv.org/abs/2307.09288.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. "Attention Is All You Need." In *Advances in Neural Information Processing Systems*, edited by I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett. Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf.

Vincent, James. 2022. "ChatGPT Proves AI Is Finally Mainstream — and Things Are Only Going to Get Weirder." *The Verge*. https://www.theverge.com/2022/12/8/23499728/ai-capability-accessibility-chatgpt-stable-diffusion-commercialization.

Weng, Lilian. 2021. "How to Train Really Large Models on Many GPUs?" *Lilianweng.github.io*, September. https://lilianweng.github.io/posts/2021-09-25-train-large/.

Xu, Yuanzhong, HyoukJoong Lee, Dehao Chen, Blake A. Hechtman, Yanping Huang, Rahul Joshi, Maxim Krikun, et al. 2021. "GSPMD: General and Scalable Parallelization for ML Computation Graphs." *CoRR* abs/2105.04663. https://arxiv.org/abs/2105.04663.

Yoo, Joanna, Kuba Perlin, Siddhartha Rao Kamalakara, and João G. M. Araújo. 2022. "Scalable

Training of Language Models Using JAX Pjit and TPUv4." arXiv. https://doi.org/10.48550/ARXIV.2204.06514.

Zheng, Lianmin, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, et al. 2022. "Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning." In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 559–78. Carlsbad, CA: USENIX Association. https://www.usenix.org/conference/osdi22/presentation/zheng-lianmin.

A Discrete Distributions

A.1 Maximum Likelihood Estimation

Suppose we receive a coin, and we wish to find out if it's biased or not. We run 100 tosses, and 62 of them come up heads, so we figure it's biased at 62% heads, 38% tails.

What just happened here? To rewind, the scenario looks like this:

- **Support**: We have a random variable X, with exactly two possible[1] outcomes: heads or tails.
- **Parametrization**: X is a discrete random variable, and each of its outcomes has a true probability value, $p(0)$ and $p(1)$. We don't know what these values are, so we *parametrize* a two-valued discrete probability distribution, with $p_\theta(1) = \theta$ and $p_\theta(0) = 1 - \theta$.
- **Sampling**: Although we don't know the distribution, we *can* generate samples from it by tossing the coin repeatedly. We gather a "dataset" of 62 heads and 38 tails.
- **Estimation**: We can then use these samples to *estimate* the value of the parameter θ. Specifically, we can find the value of θ that *maximizes* the likelihood of the outcome we observed.

Given a value of θ, we can compute the probability we'd see a given outcome:

- The probability of observing one heads is θ.
- Since each observation is *independent*[2], the probability of observing one heads and two tails is $\theta(1-\theta)(1-\theta) = \theta(1-\theta)^2$
- For the overall observation here, of 62 heads and 38 tails, we have $\theta\theta...(1-\theta)(1-\theta) = \theta^{62}(1-\theta)^{38}$

However, θ is a *variable* for which we need to *estimate* a value. One process to do so, is to see the above term as a *function* of theta $L(\theta) = \theta^{62}(1-\theta)^{38}$, known as the likelihood function. We can then find the value of θ that maximizes the likelihood function is a process, aptly called maximum likelihood estimation, or MLE. This is the value of θ that, when used in $p_\theta(\cdot)$ will result in the highest[3] probability being assigned to our observation.

Note that the likelihood function as is will produce *extremely* tiny values; for numerical stability, we usually *minimize* the negative *log*-likelihood, which here is:

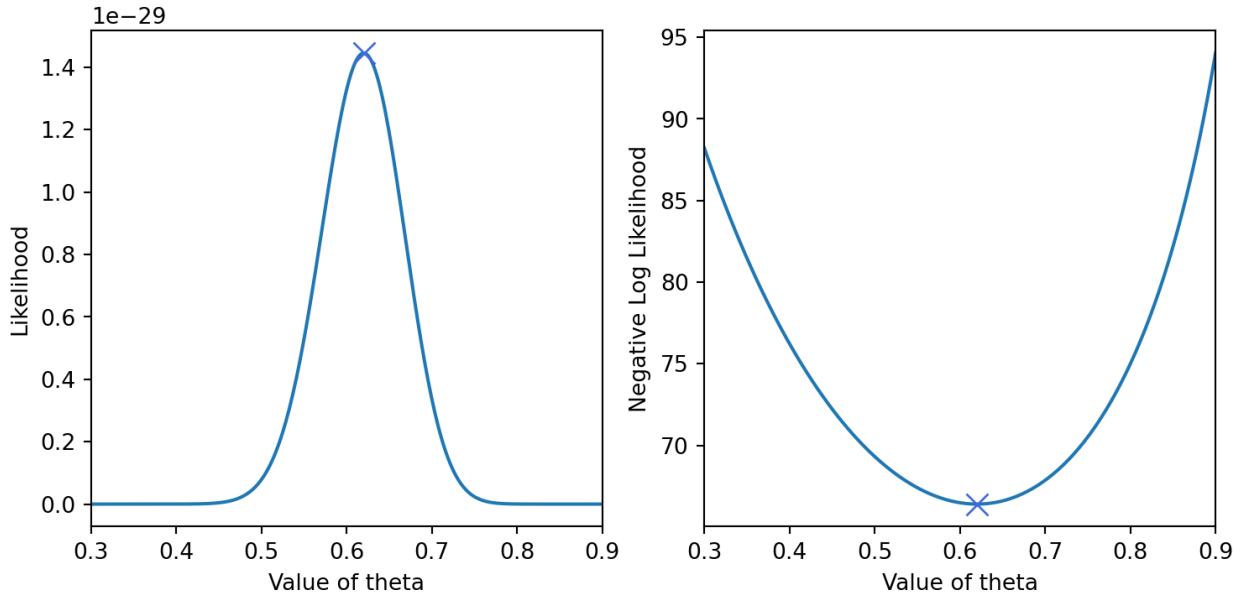
$$\begin{aligned} -\ln L(\theta) &= -\ln(\theta^{62}(1-\theta)^{38}) \\ &= -62\ln(\theta) - 38\ln(1-\theta) \end{aligned}$$

[1] Possible meaning "non-zero probability"

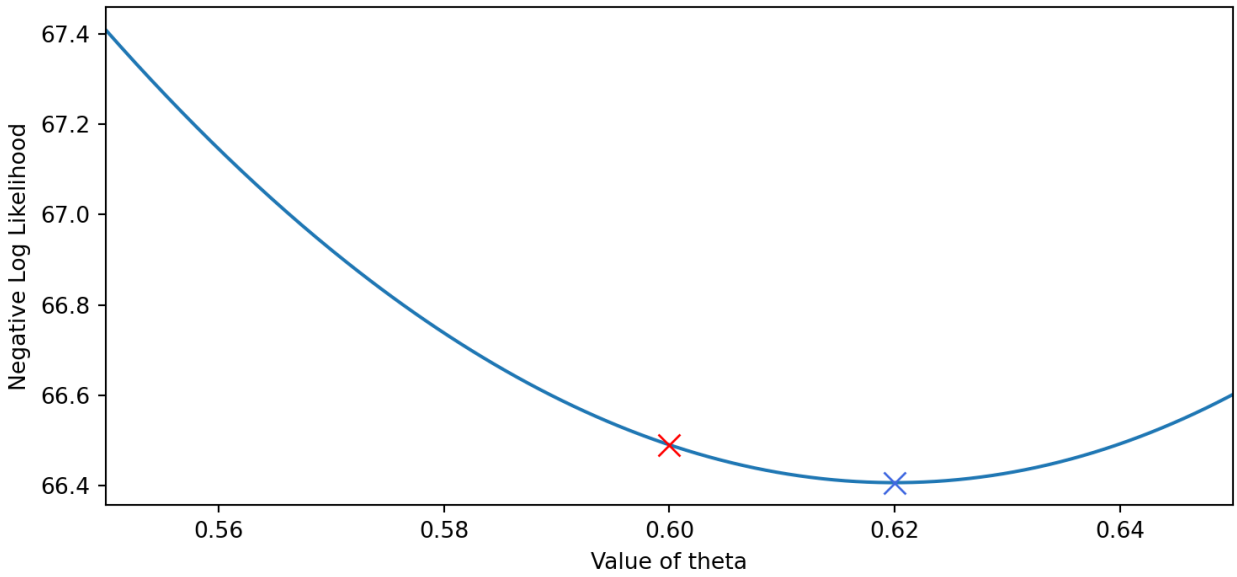
[2] Independence between individual observations is a key assumption; if they're not independent, $p(\text{obs}) = p(\text{obs}_1)p(\text{obs}_2)\ldots$ doesn't hold.

[3] compared to other values of θ

Visualizing both, we see they are maximized and minimized (respectively) at the same value, $\theta = 0.62$, the same as our earlier "back of the hand" estimate:



Now suppose we learn the *true* value of $p(1) = 0.6$. Note that 0.6 is in fact, *less* likely than our estimate of $p_\theta(1) = \theta = 0.62$:



This is because we estimated θ with the value that maximizes the probability of the *observed* samples. In the limit of an infinite number of samples, the likelihood function will be maximized at a value of θ that produces the true probabilities. But *in practice*, there will be variance in the number of heads in 100 samples (we may have 61 heads, or 58, or…); and in turn, variance in our estimates.

A.2 Discrete distributions as vectors

We can write the probability mass function of the "coin toss" random variable in the following vector. Each entry simply stores the probability for one of the possible outcomes:

H	T
0.6	0.4

In general, for a discrete random variable with V possible outcomes, there are $V-1$ free parameters. This is because all the probabilities must sum to 1:

- For $V = 2$, as in the case above, we have the probability for one outcome be θ. The other must then be $1 - \theta$ for both to sum to 1.
- For $V = 3$, we can have probabilities for two of the outcomes specified by θ_1 and θ_2. The third outcome then must have the probability $1 - \theta_1 - \theta_2$.
- For $V = n$, we have the probabilities for the first $n - 1$ outcomes specified by θ_i, and the final one by $1 - \sum_{i=1}^{n} \theta_i$.

There are potentially detrimental consequences when we "tie" the estimates of the probabilities of different outcomes, as we see next.

A.2.1 Constrained estimates

Suppose we have a discrete random variable with three possible outcomes ($V = 3$). The vector encodes these three outputs for the *true* probability mass function $p(\cdot)$[4]:

a	b	c
0.03	0.01	0.96

As before, we don't know $p(\cdot)$, so we approximate it with $p_\theta(\cdot)$. To find the best value of the parameters θ, we use maximum likelihood estimation. We define two parametrizations $p_\theta(\cdot)$:

- one that is free with $V - 1 = 2$ parameters (θ_1 and θ_2). That is, we can independently update $p_\theta(a)$ without affecting $p_\theta(b)$.[5]

[4] For example, $p(a) = 0.03$

[5] We do keep the baseline constraint that the probability of all three outcomes must sum to 1.

- another that is constrained to only have one parameter (θ_c). Note that $p_\theta(a)$ and $p_\theta(b)$ are now tied; updating one will also change the other.

As like the true $p(\cdot)$, we can also write the outcomes of these approximations $p_\theta(\cdot)$ in a vector:

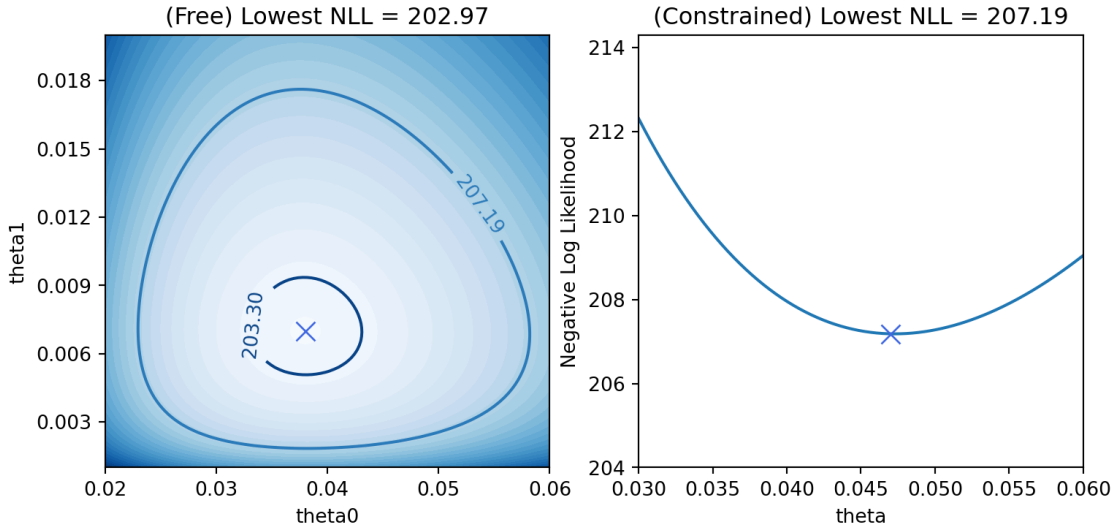
Free

a	b	c
θ_1	θ_2	$1 - \theta_1 - \theta_2$

Constrained

a	b	c
θ_c	θ_c^2	$1 - \theta_c - \theta_c^2$

Suppose we draw 1000 samples, receiving 38 a's, 7 b's and 955 c's. For each parametrization, we find the values of the parameters that maximize the likelihood of receiving this outcome:



On the left, in the "free" parametrization, the minimum is reached at $\theta_1 = 0.038, \theta_2 = 0.007$, just as would be expected by estimating from the proportions directly ($\theta_1 = \frac{38}{1000}, \theta_2 = \frac{7}{1000}$). In general, this is true for any number of outcomes V; the maximum likelihood estimate for each probability θ_i is the proportion of times said outcome appears in the entire dataset.

On the *right* however, with only one free parameter under the above parametrization, the minimum is reached at $\theta_c = 0.047$. This corresponds to $p_\theta(a) = 0.047, p_\theta(b) = 0.047^2 = 0.002$. Not only is this a bad estimate of the *true* probabilities (where $p(a) = 0.03$), it's far even from the estimate that was reached by the free model (where $p_\theta(a) = 0.038$).

The lowest NLL reached here is 207.19. By plotting a contour back on the free parametrization (left), we can find parameter pairs (θ_1, θ_2) with a lower NLL. With the free parametrization, θ_1 could be anywhere between 0.025 (2.5%) and 0.055 (5.5%) and still have a higher likelihood for the observation than the best value of θ_c.

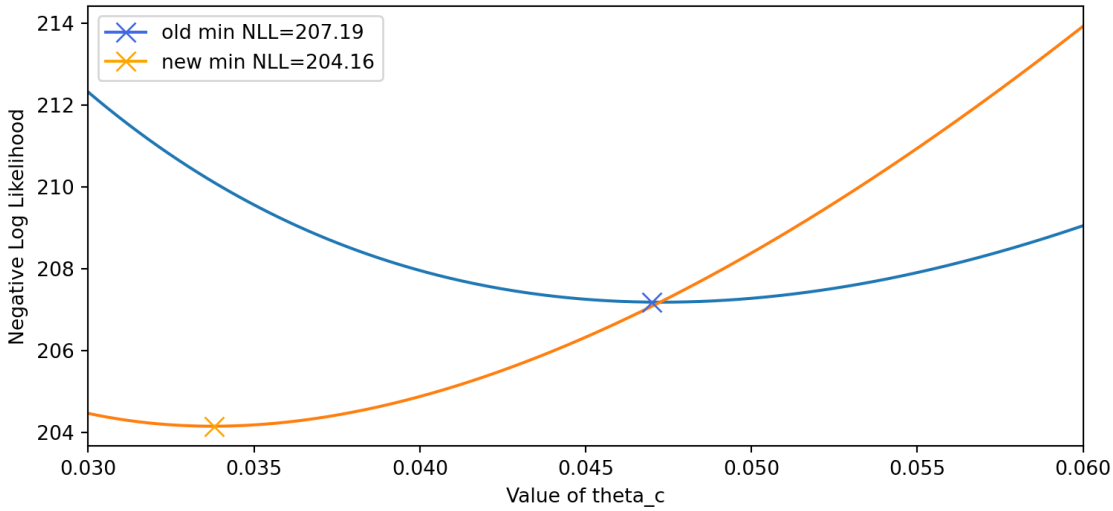
A.2.1.1 A better constraint

However, suppose we knew more about the problem: this is actually the annual failure rate of a widget, with a, b and c meaning "big gear failure", "small gear failure" and "no failure". Moreover, prior physical knowledge of the system informs us that the big gear should fail at 3x the rate of the smaller one, that is: $p(a) = 3p(b)$. This inspires the following parametrization, where this constraint holds for *all* values of θ_c:

Constrained

a	b	c
θ_c	$\frac{\theta_c}{3}$	$1 - \frac{4\theta_c}{3}$

By having $p_\theta(a) = \theta_c$ and $p_\theta(a) = \frac{\theta_c}{3}$, we guarantee the constraint $p_\theta(a) = 3p_\theta(b)$ is met for *all* values of θ_c. As before, we find the value of θ_c with the lowest NLL:



The optimal of θ_c then is $\theta_c = 0.034$, corresponding to $p_\theta(a) = 0.034$, $p_\theta(b) = \frac{0.034}{3} = 0.011$.

Not only is this new constrained estimate (0.034) better than the previous one, the probabilities are *closer* to the true ones (0.03) than even with the most flexible model (0.038). This is because we've correctly incorporated our prior knowledge that $p_\theta(a) = 3p_\theta(b)$. To summarize:

- A parametrization for a discrete distribution with V outcomes, that is guaranteed to be *capable* of converging to the true distribution (***given an infinite amount of data***) has $V-1$ parameters; one parameter per outcome.
- Bad constraints will *permanently* bias the model; even with an infinite amount of data its ability to represent the correct probabilities will be constrained (in the first constrained example, we have $p_\theta(b) = (p_\theta(a))^2$, *even though* it is false)
- *However*, a truthful constraint (here, $p_\theta(a) = 3p_\theta(b)$) *will* converge to the true parameters with an infinite amount of data. Moreover, it will converge *faster*, since the constraints will reduce the impact of variance in the samples.

A.3 Joint Distributions as tensors

Let's now look at a situation with two random variables. Suppose we now have the following setup:

- I have two time slots each day, each of which I can fill up with either reading or hiking.
- The activity done in the first and second time slots are X_1 and X_2 respectively.

Then, let the joint distribution[6] of X_1 and X_2 be the following:

$\text{hiking}_1, \text{hiking}_2$	$\text{hiking}_1, \text{read}_2$	$\text{read}_1, \text{hiking}_2$	$\text{read}_1, \text{read}_2$
0	0.5	0.1	0.4

Note that these variables are *not* independent: for instance, if we learn hiking is the second activity of the day, we can deduce hiking could not have been the first activity that day, as $p(\text{hiking}_1, \text{hiking}_2) = 0$. Estimation works the same as previously: if 21 of 200 observed days are "read first, then hike", then $p_\theta(\text{read}_1, \text{hiking}_2) = \frac{21}{200}$.

Moreover, we note that each of the X's has the same $V = 2$ outcomes: H or T. We could then convert the "vector" above into the following matrix:

$$\text{Joint } X_1, X_2$$

	hiking_2	read_2
hiking_1	0	0.5
read_1	0.1	0.4

In general, with m random variables, each with V outcomes individually, there are V^m total possible joint outcomes. Here, this is $2^2 = 4$ outcomes. One can imagine storing these joint outcomes in a tensor with m axes, each with V dimensions. Here we store these 4 outcomes in a 2×2 matrix; for 4 variables with 3 outcomes each we could use a $3 \times 3 \times 3 \times 3$ tensor (with 81 total "outcomes").

Note that this grows *exponentially* in the number of outcomes. With $V = 10$ and $m = 10$, we have 10^{10} (ten billion) possible outcomes. *Each* of these is an outcome we need to observe (multiple times) to estimate the probabilities for. Compared to the previous $V - 1$ parameters for the distribution of a single random variable, we now have $V^m - 1$ parameters for the joint distribution of m random variables.

This exponential growth (and in turn, the number of samples required) makes directly estimating probabilities for anything but the simplest of joint distributions intractable. Maybe conditional probabilities can help?

[6] That is, the function that stores the probabilities of specific outcomes for both X_1 and X_2, compared to *all* other pairs of outcomes.

A.4 Conditional Distributions as vectors

Conditional distributions also fit neatly into this tensor form. The chain rule in probability tells us that any joint probability can be decomposed into the product of conditional probabilities. Extending the previous example:

$$p(\text{hiking}_1, \text{reading}_2) = p(\text{reading}_2 \mid \text{hiking}_1)p(\text{hiking}_1)$$

Here, the joint probability of "hiking then reading" is the product of the marginal probability of hiking first and the conditional probability of reading *after* hiking. We can represent this factorization in vector form as follows:

Joint X_1, X_2

	hiking_2	read_2
hiking_1	0	0.5
read_1	0.1	0.4

Marginal X_1

hiking_1	read_1
0.5	0.5

hiking_1

Conditional $X_2 \mid X_1$

hiking_2	read_2
0	1

To analyze this factorization:

- We've initially stored the joint probabilities $p_\theta(\cdot, \cdot)$ in a matrix form. This has $V^m = 2^2 = 4$ outcomes, and so $V^m - 1 = 3$ parameters.
- We can factorize that matrix into a vector $p_\theta(\cdot)$ representing the marginal probability, and another vector $p_\theta(\cdot \mid \text{hiking}_1)$ representing the conditional probability.
- Each of these two vectors has $V = 2$ entries, and so $V - 1 = 1$ parameters. With 2 vectors, we have 2 parameters total.

It appears the factorized version has fewer parameters (2 vs 3) than the original. There's just one problem: it's incomplete! We're *only* looking at the conditional probability table for where $X_1 = \text{hiking}_1$; there's another one for $X_1 = \text{read}_1$:

Joint X_1, X_2

	hiking_2	read_2
hiking_1	0	0.5
read_1	0.1	0.4

Marginal X_1

hiking_1	read_1
0.5	0.5

hiking_1

read_1

Conditional $X_2 \mid X_1$

hiking_2	read_2
0	1

hiking_2	read_2
0.2	0.8

With three vectors (one marginal, two conditional), we have 3 parameters, the same as the original joint probability matrix. This unfortunately is true in general as well; for m discrete random variables (each with V outcomes):

- the conditional probability vector of $p(\cdot \mid x_{1:k-1})$ has V entries ($V-1$ parameters), as X_k only has V outcomes.
- However, there are V^{k-1} such tables, one for *every* possible combination of prior values $x_{1:k-1} = x_1, ..., x_{k-1}$.
- The total number of parameters for a variable conditioned on $k-1$ "already observed" outcomes then is $V^{k-1}(V-1)$.

A.4.1 General Case

The *general* chain rule for a joint distribution with m variables is:

$$p(x_1, x_2, ..., x_m) = p(x_m \mid x_{1:m-1})p(x_{m-1} \mid x_{1:m-2})...p(x_2 \mid x_1)p(x_1)$$

Using the result in the previous section, let's compute the number of free parameters in the distributions in the product above:

- For the marginal $p(\cdot)$, there are $V-1$ parameters.
- For $p(\cdot \mid x_1)$, conditioned on one outcome there are $V(V-1)$ parameters.
- For $p(\cdot \mid x_{1:m-1})$, conditioned on $m-1$ outcomes there are $V^{m-1}(V-1)$ parameters.

Summing up the number of parameters for all m distributions in the product, we have:

$$\begin{aligned} N &= (V-1) + V(V-1) + V^2(V-1) + ... + V^{m-1}(V-1) \\ &= \sum_{k=0}^{m-1} V^k(V-1) \\ &= V^m - 1 \end{aligned}$$

We're back at $V^m - 1$ parameters, same as in the joint distribution case. Breaking the joint probability matrix into conditional probability vectors *changes* the parametrization, but doesn't change the *number* of parameters.

A.4.2 Independence as constraints

Earlier with the widgets example, we used context to reduce the number of parameters needed. Now suppose here, we knew that each of the X_i was a coin toss (with 60% probability of heads as before), and we had a sequence of $m = 30$ coin tosses. There's 2^{30} ($\approx$ a billion) possible sequences of coin tosses here. And you'd need to observe each of these billion sequences multiple times to estimate the proportion for each sequence compared to the total.

But it's silly to store a separate probability for each unique sequence (e.g. HTH...T): we know the coin tosses are independent from each other, and identically distributed. We could simplify the chain rule from before as follows[7]:

$$\begin{aligned} p_{X_1,X_2,...,X_m}(x_1, x_2, ..., x_m) &= p_{X_m}(x_m \mid x_{1:m-1}) p_{X_{m-1}}(x_{m-1} \mid x_{1:m-2}) ... p_{X_2}(x_2 \mid x_1) p_{X_1}(x_1) \\ &= p_{X_m}(x_m) p_{X_{m-1}}(x_{m-1}) ... p_{X_2}(x_2) p_{X_1}(x_1) \\ &= p(x_m) p(x_{m-1}) ... p(x_2) p(x_1) \end{aligned}$$

This simplifies things greatly:

- Previously, each $p(x_k \mid x_{1:k-1})$ required $V^{k-1}(V-1)$ parameters to represent: each conditional probability vector has $V-1$ parameters, and there's V^{k-1} such vectors as there are V^{k-1} combinations of $k-1$ previous outcomes.
- But you don't actually *need* a separate table for each combination of past outcomes: independence means $p(x_k \mid x_{1:k-1}) = p(x_k)$. Each of the V^{k-1} vectors are the *same* vector with 2 outcomes, which means there's only $V - 1 = 1$ parameter here.
- Moreover, each of the X_i's are identically distributed (since it's the same coin), that is, they *all* share that 1 parameter.

Overall, we see that as the coin tosses are independent, one parameter is sufficient to express the entire joint probability for *any* of the 2^{30} possible sequences. In general, knowing (conditional) independence between variables helps greatly: if (some) of the previous variables don't change the probability, you don't need a separate conditional probability vector for each of those previous combinations of outcomes. Here, we cut $V^m - 1$ free parameters to $V - 1 = 2 - 1 = 1$ parameter.

[7] Note that here we explicitly subscript each p with the random variable it is of. This is to clearly demonstrate when we drop them in the third line. We do so as we integrate the knowledge all variables are identically distributed (and hence have the same p.)

B Preprocessor Regexes

The three-stage transform used in the text preprocessing from Chapter 2 can be fully implemented in the 30 lines of Python code below:

```
1  import regex as re
2
3  # Replacement function for re.sub, with the <capss> and <capse> tags.
4  def replace_caps_fn(m):
5      # Concatenate <capss> and <capse> tags with lowercase version of matched group.
6      return '<capss>' + m.group(1).lower() + '<capse>' + m.group(3)
7
8  # Main function to replace uppercase words, and insert tags.
9  def replace_caps(text):
10     # Add a space at the beginning of the text, if it doesn't start with a space.
11     if text[0] != ' ':
12         text = ' ' + text
13         add_bksp_start = True
14     else:
15         add_bksp_start = False
16
17     # Insert a '<bksp> ' for all words that are immediately preceded by punctuation.
18     text = re.sub(r'([^\x20\w])([^\s\p{P}\p{Nd}])', r'\1<bksp> \2', text)
19
20     # Regex pattern matching all spans on ALL CAPS text.
21     pattern = u'(\x20*\p{Lu}{2,}([\s\p{P}]*\p{Lu}{2,})*)([\s\p{P}])'
22     # Replace all caps span with tags + lowercase
23     text = re.sub(pattern, replace_caps_fn, text)
24
25     # Replace all caps letters with <shift> + lowered version.
26     text = re.sub(u'(\x20*[\p{Lu}])', lambda x: '<shift>' + x.group(1).lower(), text)
27
28     if add_bksp_start:
29         text = '<bksp>' + text
30     return text
```

The three regexes used here (lines 18, 21 and 26) enable extremely concise implementations of complex string transformation logic. We detail the exact transformations for completeness:

B.1 <bksp> transform

This is the transformation in line 18. Specifically:

```
# Insert a '<bksp> ' for all words that are immediately preceded by punctuation.
text = re.sub(r'([^\x20\w])([^\s\p{P}\p{Nd}])', r'\1<bksp> \2', text)
```

The goal here is to insert a `'<bksp> '` before all words that are not preceded by a whitespace, to enable consistent tokenization (avoiding the issue from Section 2.2.1). This happens via two "capture groups":

1. `([^\x20\w])`: This matches a single character, which must be neither a whitespace (`\x20`) or a word character (`\w`), with the `^` being the negation operator.
2. `([^\s\p{P}\p{Nd}])`: This also matches a single character, which must neither be a space character (`\s`), a punctuation character (`\p{P}`) nor a numeric digit(`\p{Nd}`); in practice this means primarily activating on words.

Once activated, we insert a `'<bksp> '` between the two capture groups. For instance, `' hello'` would not activate it, whereas `'(hello)'` would (as the `h` is preceded by a `(`, which is neither a whitespace or alphabetical character). This in turn, transforms `'(hello)'` into `'(<bksp> hello)'`.

B.2 <capss> and <capse> transform

This is the transformation beginning in line 21:

```
# Replacement function for re.sub, with the <capss> and <capse> tags.
def replace_caps_fn(m):
    # Concatenate <capss> and <capse> tags with lowercase version of matched group.
    return '<capss>' + m.group(1).lower() + '<capse>' + m.group(3)

def replace_caps(text):
    ...
    # Regex pattern matching all spans on ALL CAPS text.
    pattern = u'(\x20*\p{Lu}{2,}([\s\p{P}]*\p{Lu}{2,})*)([\s\p{P}])'
    # Replace all caps span with tags + lowercase
    text = re.sub(pattern, replace_caps_fn, text)
    ...
```

The goal here is to identify spans of capitalized text, and lowercase them while adding a `<capss>` and `<capse>` at the stard and end of the spans. This is achieved through the most sophisticated regex of the three. Going left to right:

1. `\x20*`: First match any number of whitespace characters (including potentially 0).

2. `\p{Lu}{2,}`: Match *at least* two uppercase characters. This is to avoid matching to simply words properly capitalized (where only the first character is a capital letter).
3. `([\s\p{P}]*\p{Lu}{2,})*)`: This is a complex capture group, so we break it down further. Of note is the `*` at the end, which means this subpattern can be matched to 0 or more times. Overall, it matches all the capitalized words beyond the first word, even if spaces/punctuation are present in between.
    a. `[\s\p{P}]*`: The first part of this subpattern matches to any number (0 or more) of whitespace (`\s`) *or* punctuation (`\p{P}`) characters.
    b. `\p{Lu}{2,})*`: The second part of this subpattern keeps matching further into the sequence if at least 2 uppercase characters are present.
4. `([\s\p{P}])`: This ensures the match always ends on a whitespace or punctuation character (instead of potentially "gripping" onto the next word if it happens to start with 2 or more uppercase characters, but has lowercase in between, such as `'MPSoC'` from Section 2.3.1).

Once activated, we insert a `<capss>` at the start, and `<capse>` at the end of the capitalized span of text, using the `replace_caps_fn`.

B.3 `<shift>` transform

This is the most straightforward: after the previous transform, the only capitalized characters left are ones that are not from a "capitalized span", and should have individual `<shift>`'s added before each. This is achieved via the following:

```
# Replace all caps letters with <shift> + lowered version.
text = re.sub(u'(\x20*[\p{Lu}])', lambda x: '<shift>' + x.group(1).lower(), text)
```

The regex has two simple components: 1. `\x20*`: As in the previous regexes, first match any number of whitespace characters (including potentially 0). 2. `\p{Lu}`: Then, match a single uppercase character.

Once activated, a `<shift>` is added before the character (but preserving any spaces prior to the start of the word). For instance, `' Hello'` is transformed into `'<shift> hello'`.

C Scaling Law Details

C.1 Pre-Chinchilla Scaling

Kaplan et al. (2020) had first established scaling law behaviors in large, neural language models two years prior to Chinchilla. This was an expansive work, covering many results that are very much worth learning about, such as:

- Transformers asymptotically outperforming LSTMs (as N grows larger), and the per-token loss going down across the context (whereas LSTMs plateau after 100 tokens), in Figure 7.
- Ablation between various model shapes (ratio between feedforward hidden dim and model dim, number of layers, attention head dimension) and model size, finding that for a fixed N, these parameters affect loss only mildly.
- Extending the literature on critical batch size (McCandlish et al. 2018) to Transformer language models (Section 5.1).
- Early work observing additional compute can be traded off for smaller model sizes (Figure 12).
- Observing a conspicuous lump at 10^{-5} PF-days at the transition between 1-layer to 2-layer networks (Figure 13), which subsequent work from Olsson et al. (2022) attributed to the formation of induction heads (which one-layer attention networks cannot form).

This work also fitted scaling laws between compute C, and model size N and number of tokens D. However, the estimates in this paper were $a \approx 0.73$ and $b \approx 0.27$; that is, $\log N$ needed to be scaled up ~3x faster than $\log D$:

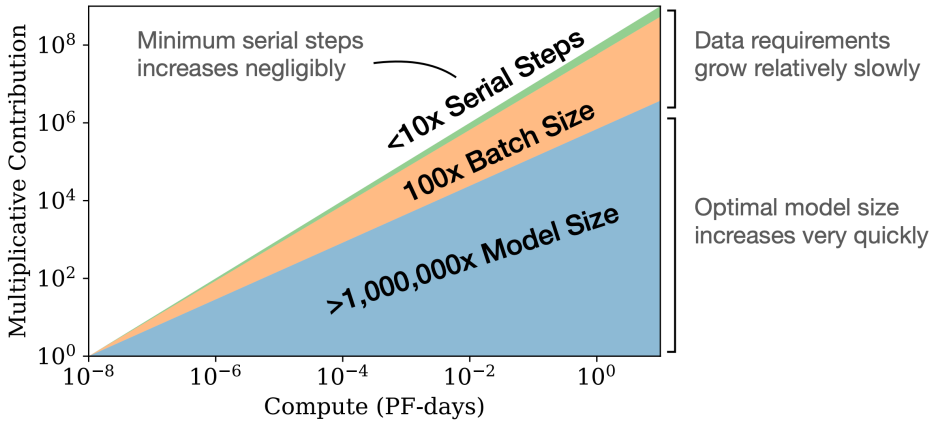


Figure C.1: Figure 3 from Kaplan et al. (2020)

However, these estimates were derived from the "early-stopped" loss (and the authors explicitly state it as such). That is, if a 1B model was trained on 100B tokens, to estimate the loss of a 1B model trained on 50B tokens, they would use the intermediate loss (at 50B tokens processed) from the 100B tokens run.

As the Chinchilla authors subsequently pointed out, using the intermediate loss value from a longer run means the learning rate schedule has not fully decayed at that point (as that happens at the end of the run). As they show in Figure A1, using a schedule with an endpoint more than 25% beyond the measurement point[1] leads to clear increases in the measured loss.

To correct for this, the Chinchilla authors trained *seperate* models of the same model size N for *each* value of D, making sure that the learning rate schedule fully finishes decaying when D tokens are processed. This corrected for the overestimated measured losses, yielding $a \approx 0.5$ and $b \approx 0.5$, yielding the now familiar 1:1 scaling ratio.

C.2 Deriving $C \approx 6ND$

A sketch for the $C \approx 6ND$ approximation introduced in Kaplan et al. (2020) is as follows:

1. Where N is the number of parameters, the total non-embedding FLOPs per token for the forward pass can be written out as[2].

$$\begin{aligned} C_{\text{forward}} &\approx 2 \cdot 2d_{\text{model}}n_{\text{layer}}(2d_{\text{attn}} + d_{\text{ff}}) + 2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}} \\ &= 2N + 2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}} \end{aligned}$$

2. C_{forward} can be described as having two terms: the first corresponding to model size, and the second corresponding to how increasing the context side n_{ctx} increases the number of FLOPs needed. For sufficiently large models with the context window sizes commonly used in training, $2N$ is ~2 orders of magnitude larger than $2n_{\text{layer}}n_{\text{ctx}}d_{\text{attn}}$. This allows simplifying C_{forward} to just the first term, $C_{\text{forward}} \approx 2N$[3].

3. Since this value is per token, for D tokens this is $C_{\text{forward}} \approx 2ND$.

4. Now, the backward pass takes 2x as much compute as the forward pass. This is because, at each layer you need to compute the gradients for *both* the activations of the previous layer *and* the weights of that layer. Hence,

$$\begin{aligned} C &\approx 2ND + 4ND \\ &= 6ND \end{aligned}$$

[1] In other words, a schedule that has not fully decayed at the measurement point.

[2] Note this leaves out the compute used for biases, nonlinearities and layer norms, which are a tiny fraction of total compute.

[3] For instance, for GPT-3 175B (with $n_{\text{layer}} = 96$, $n_{\text{ctx}} = 2048$ and $d_{\text{attn}} = 12288$), about ~98% of C_{forward} comes from the $2N$ term.