Training LLMs

Lecture 6

EN.705.743: ChatGPT from Scratch

Lecture Outline

- Recap of LLM model, with batch dimension
- Batching and Special Tokens
- Datasets and Training Hyperparameters
- Loss Curves and Perplexity

Recap of the Transformer Model

Tokenize the text

The text (as a string) is fed through the tokenizer, which splits words into individual characters and then follows a set of merge rules until the text is re-assembled as much as possible.

The resulting snippets ("tokens") correspond to entries in the vocabulary, and thus we can replace each token with an integer token id.

grokking

g r o k k i n g

Follow merge rules until:

gr ok king

Replace with vocab ids:

[4327][33412][12991]

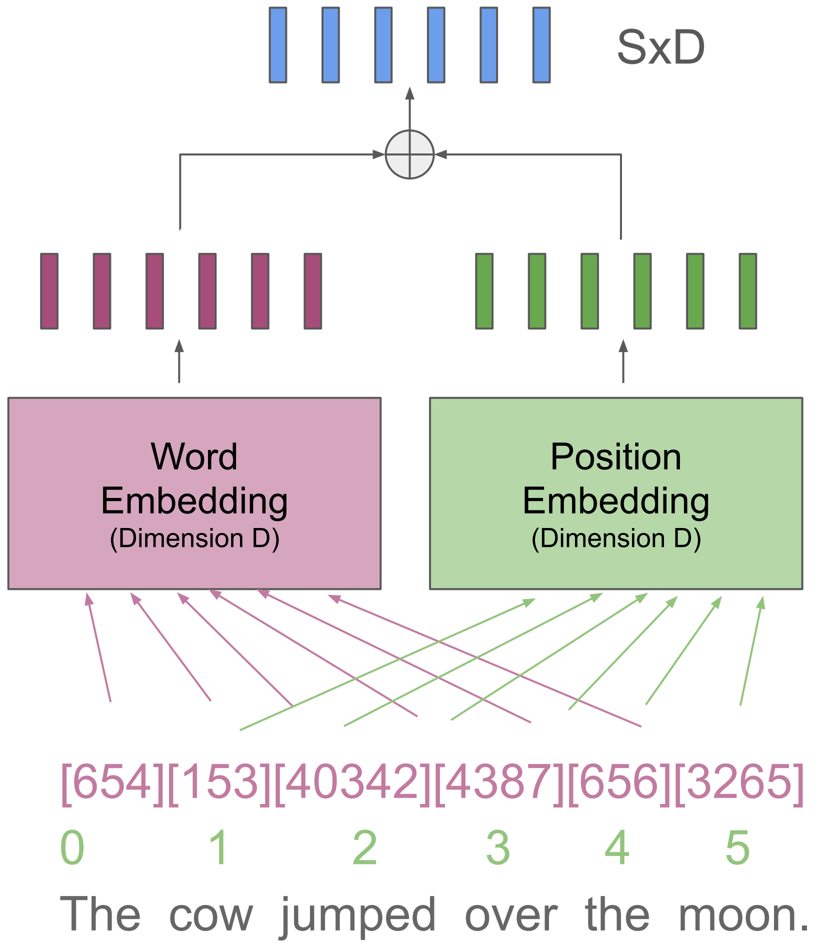
Embed the tokens

Each token id is fed through two embedding layers, which are updated during training:

(1) A word embedding, which represents the meaning of the token.

(2) A position embedding, which represents the position of the token in the sequence.

These embeddings each use dimension D. The embeddings are summed.


SxD
Word Embedding (Dimension D)
Position Embedding (Dimension D)
[654][153][40342][4387][656][3265]
0 1 2 3 4 5
The cow jumped over the moon.

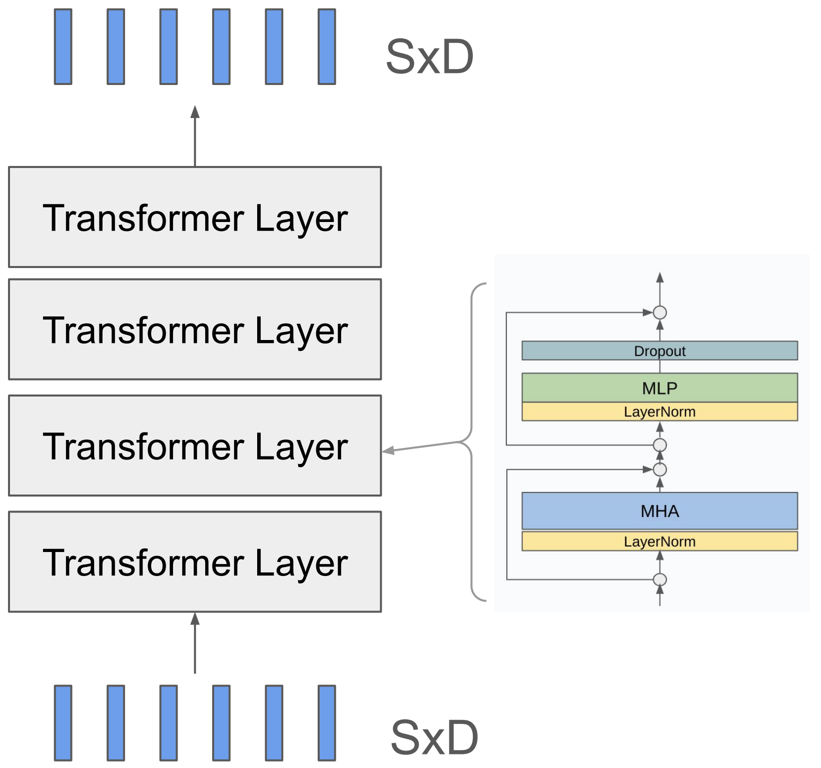
Pass Through Transformer

The S vectors of size D are fed through many transformer layers.

Each transformer layer consists of:

1) A LayerNorm
2) MHA (with residual connection)
3) A second LayerNorm
4) An MLP (with residual connection)

Only the MHA considers all vectors. The other operations are applied to each vector independently.

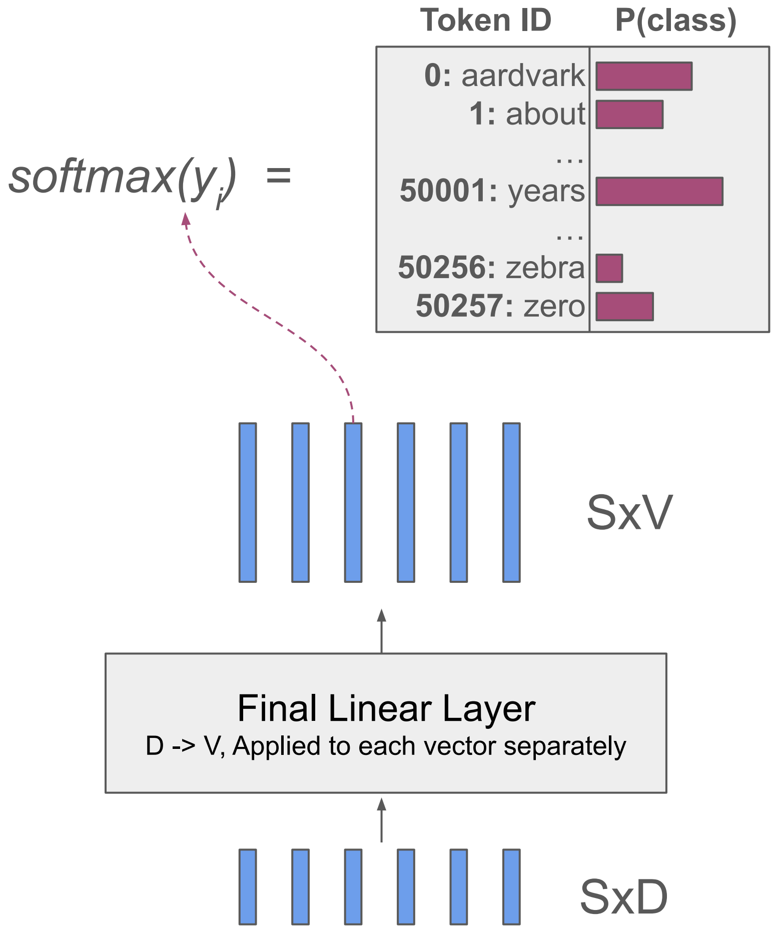


Project to Output

Each vector passes through a final linear layer which projects to the size of the vocabulary.

This is applied to each vector separately.

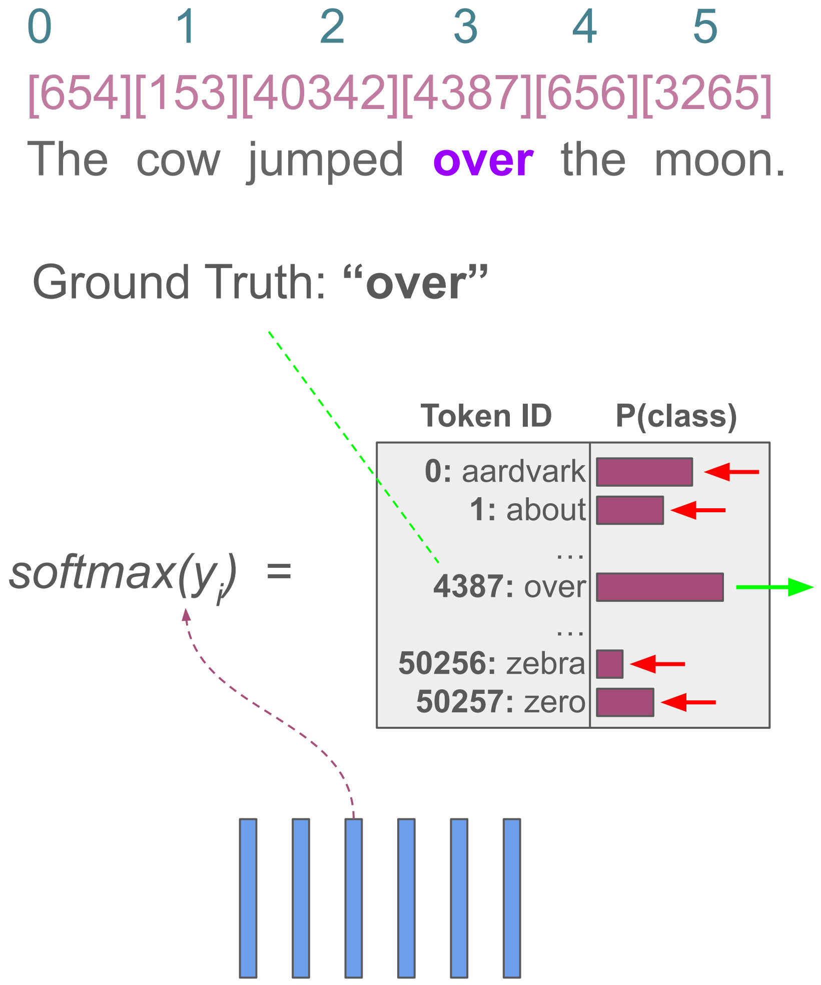
Applying a softmax to these vectors, they represent a classification prediction over all words in our vocabulary.



Loss

For each example, we know the ground truth for each word. We use this to compute a cross-entropy loss at every output. The final loss is the mean of losses at all outputs.

The output at location [i] should predict the word at location [i+1] in the input sequence.



Batching

Batching

As with most deep learning models, we actually train on batches of data in practice.

The input to the transformers is size (B, S, D), where B is the batch dimension, S is the sequence length, and D is the model dimension.

Note that some code expects (S, B, D) instead. If the first dimension is the batch as above, it is called “batch-first”.

Uniform Sequence Length

Batching presents a problem: our samples are all going to be of different lengths! Therefore, we cannot naively stack them into a batch.

Training Examples:

The cow jumped over the moon.	6 tokens.
Four score and seven years ago…	~300 tokens.
Chapter 1. Call me Ishmael…	~250,000 tokens.
…	

Uniform Sequence Length

There are two common solutions to this. Both are governed by our maximum sequence length (typically 2048 or 4096, mainly a matter of compute power).

The process of compiling multiple samples into a batch is called “*collating*” (the function or object that does this is called a “*collator*”).

Collator Option 1: One sample per entry

There are two common solutions to this. Both are governed by our maximum sequence length (typically 2048 or 4096, mainly a matter of compute power):

1) Truncate / Pad: If a sample is too short, we add “padding tokens” to make it long enough. If a sample is too long, we just truncate it.

Padding token: A special token that we add to our vocabulary which represents empty space.

Typically represented by <pad> or <|pad|>.

I love New York. <pad><pad><pad><pad><pad>	(9, padded)
Four score and seven years ago our fathers brought	(9, truncated)
The cow jumped over the moon.<pad><pad><pad>	(9, padded)
Call me Ishmael. Some years ago – never mind	(9, truncated)

This gives us a nice batch of size (4,9).

Special Tokens

A padding token is an example of a special token. These are tokens in our vocabulary which are not words, but instead convey something important to the model. Some common special tokens include:

<|bos|> : beginning of sequence, used to show that a new sample is starting.

<|eos|> : end of sequence, used to show that a sample is ending.

<|pad|> : padding token, used to note empty space.

These are typically placed at the very beginning or very end of the vocabulary. For example, GPT2's vocabulary ends with:

… [50254: informants], [50255: gazed], [50256: <|eos|>]

Notes about <pad>

Loss is not calculated for <pad>, because recurring padding tokens will dominate all other input patterns.

Imagine if we had a longer sequence length:

I love New York. <pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to
The cow jumped over the moon.<pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Call me Ishmael. Some years ago- never mind how long precisely- having little or no money in my purse, and nothing particular

Most of this consists of <pad><pad><pad>...

Notes about <pad>

Loss is not calculated for <pad>, because recurring padding tokens will dominate all other input patterns.

Imagine if we had a longer sequence length:

I love New York. <pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to
The cow jumped over the moon.<pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Call me Ishmael. Some years ago- never mind how long precisely- having little or no money in my purse, and nothing particular

This is also very inefficient! In the case above (mix of long and short samples), almost half our tokens do not contribute to the loss.

Collator Option 2: Packing Samples

There are two common solutions to this. Both are governed by our maximum sequence length (typically 2048 or 4096, mainly a matter of compute power):

1) Truncate / Pad: If a sample is too short, we add “padding tokens” to make it long enough. If a sample is too long, we just truncate it.

2) Packing: We consider our data as one long stream of text, and chop it into units of length max_sequence_length. This is called “packing”, because we “pack” multiple samples into one entry, if possible.

Collator Option 2: Packing Samples

To designate the start and end of samples, we use <bos>, <eos>, or both. The most efficient option is to just use <eos> between samples.

Option 1: One sample per entry.

I love New York. <pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in Liberty, and dedicated to
The cow jumped over the moon.<pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad><pad>
Call me Ishmael. Some years ago- never mind how long precisely- having little or no money in my purse, and nothing particular

Option 2: Packing

I love New York. <eos>Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived in
…
the people, for the people, shall not perish from the earth. <eos> The cow jumped over the moon. <eos> Call me Ishmael.
…

Multiple samples per entry.

Collator Option 2: Packing Samples

Other Notes:

- Packing is much more efficient if it is possible to implement it (for huge datasets it can be awkward).
- The model will learn to pay attention to <eos> by itself- we do not need to modify the model via a new attention mask or special loss functions.
- For our homework this week we will use a small dataset, so we will implement packing.

Subtle Point: The Last Entry

If we are packing samples, the very last entry (sequence) in our dataset will not be full.

To avoid having a single entry that is heavily padded, I recommend just discarding this last partial entry.

Tokenization vs Words (Reminder)

In reality, these methods are applied to **tokenized** text, I just used words in the previous slides because that is easier to look at. This sentence:

Godzilla loves New York. <pad><pad><pad><pad><pad>

Might actually be more than 4 words plus padding:

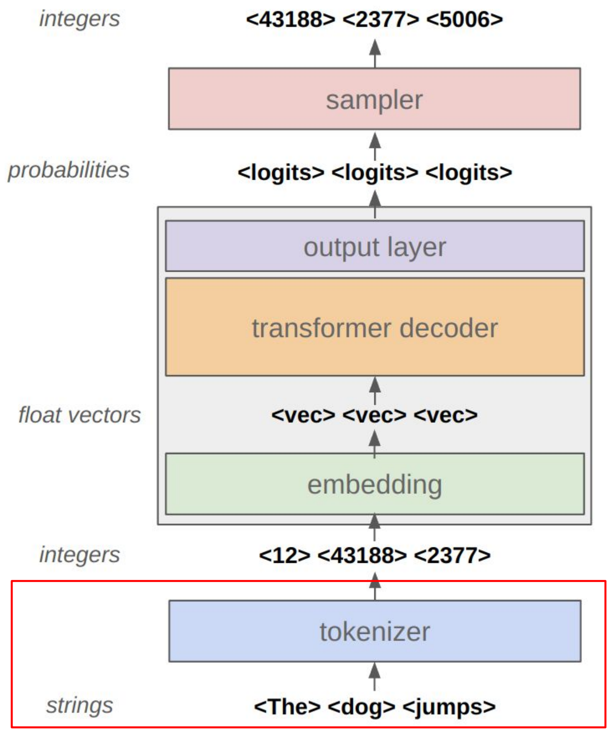
[443][1043][22243][5623][43111][45768][50256][50256][50256][50256][50256]

It is critical that our sequence length is uniform in terms of tokens, not necessarily in terms of words (one word is not always equal to one token).

Pre-Tokenization

A common practice is to tokenize the entire dataset ahead of training, to save time during training.

All samples are converted to token ids, so that for each pass we start with directly inputting sequences of integers into the model.



Can do this ahead of time.

Datasets

The rest of this lecture is a hodgepodge of notes and tricks about training models. I have tried to organize it somewhat but it's more or less a list of important tidbits.

Data (Very Big Data)

There are many datasets that are used to train LLMs, and most LLMs are actually trained on multiple datasets pulled together.

Data is usually measured in hundreds of billions or trillions of tokens.

It is very hard to comprehend how large this is. All of English Wikipedia is around 6 billion tokens. So 1T tokens is ~160 times larger than Wikipedia.

Common Datasets and Sources

General Web Crawl (Common Crawl, OpenWebText). These are low quality but can be filtered.

Wikipedia

Reddit

ArXiv

Stack Exchange

GitHub

Medicine and Law databases

Digitized Books (Books2, Books3, Project Gutenberg)

YouTube Subtitles

Aggregates

These datasets are often aggregated into large pooled datasets. The previous slide was the main contents of an open-source aggregate dataset called “The Pile”.

As of making these slides (March 2024), a popular aggregate dataset is “Dolma” by AllenAI, which has 3T tokens (approximately 200TB of raw text). Most of its contents seem to come from web crawls.

Aggregates

These datasets are often aggregated into large pooled datasets. The previous slide was the main contents of an open-source aggregate dataset called “The Pile”.

As of making these slides (March 2024), a popular aggregate dataset is “Dolma” by AllenAI, which has 3T tokens (approximately 200TB of raw text). Most of its contents seem to come from web crawls.

Update (April 18 2024) - LLaMA 3 released, and is trained on 15T tokens.

By my rough math, if you printed this out single-spaced it would require 22.5B pieces of paper, and stack roughly 1400 miles high into our upper atmosphere.

Assuming 6.5”x9” space per page (8.5x11 with 1-inch margins), this would cover ~328 square miles of contiguous printing if laid flat. For reference, Rhode Island is ~1200 square miles.

Considerations

When selecting training data, care may be taken to isolate a specific language (i.e. only English) or to balance different sources (if your LLM will be used as a coding assistant, it should see lots of code!).

Many giant LLMs probably just train on as much data as they can find, assuming it is not complete garbage.

There is also a growing trend to try the opposite- a few hundred billion high-quality tokens can be competitive with trillions of mediocre tokens. Microsoft’s Phi models are small LLMs that are trained exclusively on high-quality data (i.e. textbooks).

Huggingface Datasets

https://huggingface.co/datasets?task_categories=task_categories:text-generation&sort=trending

Model Size vs Data Quantity

You may think that smaller model = less data, but this is not true.

Even a small model (that might run on your own machine) will typically use 100s of billions of tokens.

This is simply a result of the fact that **more data = better model**, so if you have a lot of data you might as well train on it. More on this in Lecture 8 (Scaling).

Streaming Data

10s or 100s of TB of text presents some practical problems:

- Will often not fit on a typical hard drive (but, storage is cheap.)
- Will *definitely* not fit in RAM.

An interesting option is dataset streaming. Repeatedly:

- Pull down one batch of samples from a cloud storage system
- Do an update on the model
- Delete the batch

This is not always needed, but if you ever experiment with a large training effort I highly recommend it.

Note: You can stream HuggingFace data just by passing “streaming=True” to load_dataset().

Hyperparameters

Epochs

Epochs = 1

This is very different from most deep learning, and has some interesting implications:

- “Overfitting” kind of doesn’t apply, unless your data has duplicates or is imbalanced
- Don’t need to track validation loss

Sometimes, data is duplicated on purpose to re-balance, but it is very unusual for data to exist more than 2 or 3 times.

Optimizer

Use Adam or AdamW.

Adam is an extension of SGD that tracks two values for every parameter:

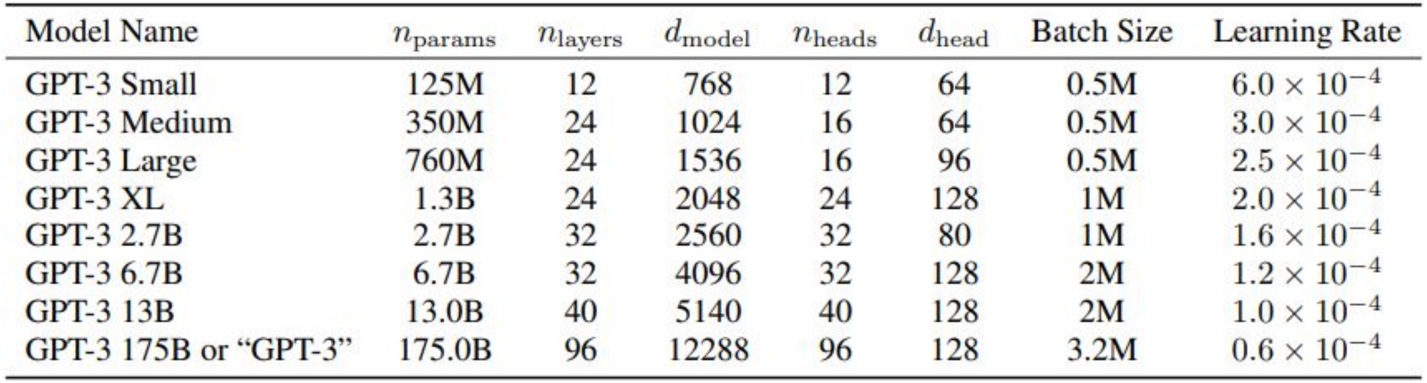
1) Moving average of the gradient of that parameter
    - This is used to apply momentum
2) Moving average of the square of the gradient of that parameter
    - This is used to smooth the magnitude of the gradient

This means that for a model with N parameters, we need to store an additional 2N terms of recent gradient information. For float32, each parameter takes 4 bytes. So a 1B parameter model needs at least 12GB of VRAM.

Note this is just for the model+optimizer to exist! Much more memory is needed to do forward and backward passes.

Learning Rates

Somewhat proportional to model size, according to GPT3 paper. Generally around [X]e^{-4}:



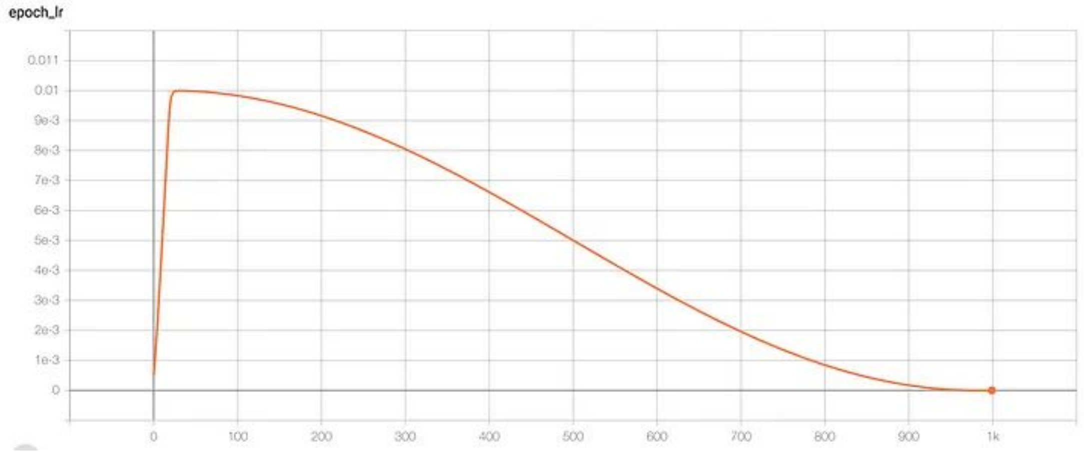
Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Weight decay is also used, something like 0.1 or 0.01. Sometimes this is ***not*** applied to biases, layer norms, and embeddings.

Learning Rate Schedules

Typically, the learning rate is first annealed from 0 to the maximum (warmup phase) so that the initial updates and the rolling averages learned by Adam are not overly biased towards the first few batches.

For the remainder of training, learning rate is annealed towards a small value. There are many ways to do this but a common one is cosine annealing:



https://scorrea92.medium.com/cosine-learning-rate-decay-e8b50aa455b

Batch Size

There are two ways to consider batch size: number of tokens, or number of sequences.

You can convert between the two with the model sequence length:

$$\textbf{batch\_size}_{\textbf{tokens}} = \textbf{S*batch\_size}_{\textbf{sequence}}$$

This can lead to some crazy-sounding values. GPT3-175 used a “batch size of 3.2M”. This really means 3.2M tokens. With a sequence length of 2048, this means about 1500 samples. This is still big, but a little bit easier to get your head around.

Weight Initialization

Not sure how critical this really is, but one technique is:

Init from Normal(0.0, 0.02) for all typical weights.

Init weights of LayerNorm to 1.0.

Init biases to zero.

Init the output projections W_o of MHA to Normal(0.0, 0.02/sqrt(2*num_layers))

Weight Tying

The word embeddings are size (V, D) and are meant to project an integer to a D-dimensional representation.

The output layer is meant to do the reverse, and also has weights of size (V, D).

Some implementations will tie these parameters together so that they reference the same parameters.

Gradient Clipping

Always a good idea to add gradient clipping:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
```

This scales the gradient if its magnitude (as a vector) is above some amount, in this case 1.0.

Gradient Accumulation

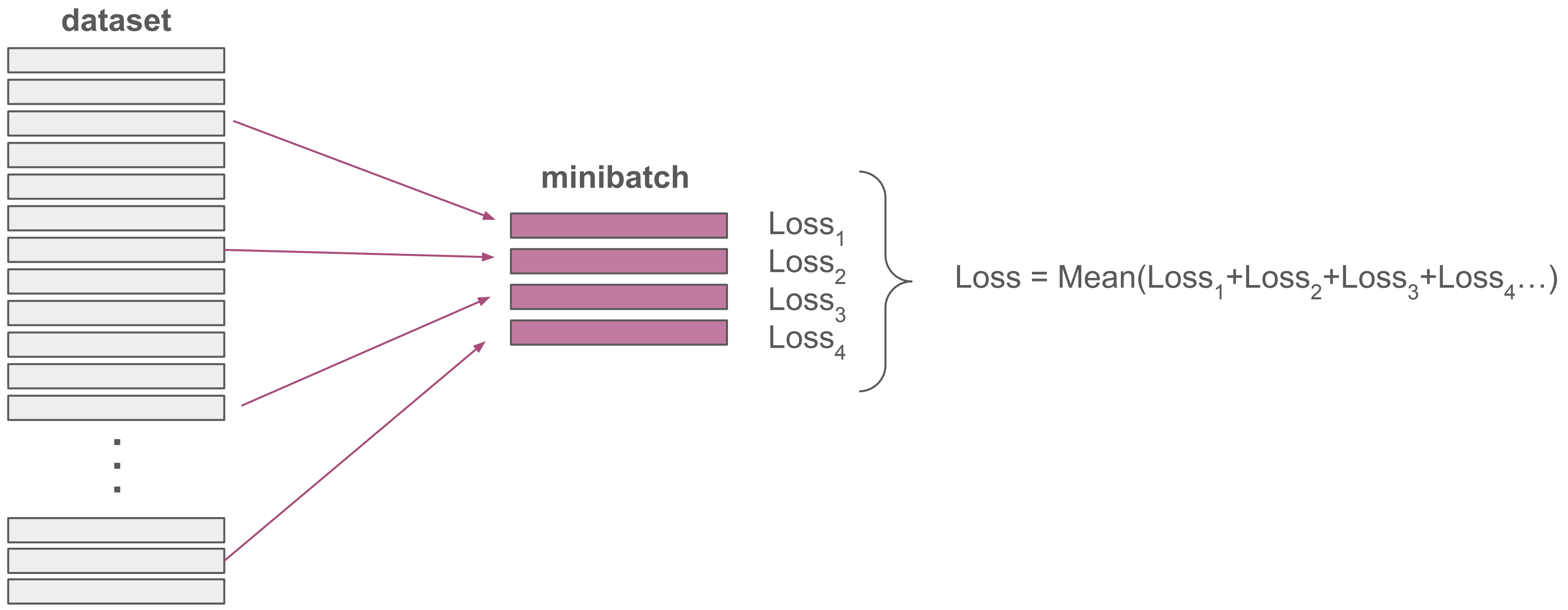
It can still be difficult to use batch sizes that are significant due to memory constraints. When working with a single GPU, you may find you can only accommodate a batch size of 2 or 4 or some otherwise tiny number.

This would probably be unstable.

Gradient Accumulation is used to make our effective batch size larger: we simply do multiple forward passes before backpropagating.

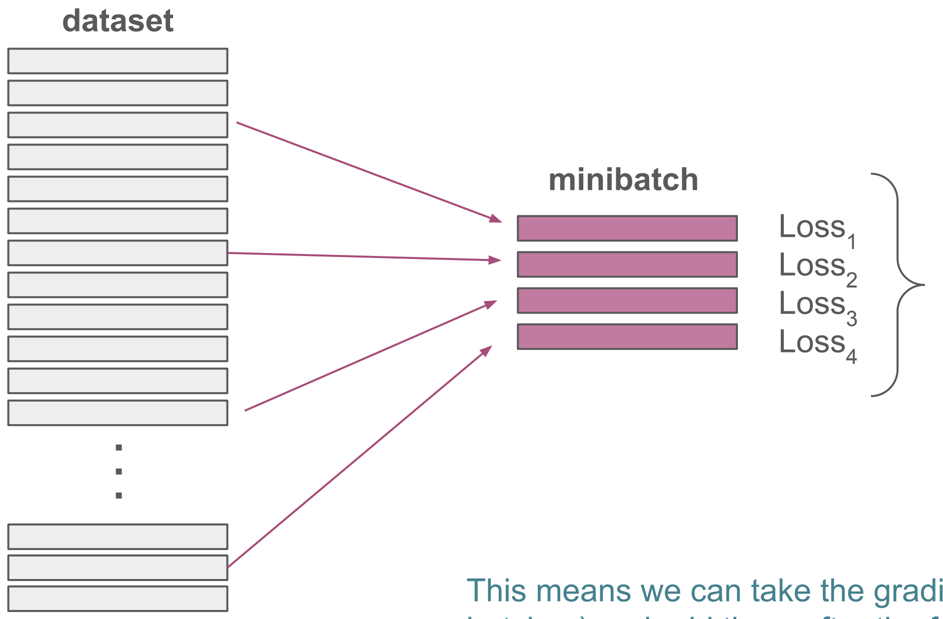
Gradient Accumulation

In standard minibatch SGD, we take a subsample of our data and compute the average loss over the samples:



Gradient Accumulation

In standard minibatch SGD, we take a subsample of our data and compute the average loss over the samples:



$Loss = Mean(Loss_1+Loss_2+Loss_3+Loss_4\ldots)$

$= (1/N)(Loss_1+Loss_2+Loss_3+\ldots Loss_N)$

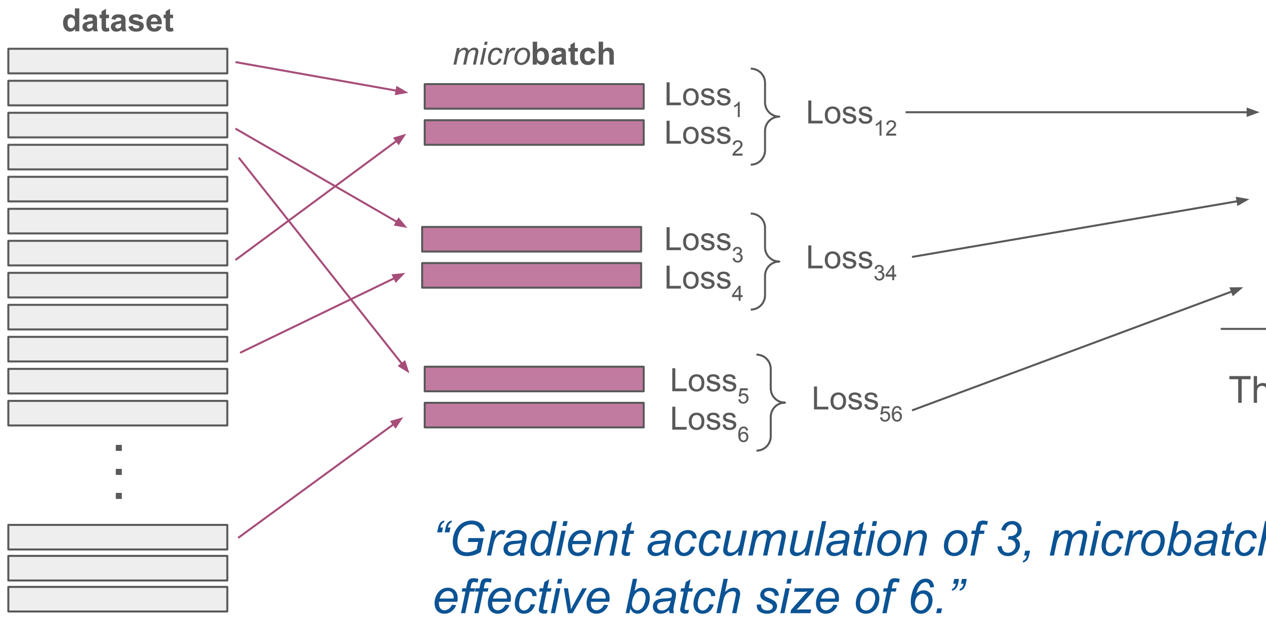
So:

$\nabla L = (1/N)(\nabla Loss_1+\nabla Loss_2+\nabla Loss_3+\ldots \nabla Loss_N)$

This means we can take the gradient of different samples (or batches) and add them after the fact.

Gradient Accumulation

In gradient accumulation, we split our minibatch into several microbatches, and sum (or average) the gradients from each one:



$\text{grad} = \nabla Loss_{12}$

$\text{grad} \mathrel{+}= \nabla Loss_{34}$

$\text{grad} \mathrel{+}= \nabla Loss_{56}$

Then we step the optimizer.

“Gradient accumulation of 3, microbatch size of 2, for an effective batch size of 6.”

In PyTorch

Gradient accumulation is really easy to do in practice. We just wait multiple batches before stepping the optimizer:

```
Standard:

for batch in dataset:
     opt.zero_grad()
     pred = model(batch)
     loss = lossfn(pred, target)
     loss.backward() # grad
     opt.step()
```

```
Accumulation every N steps:

for microbatch in dataset:
     pred = model(microbatch)
     loss = lossfn(pred, target)
     loss.backward() # grad

     if microbatch_index % N == 0:
          opt.step()
          opt.zero_grad()
```

Summary

Dataset: Some huge aggregation of internet data. Can stream it if needed.

Epochs = 1

Adam(W) Optimizer, Weight decay on weight matrices (not biases, norms, embeddings).

Learning rate = around 1e-4

Cosine Schedule with Warmup

If batches are small, use gradient accumulation. Aim for an effective batch size of something like 64, 128, etc. (Something large enough that you would expect it to work).

Use gradient clipping.

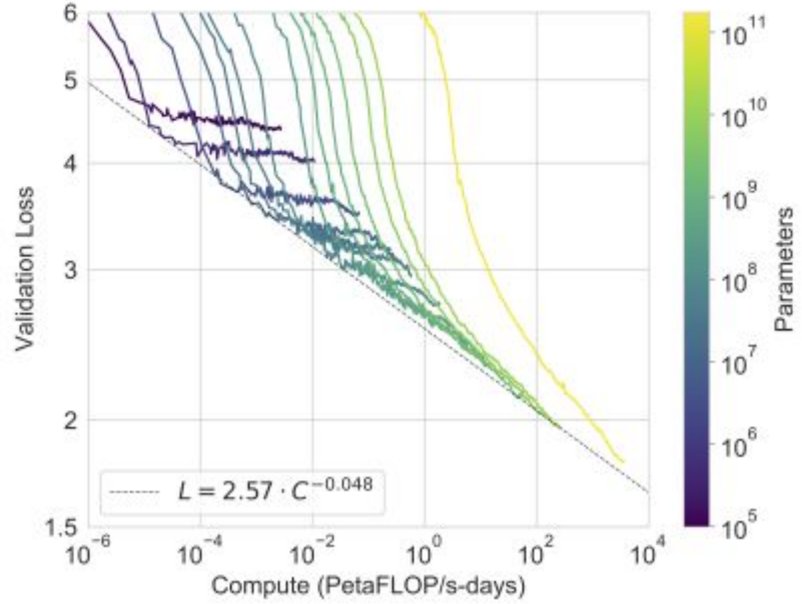
Initialize weights to some small random values, maybe from N(0.0, 0.02).

Loss Curves

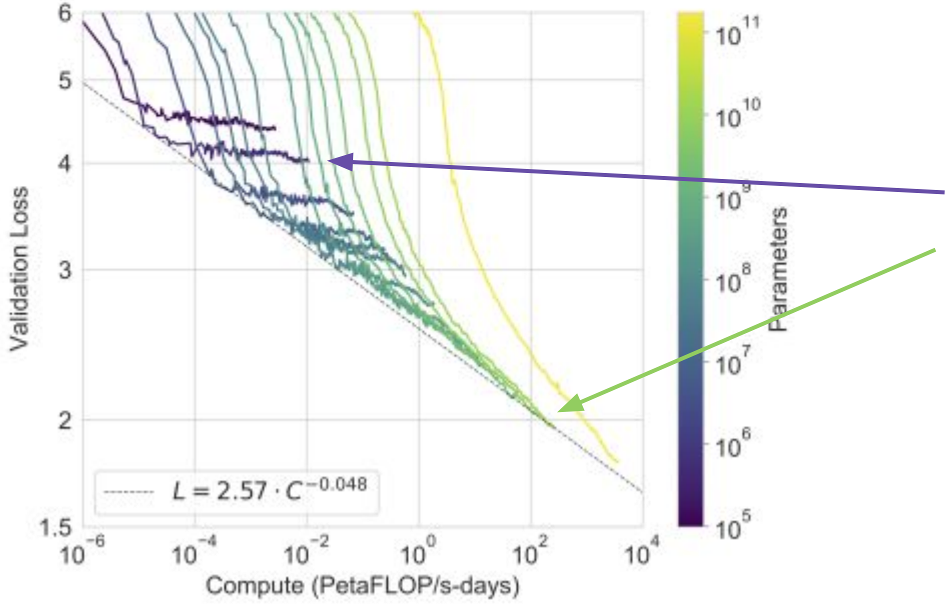
Is it working?

Since all LLMs share a similar objective and generally use similar data, their losses can (roughly) be compared to indicate performance. The final loss of a model is some function of dataset size, dataset quality, training time, and model size.

On the right is a set of loss curves from GPT3 variants, showing that they approach some predictable boundary (dotted line).



Is it working?

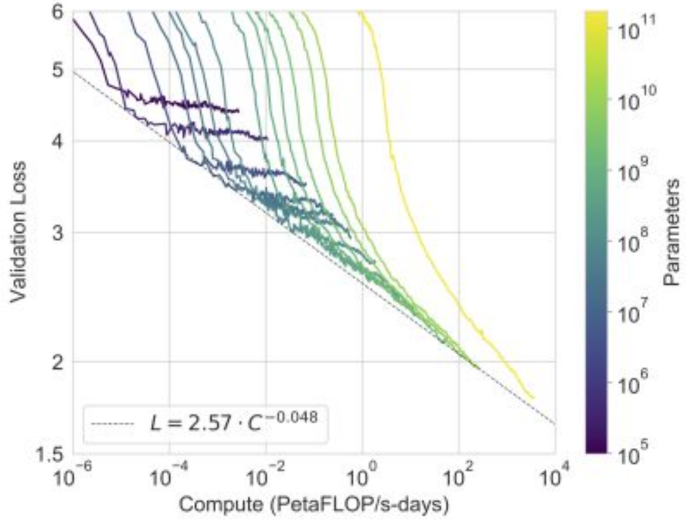


Note that the scales are logarithmic. We have a very bad case of diminishing returns.

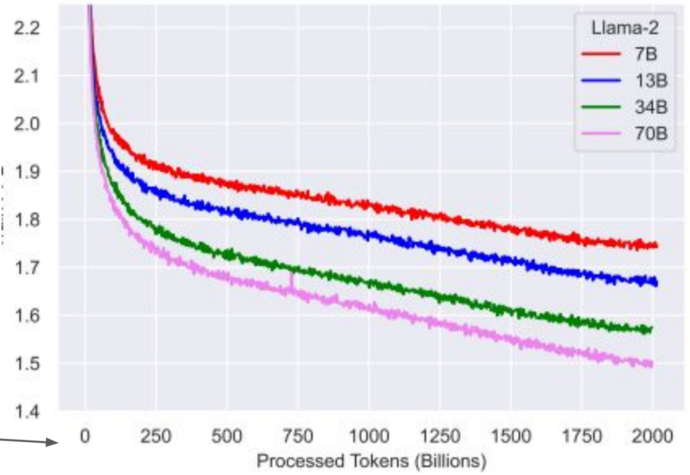
Comparing models with losses of 4 and 2:

The model that achieves a loss of 2 is maybe 1,000x larger and requires maybe 10,000x compute.

Is it working?



Generally, losses will be around 1 to 3. Here is another loss curve from the more recent LLaMA 2 models:



Since we only have 1 epoch, the x-axis is usually “tokens”

Why doesn’t the loss approach zero?

Note that the loss does not approach 0, it approaches some constant C.

This is the inherent entropy of language, which will always cause training error. For example, if we chose the next word for the phrase:

The man goes to the ____.

There are plenty of valid options- store, meeting, doctor, event, party, etc… Some are more likely than others, but we will never be able to predict correctly against a known ground truth example.

Note that *our model can represent this distribution correctly* (probabilistic), but it will never translate into zero error since we have a ground truth (deterministic).

Perplexity

A related metric is “perplexity”, which is directly related to loss:

Perplexity = exp(Loss)

Loss of 1 = perplexity of 2.71

Loss of 2 = perplexity of 7.39

Loss of 3 = perplexity of 20.09

When reading an LLM paper or blog you may see this metric instead.

Evaluations and Benchmarks

We can also test a model against a set of standardized questions which may indicate performance on certain types of inputs or certain topics.

These range from simple (true/false) questions to multiple choice, math questions, trivia questions, etc.

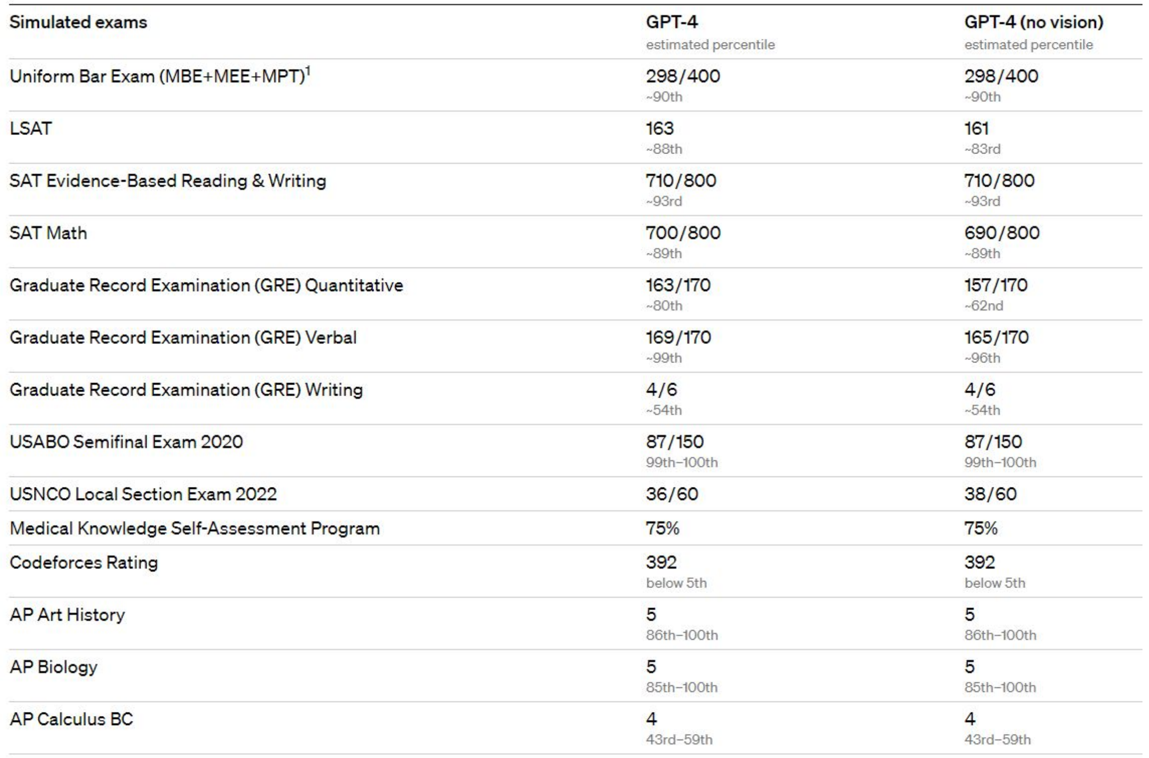
Like datasets, these have become aggregated so that we have “benchmarks made of benchmarks”.

MMLU is an interesting one to examine, which splits questions into various subject matter:
https://paperswithcode.com/sota/multi-task-language-understanding-on-mmlu

Using Human Tests

Another interesting option (and one that gets lots of press) is to use existing tests.

GPT4 did this upon its release.



Simulated exams	GPT-4 estimated percentile	GPT-4 (no vision) estimated percentile
Uniform Bar Exam (MBE+MEE+MPT)[1]	298/400 ~90th	298/400 ~90th
LSAT	163 ~88th	161 ~83rd
SAT Evidence-Based Reading & Writing	710/800 ~93rd	710/800 ~93rd
SAT Math	700/800 ~89th	690/800 ~89th
Graduate Record Examination (GRE) Quantitative	163/170 ~80th	157/170 ~62nd
Graduate Record Examination (GRE) Verbal	169/170 ~99th	165/170 ~96th
Graduate Record Examination (GRE) Writing	4/6 ~54th	4/6 ~54th
USABO Semifinal Exam 2020	87/150 99th–100th	87/150 99th–100th
USNCO Local Section Exam 2022	36/60	38/60
Medical Knowledge Self-Assessment Program	75%	75%
Codeforces Rating	392 below 5th	392 below 5th
AP Art History	5 86th–100th	5 86th–100th
AP Biology	5 85th–100th	5 85th–100th
AP Calculus BC	4 43rd–59th	4 43rd–59th

Ask People

Ultimately, once a language model reaches a certain level of competency (correct grammar, spelling, etc), measuring “how good it is” can be subjective. What makes one piece of writing better than another?

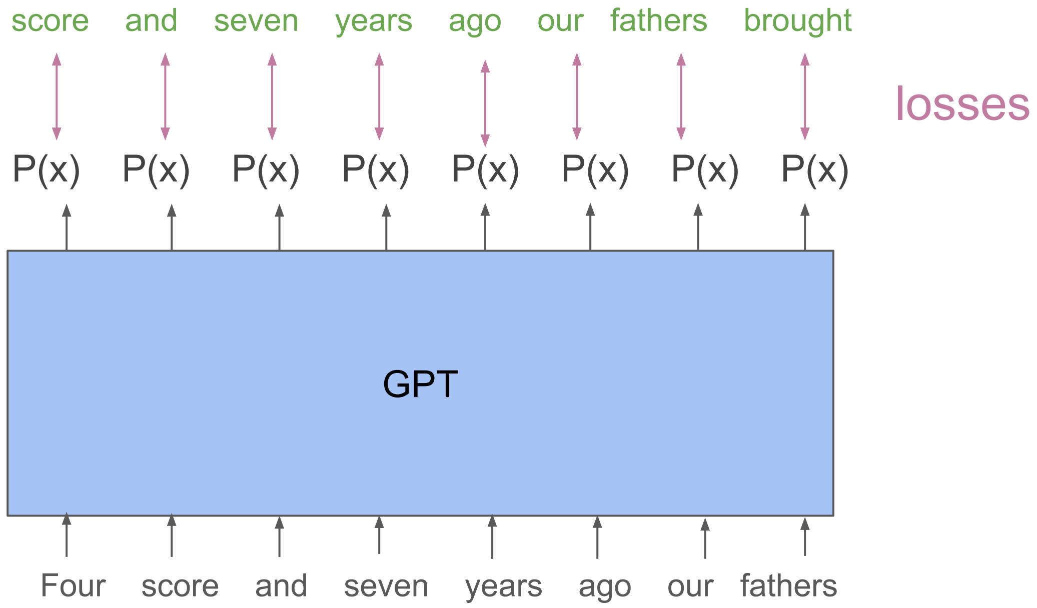
An expensive but very insightful method is to simply ask a group of people what they think of various outputs, or to compare outputs (which output is better?).

This is the basis of a training extension called RLHF, which we will cover in a few lectures.

Concrete Implementation Tips

Losses at each output

Remember, we calculate a loss at every output position. Every token is a training target (except the first) and every token is an input (except the last):



Targets = Inputs Shifted by 1

To enable this behavior, we re-use the inputs as our target outputs, but shifted by one position! Pseudocode for our inner training loop might look like:

```
batch = get_next_batch() # size (B, S+1)

inputs = batch[:, 0:seq_len]   # first S tokens, size (B, S)
target = batch[:, 1:seq_len+1] # last S tokens, size (B, S)

pred = model(inputs) # size (B, S, vocab_size)
loss = cross_entropy(pred, target) # this will average over everything
```

Targets = Inputs Shifted by 1

To enable this behavior, we re-use the inputs as our target outputs, but shifted by one position! Pseudocode for our inner training loop might look like:

```
batch = get_next_batch() # size (B, S+1)

inputs = batch[:, 0:seq_len]   # first S tokens, size (B, S)
target = batch[:, 1:seq_len+1] # last S tokens, size (B, S)

pred = model(inputs) # size (B, S, vocab_size)
loss = cross_entropy(pred, target) # this will average over everything
```

This means when prepping our dataset, we need to pack into sequences of S+1, not S.

These are integers i.e. torch.long.

Pred is floats (logits).

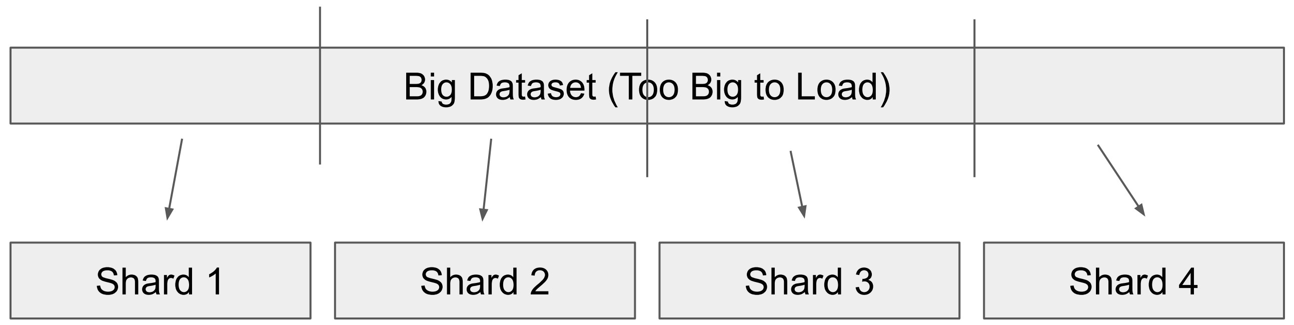
You can use cross entropy directly from PyTorch. Look at the documentation on how it works.

Brief Study on Sharding and Shuffling

Shards

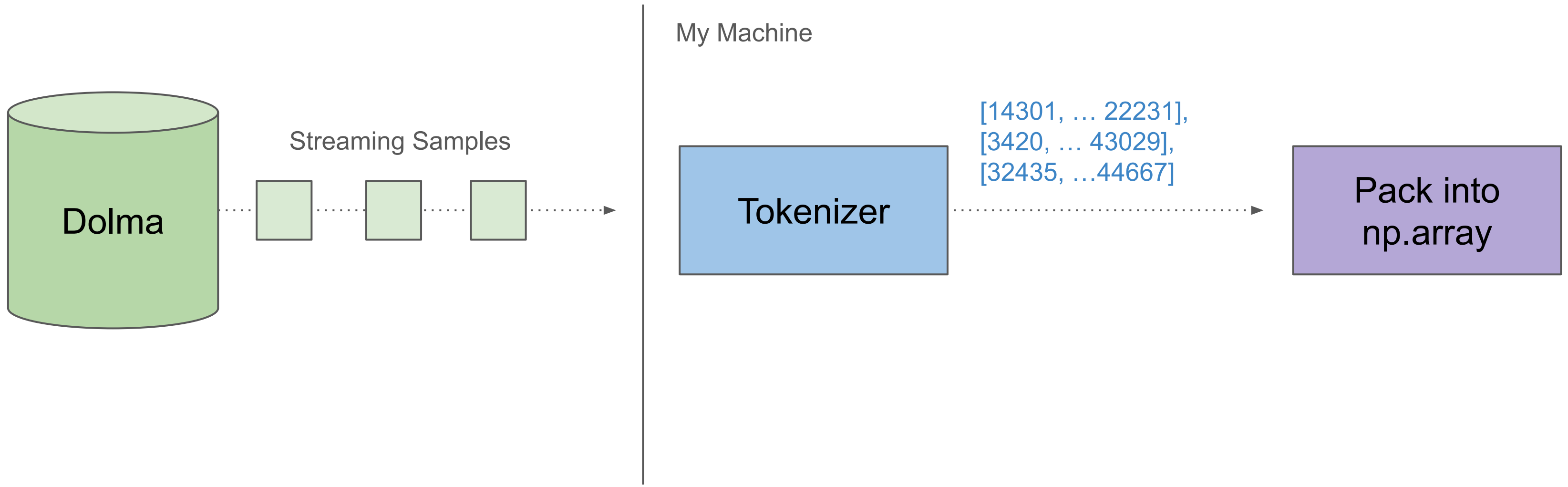
This is not relevant for homework but an interesting case I ran into last week (7/1).

Large datasets are often broken into “shards”- subsets of the dataset that can be loaded individually (so you don’t need to hold a huge dataset in memory):



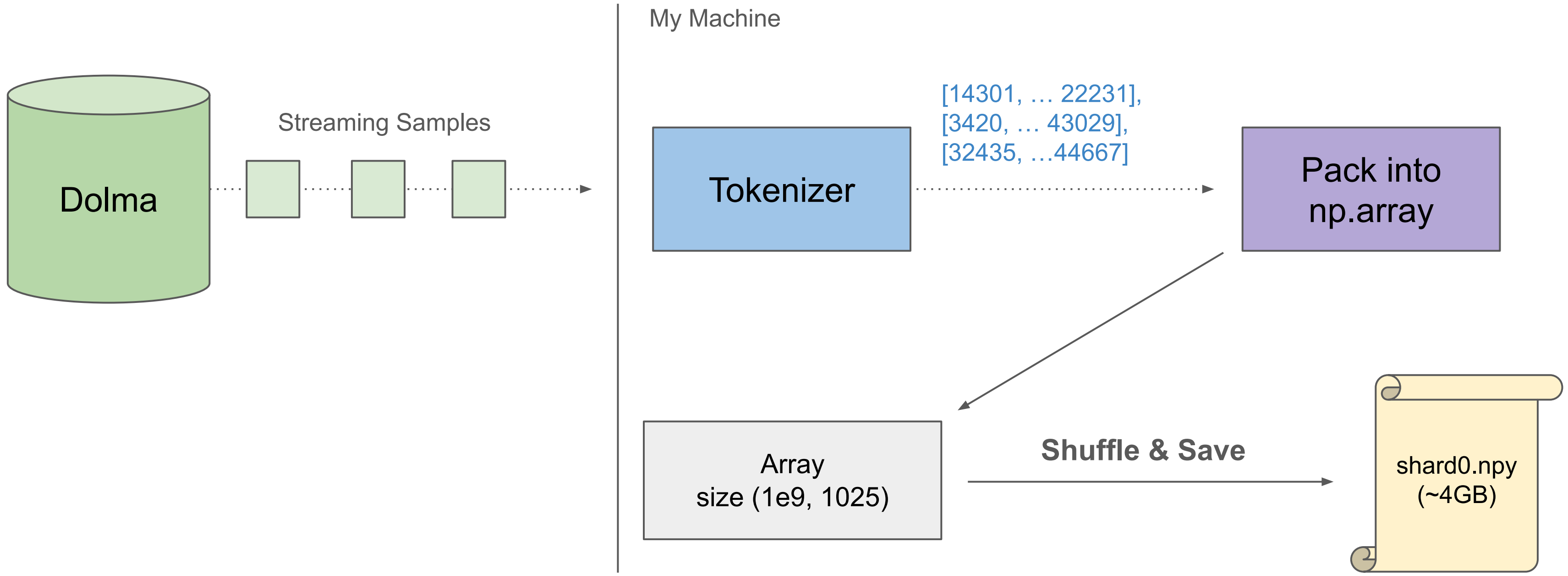
My Sharding

I streamed the first 10B tokens from Dolma, tokenized them, and stored in 10 shards of 1B tokens each.



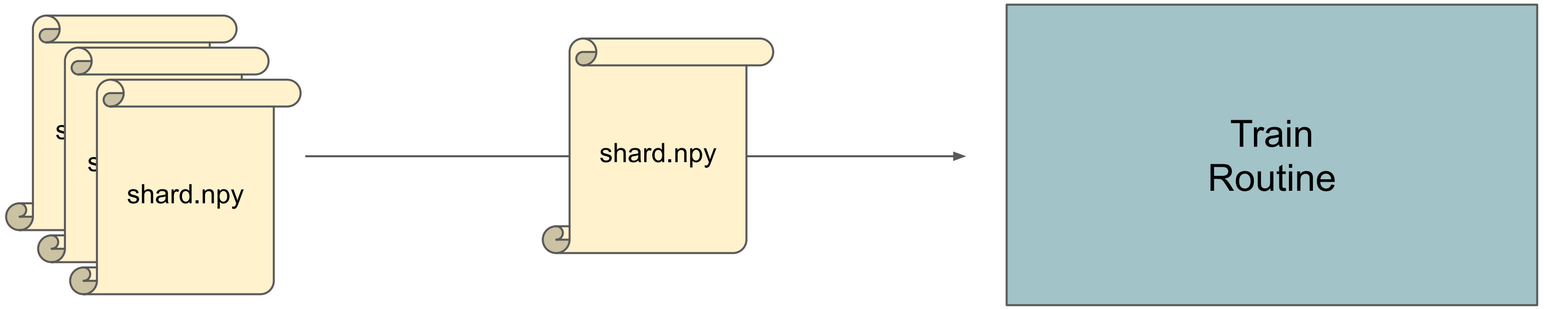
My Sharding

I streamed the first 10B tokens from Dolma, tokenized them, and stored in 10 shards of 1B tokens each.



Training

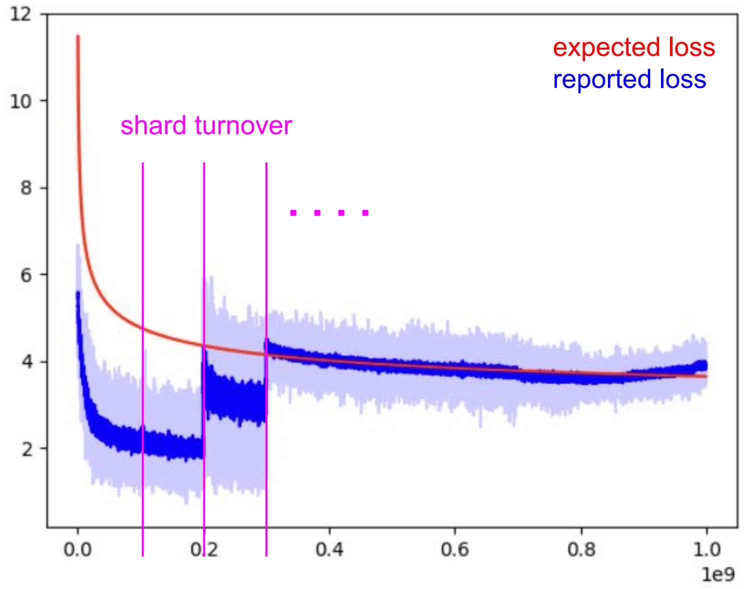
I trained on one shard at a time, loading them in sequentially.



Wonky Loss Curve

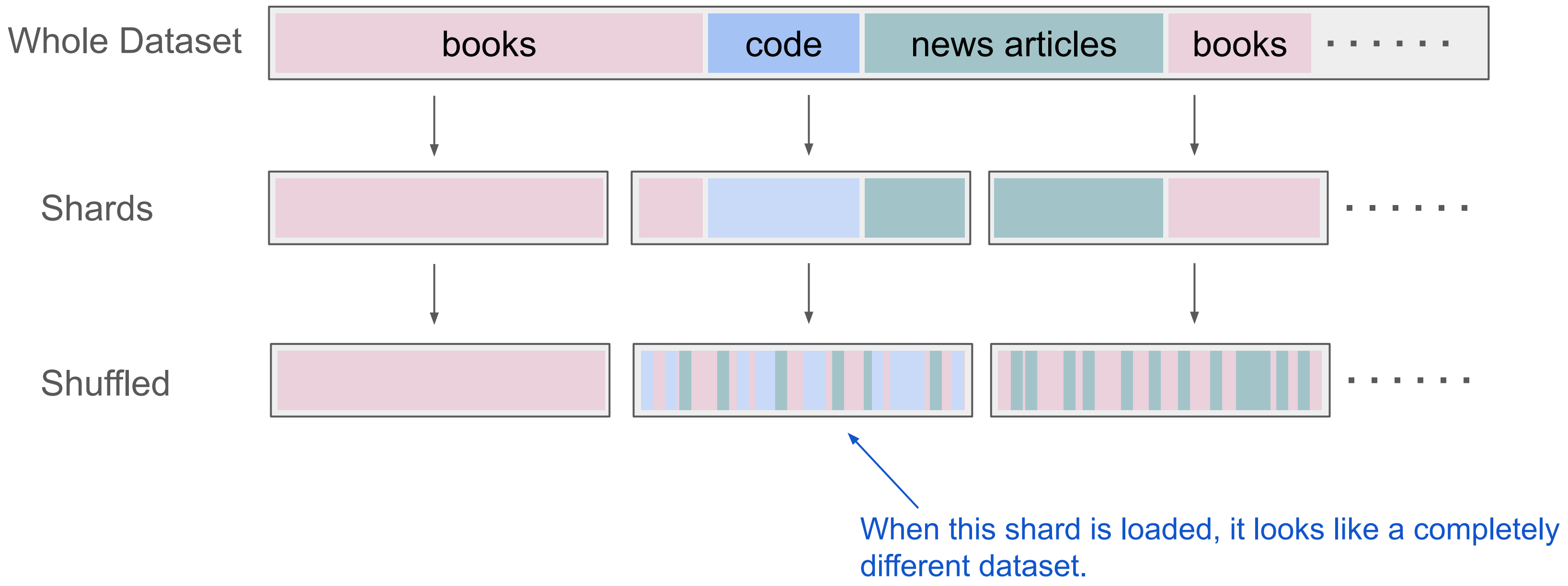
The loss curve changed drastically when the shards were changed out.

(This is a reproduction using shards of 100M instead of 1B):



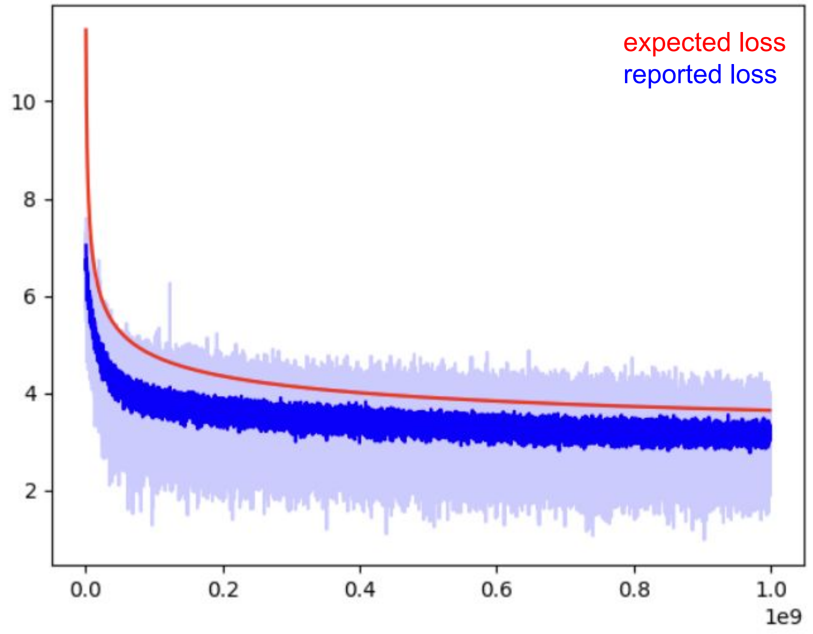
Shuffling…

It turns out that shuffling each shard independently was not enough! Imagine our dataset has several different types of data:


Whole Dataset
books
code
news articles
books
Shards
Shuffled
When this shard is loaded, it looks like a completely different dataset.

Complete Shuffling

After shuffling all the shards together, the problem is resolved:



Note: The expected loss is very rough so this is not an improvement on state of art or anything. The red line just helps see if I am in the right ballpark.

Review Assignments