CSE 493S/599S HW2

Matthew Montelaro, Patrick Wyrod, Bolong Zheng, Anderson Lee

June 2023

1 Data Processing

All data is sourced from The Pile, [1] a publicly-available collection of datasets. [1] Its stated purpose is for training large language models in addition to serving as a benchmark for cross-domain knowledge and generalization ability of models. This ensures our model is being trained on diverse and high-quality data, allowing us to make generalizable inferences and conclusions through our experimentation and ablation study.

Prior to input, the data used for training, testing, and validation are stored in separate directories to ensure no tainting occurs during ingestion. All hardware constraint arguments are defined as global variables. The data-processing procedure for each dataset is as follows:

0. **Tokenizer:** Prior to data processing, a trained SentencePiece tokenizer [2] is used to create a Tokenizer instance.

1. **Ingestion:** Raw data is extracted sequentially from a JSON file into a list. Each element is a dictionary with keys 'text' and 'meta'. The function allows an argument to cap the number of entries read. The metadata is deemed irrelevant for this task and hence discarded in the following step. (See Fig. 1)

2. **Dataset:** A PyTorch Dataset is initialized using the data list and tokenizer. Its __getitem__ method extracts all 'text', encodes it using the tokenizer, and truncates it if it exceeds a global length restriction. It then returns inputs and targets determined by the tokenizer's bos and eos values.

3. **Collation:** While the Dataset controls sequence overflow, the collation function handles underflow. The default configuration is 0-padding. Other techniques are explored in the ablation study.

[1] https://the-eye.eu/public/AI/pile/
[2] https://static.us.edusercontent.com/files/rJIgocw097NO80lycWdfN5wi

Figure 1: Properties of an ingested sample

```
examine_data(train_dataset.data)

First 50 chars: ---
abstract: 'We categorify the notion of an infi
Input type: <class 'list'>
First element type: <class 'dict'>
Keys: dict_keys(['text', 'meta'])
```

Figure 2: Example of batch used in training

```
{'input_ids': tensor([[    1,   660, 29901,  ...,     0,     0,     0],
...
[    1, 11474,    13,  ...,   278, 12151,  2264]]),
'target_ids': tensor([[  660, 29901,    13,  ...,     0,     0,     0],
...
[11474,    13, 16595,  ..., 12151,  2264,     2]])}
```

4. **Dataloader:** Finally, the Dataset and collation function instantiate the Dataloader. Shuffling is enabled for generalizability (with the exception of the validation set); all other behaviour is default. (See Fig. 2)

2 Codebase Modifications

The following modification were implemented for training:

model.py

1. All mentions of `init_method=lambda x: x` were removed in favour of the default method of each fairscale function calling it, which were observed to be more performant.
2. All mentions of `self.cache` (found in the Attention class' `__init__` and `forward`) were removed. Unlike inference, training requires multiple backward passes, which is an invalid operation for cached PyTorch tensors.
3. The Transformer class' forward method was originally configured for inference. For training, `torch.inference_mode()` was removed and the output slicing (`output = self.output(h[:, -1, :])`) was altered output all logits instead of only the last ones.

tokenizer.py

1. The default configuration inherits the SentencePiece model's pad_id; this was replaced with `self.pad_id: int = 0`. Our tokenizer's default padding of -1 triggered nonnegativity assertions within the model; 0 was chosen as the override value as it is not used for any other encoding.

3 Experiments

3.1 Epoch Frenzy

Setup

We noticed that the full LLaMA model was trained for a fairly small number of epochs (around 0.5-2.5) on various different data sources, so we thought to evaluate how training for many epochs over a smaller amount of data would affect the model. The setup was done on a NVIDIA GeForce RTX 2080 SUPER. It was run with the following hyperparameters: a dimension of 512, 8 layers, 8 heads, and a max sequence length of 256. The batch size was 10. The first two experiments took about the same amount of time to train, around 20 minutes. The third took longer, about 50 minutes because of frequently calculating validation error, but would be around the same amount of compute if validation was not calculated so frequently (This was found to be true on accident due to originally calculating validation error at the wrong time). The results are plotted in the following section.

Results

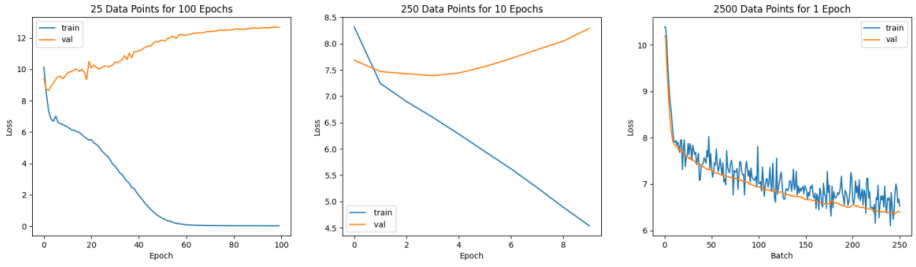


Figure 3: Training and validation error for multiple amounts of training data with inversely related amounts of epochs [A]

Analysis

It is clear from this experiment that equivalent compute time with more epochs, which will require training on fewer data points, is not an effective alternative

to training on a large data set for a small number of epochs. When trained for 100 epochs on 25 data points, the validation error fairly consistently went up while training error decreased to be very near zero. The 250 data points for 10 epochs performed better on validation, as its validation error at its highest is lower than the validation error of the previous at its lowest, but its training error did not get quite as low. Over the course of one epoch, the model trained on 2500 data points consistently lowered its validation error to lower than either of the other two models reached, while not getting its training error as low. The two models with fewer data points thus seem to be overfitting their small data set and lowering their training loss while their validation loss is not as low and even increases over more epochs. Overall, this experiment reinforced the practice that is standard with LLMs, feeding in a very large amount of data with a smaller amount of epochs.

3.2 Hyperparameter Gridsearch

Setup

We searched over 36 hyperparameter configurations trained on 2500 sequences for 1 epoch. The validation set consists of 300 sequences. The search space is demonstrated in table 1. The setup for this experiment is on M1 Macbook Pro CPU because M1 GPU is not supported for `torch.view_as_complex` in the training code and we didn't find a substitution for this function. The training time for each configuration varies because each hyperparameter contributes to some amount of training time. The total training time is 1035 minutes with batch size of 16 to prevent from abusing memory.

Hyperparameter	Search Space
dim	128, 512
n_layers	2, 8
n_heads	2, 4, 8
max_seq_len	32, 128, 512

Table 1: *Hyperparameter Search Space*

Learning Curve

Figure 1 shows the effects of `dim` where the first row has `dim = 128` and the second row has `dim = 512`. We can clearly see that, with more dimensions, the learning curve converges faster as shown in sharp drop in loss to the left part of the learning curve. This appears throughout all configurations with `dim = 512` compared to `dim = 128`.

Figure 2 shows the effects of `n_heads` plotted in the same fashion as the previous part, where each row has a fixed `n_heads`. We didn't find any significant difference in between the rows. The possible reason we speculated might be the setup involves insufficient number of sequences to show the difference.

Figure 3 shows the effects of `max_seq_len`. A noticeable result is that as `max_seq_len` increases, the fluctuation of learning curve decreases after convergence. Namely, the model is more stabilized when trained on a larger `max_seq_len`. This is intuitive because a model trained on a smaller number of `max_seq_len` is more likely to lose the whole information of the training sequence. In addition, this reduces it ability to capture context and thus to generalize and stabilize well when encountering other data.

Figure 4 shows the effects of `n_layers`. We didn't find a significant difference between the two rows. As model complexity increases, namely `n_layers` increases, we assume it is more likely to make a difference when trained with more data.

In conclusion, we found that `dim` helped convergence in training and `max_seq_len` was beneficial to reduce fluctuation in training loss and stabilize.

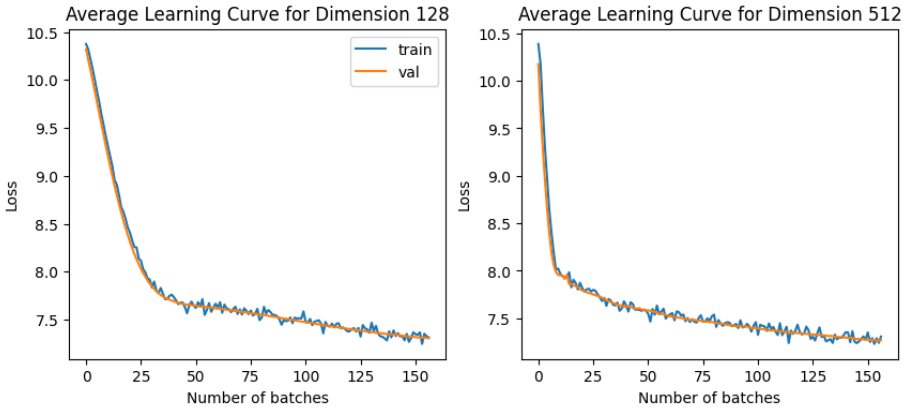


Figure 4: Learning curve y-axis by dim. Only n_heads=8 is shown to reduce plot size

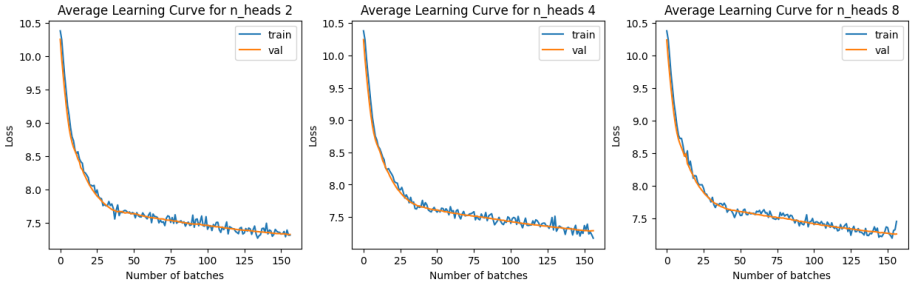


Figure 5: Learning curve y-axis by n_heads. Only max_seq_len=512 is shown to reduce plot size

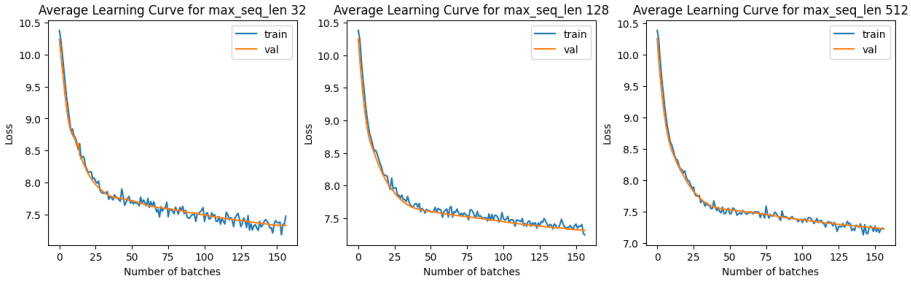


Figure 6: Learning curve y-axis by max_seq_len. Only n_heads=8 is shown to reduce plot size

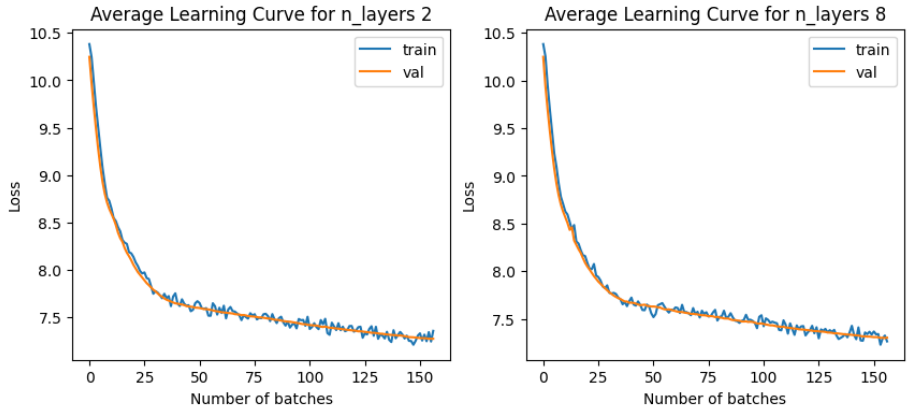


Figure 7: Learning curve y-axis by n_layers. Only max_seq_len=512 is shown to reduce plot size

Prompt Result

None of the configurations has a meaningful prompt results. We didn't find noticeable difference in any aspect of the prompt result to compare with different configurations. However, we did see some clue of learning. For instance, the prompt goes "The meaning of life is". The model trained with `dim=512, n_layers=2, n_heads=8, max_seq_len=512` generates " of9,, The,1enst byter inul W his 673 in7 yearsl and supp the of6 the v the unknown." Although it seemed meaningless, it did show some keywords about life, which are "years" and "unknown." To get a more concrete result, a language model still demands training on larger number of data so that we can compare the difference between different configurations. Due to the hardware limitation, we were not able to perform such experiments.

Scaling Up Best Model

Setup

Scaling is generally believed to be the key to language modeling. Although we don't have enough computational resource to scale up the model to a mature point, we were able to feasibly scale up the model to compare with the hyperparameter search models. We trained the model on 27000 sequences for 1 epoch with batch size of 16 on M1 Macbook Pro CPU for roughly 7 hours. The hyperparameters for the model is the maximum of the search space. The learning curve is shown in figure 8.

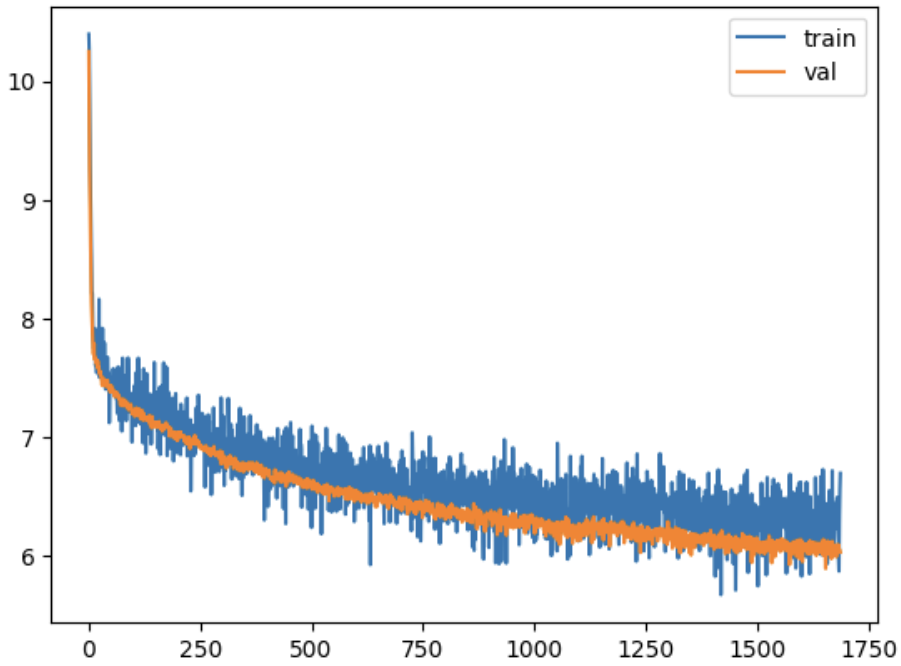


Figure 8: Learning Curve for Scaling Up Best Model

Initial Result

The validation loss on the first 10000 sequences in the validation set provided based on `CrossEntropyLoss` was 6.19. Using the same prompt, "The meaning of life is," the model generates "100 in8,,, it still home Im the over1 because together you the in 0 back the off.. Ive for." Indeed, the result is not prominent and at all meaningful, but we can see the model's ability to construct sentences. For instance, the use of "because" between two segments of the sentence or any meaningful usage of concatenation words didn't appear in the prompt results

from hyperparameter search models. This tiny version of scaling showed a promise of scaling in a limited hardware setup.

Conditional Prompt

Since the data we trained the best model on is `02.jsonl.zst`, which is largely related to computer programming. We evaluated our best model under the condition of code-related prompts. The results are shown in figure 9. We noticed that our model was able to complete the curly braces in both the JavaScript and Java prompts. Besides, it was capable of appending semicolon at the end of Java code. In the JavaScipt result(top-left), the model creates another function but without meaningful structure afterward. Finally, in the HTML result(bottom-right), we observed multiple tags structure including `href`. Overall, the model showed achievement in learning code-related language tasks.

Unconditional Prompt

To examine the ability of our model to handle general prompts unconditionally, we saw from the initial result that the model was not able to produce meaningful results. We further prompted two more sequences where the first one was completely in general fashion and the second one was in between general and technical fashion shown in figure 10. We observed that the first one showed some meaningful result, especially that the homework is going to be "read." Secondly, the prompt between general and technical context didn't seem to deliver evidence of achievements of the task.

4 Ablation Study: Sequence Length Handling

Background

Since the hypothetical non-Eucldiean tensor defined by sequences of differing lengths creates an intractable problem, uniformity with respect to a fixed size must be enforced. However, not all sequences are of the same length. This ablation endeavours to explore the considerations and implications surrounding the options to impose this constraint.

If we take a look at the distribution of text lengths in our sampled data, the lengths vary wildly and over several orders of magnitude 11. Although ideally, to preserve information in the training data, we would want to use all texts without any truncation, this is computationally infeasible due to time and space constraints.

A few techniques for this are discussed below:

1. **Padding:** One common underflow technique is to pad shorter sequences with a special padding token (such as 0) until they reach a predefined

```
================================
Prompt:
function () => {
 x = x + 2 ret
================================
Result:
function () => {
 x = x + 2 rets
 }
function*() 0
 the common to4 is in1,121 -580791
 is remainder
================================

================================
Prompt:
public static int sum(int a, int b) {
         return
================================
Result:
public static int sum(int a, int b) {
         return1 ;  ;   }
        x2 21 5 37
0
 b     0

================================
```

```
================================
Prompt:
def sum(a, b):
         c = a + b
================================
Result:
def sum(a, b):
         c = a + b 1 ( -3 -)
mh() * =1 +12 * +23 assumingve
3722 -
================================

================================
Prompt:
<h>Heading1
================================
Result:
<h>Heading183p#6c0
<>-=":"//--"><  href../"="____"> B4/"
 <
================================
```

Figure 9: Conditional Prompt Results

```
================================
Prompt:
My homework is going to be
================================
Result:
My homework is going to be' read
So' something' have like a phone have grateful
 my- I to a of in?
The we to the time happen on I over
================================
================================
Prompt:
Simply put graph theory,
================================
Result:
Simply put graph theory, ., your, and the time.
This- imagesising is as as as as example and theling of dynamic., need store be to by or
================================
```

Figure 10: Unconditional Prompt Results

maximum sequence length. The loss function computation is adjusted to ignore these token.

2. **Truncation:** This is an overflow technique in which sequences that exceed the maximum sequence length are simply truncated to fit. Whether to discard or impute the trailing tokens presents a dilemma: information is inherently lost, but the resources spent on a imputing the tokens as a new sequence are diverted from computing a new one. Further, the artificial break in continuity may impact the usefulness of these sequences relative to unaltered ones.

Setup

The data ingested is a fixed, controlled variable consisting of the first 128 samples of Pile trainset 22. This is chosen to minimize computational overhead while remaining representative of the dataset as a whole, as can be inferred from the apparent Poisson convergence in 11. Several model arguments are also fixed, as shown in 2. Complexity is gradually introduced as the ablation study progresses.

Fixed variables	Values
data source	Pile train 22
samples	128
`model_args`	
dim	512
n_layers	8
n_heads	8
norm_eps	1e-5
max_batch_size	8

Table 2: *Overview of controlled variables*

4.1 Naive underflow handling

The first two ablations examine the model's behaviour without explicit underflow handing. To do so, we define max_len to be the length of the shortest ingested sequence. The trivial case of omitting entire sequences whose length does not precisely match max_len is not considered.

Responding variables	Discard	Impute
seq_len	39	39
num_batches	16	915

Table 3: *Resultant inputs for naive underflow handling*

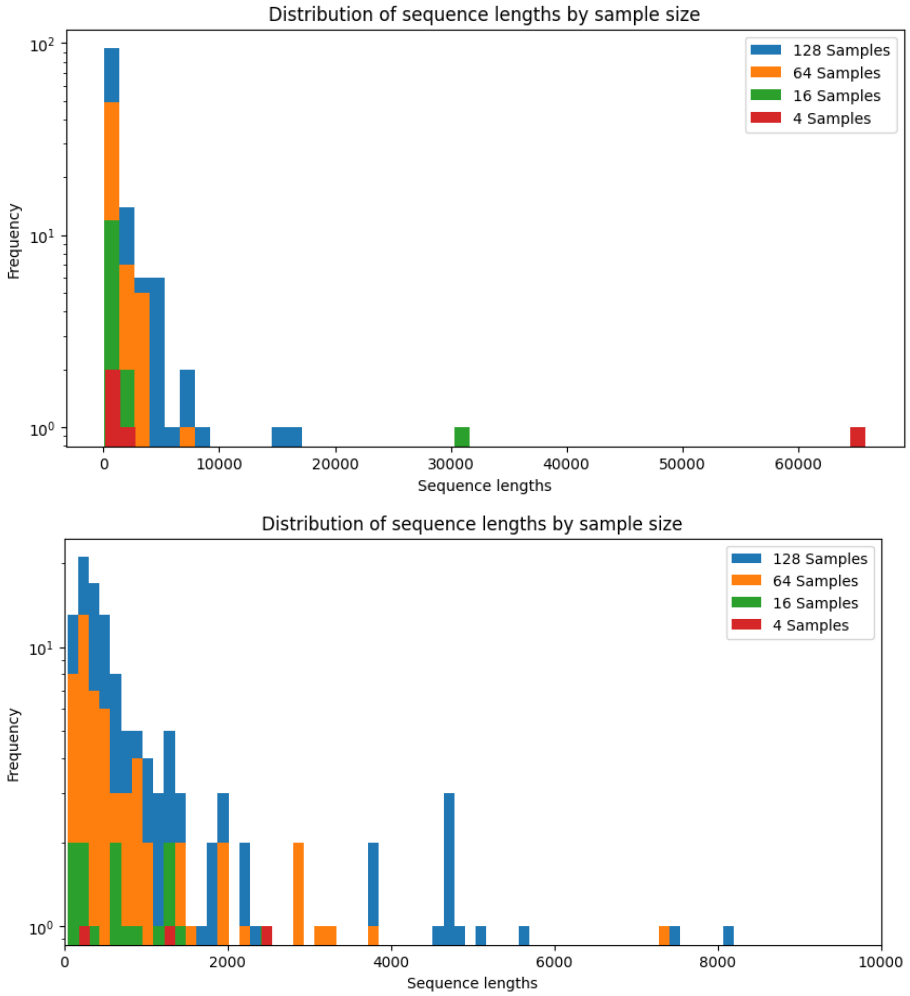


Figure 11: The Poisson-like distribution of sequence lengths across 2 scales

Discard only

The simplest nontrivial approach is to truncate sequences at max_len and indiscriminately discard them. This naturally results in few batches and a large proportion of data is lost.

Impute only

The next step up is a naive "snaking" approach: truncated values form the beginning of the next sequence, which is in turn filled to min_seq_len using the following subsequence. This process is repeated until all sequences are depleted,

when the final sequence is discarded if it is not of length min_seq_len.

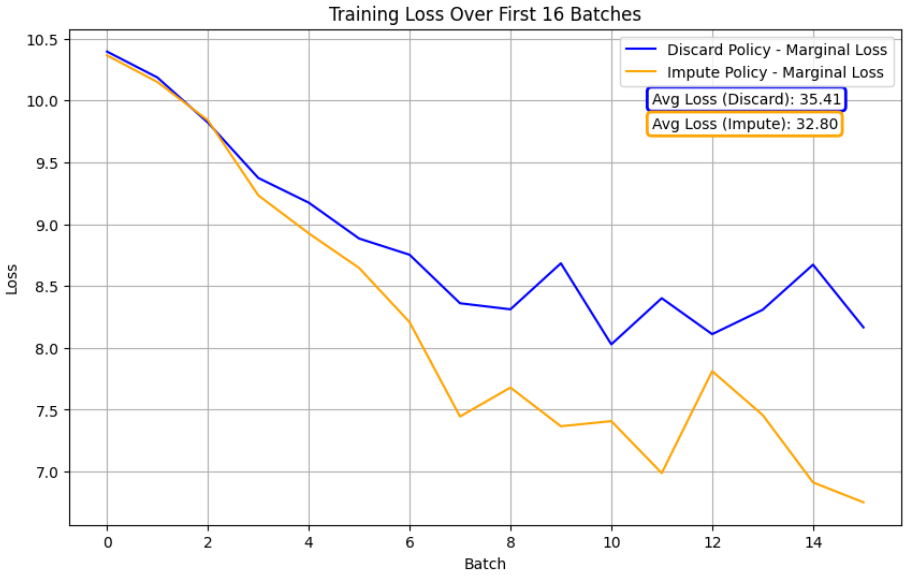


Figure 12: Training loss over first 16 epochs for overflow-only policies

4.2 Naive padding

Now we introduce explicit underflow handling through padding. A new, artificially-defined parameter max_len parameter replacing the previous data-determined one is introduced. Rather than discarding tokens all tokens beyond the smallest sequence length, the underflow is compensated with a specific padding value which the loss function is configured to ignore. These experiments are conducted using 0 as our tokenizer does not use this value for encoding.

Padding with discard-only

First, after applying 0-padding where necessary, we consider discarding all sequences beyond max_len.

Padding with imputation-only

Now we consider imputing all sequences beyond max_len as temporary "overflow" sequences. Since we now have an underflow mechanism, these can be imputed as new sequences altogether, thus removing the artificial continuity of the naive imputation in the first round of ablations.

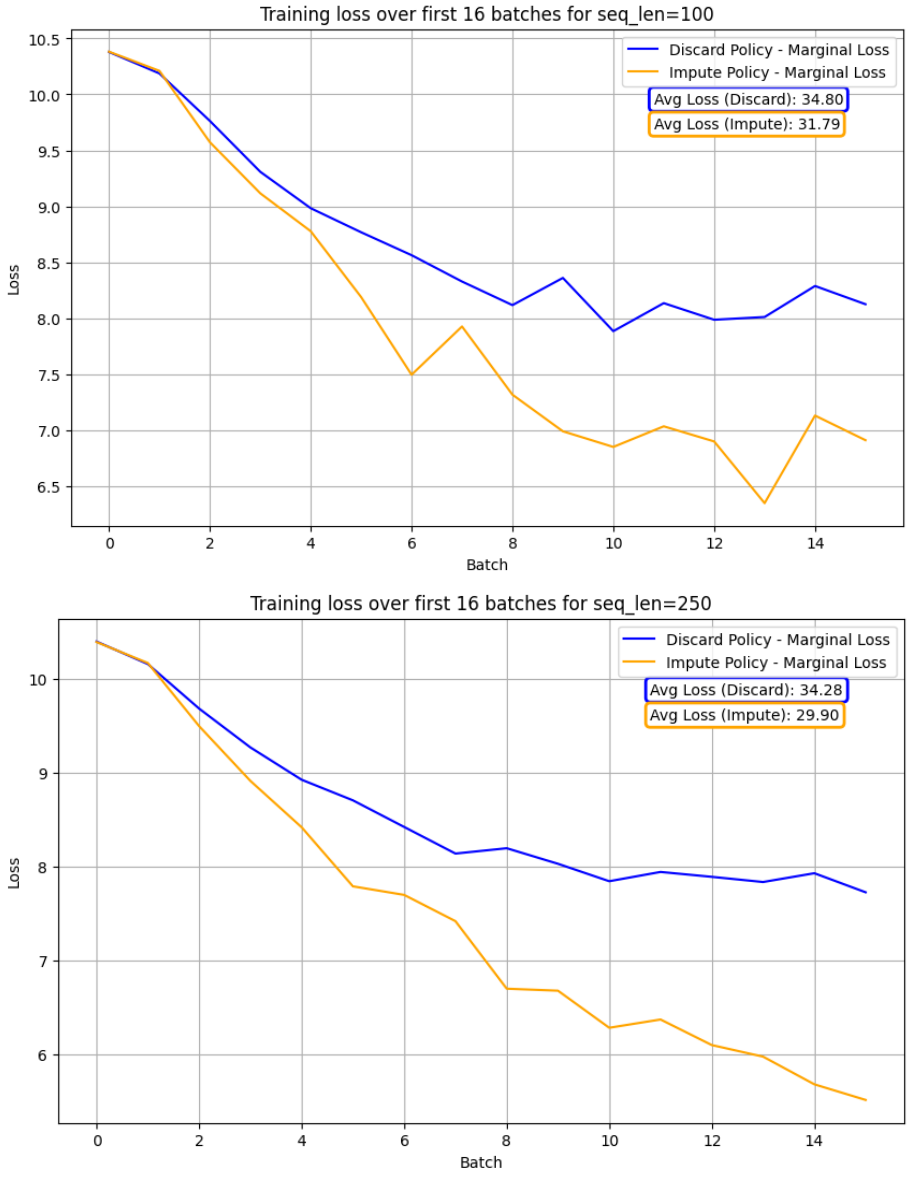


Figure 13: Training loss over first 16 epochs for the the overflow policies with padding at lengths 100 and 250

4.3 Informed padding

We now explore a more sophisticated approach taking into account properties of any given truncated "overflow" sequence to determine which overflow policy

to dispath on a case-by-case basis.

Length overflow policy

This approach determines whether to discard or impute the tokens exceeding based on their length: a sequence is discarded if it is shorter than the predetermined threshold and imputed otherwise.

Overflow tol.	num_batches	data_length	loss over max(num_batches, 16)
50	9	72	20.62
100	16	124	34.40
150	25	194	34.72
200	30	240	34.84

Table 4: *Responding variables with respect to 4 overflow tolerances with max_len fixed at 100. Note that the first entry did not meet the 16 batch threshold for direct comparison*

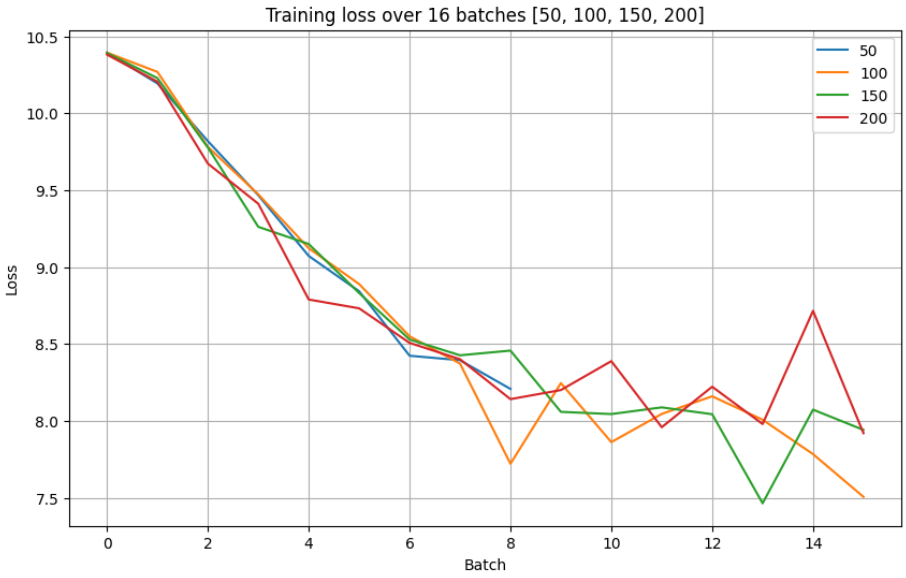


Figure 14: Training loss over the first 16 batches at 4 levels of overflow tolerance with max_len=100

4.4 Analysis

The primary conclusion we draw from this study is that data continuity – primarily a function of overflow control– appears to have a strong positive effect

on the model's ability to learn: holding the total data constant, it's best to complete fewer sequences than partially complete more. Furthermore, this notion of continuity appeared to be preserved with sequential truncation and imputation.

This study observes the importance of a well-designed overflow policy. This appears to hold independently of underflow control, as can be seen by comparing the relative training loss rates in 4.1 12 (no underflow policy) as well as in 4.2 13 (0-padding). Comparing each overflow policy before and after adding 0-adding does demonstrate the benefit of underflow policy through the lower average loss, however, the result is still less significant. Within the results in 4.2: while the largest sequence length produced an expected smoothing effect, the learning rates and average losses remain virtually unchanged despite considering two sequence length within a dense portion of the sequence length distribution. This suggests the model is capable of retaining continuity through across adjacent sequences.

We hypothesized our sophisticated policy would yield substantial improvement, but the results appear inconclusive. Perhaps more sophisticated sequence length handling procedures (such as considering specific characters and/or length of recent overflow sequences) will show stronger improvement. From our findings, we hypothesize that algorithms which emphasize whole-sequence completion with less concern for breaking them up will perform the strongest.

5 Codebase

Code: https://github.com/mmontelaro/HW2
Epoch Frenzy Checkpoints: https://drive.google.com/drive/folders/1weLUdP5O0gexnw6u64zFeMZpGDADO4
Hyperparameter Search Checkpoints: https://drive.google.com/drive/folders/193L7Sv77_aW-iFsYQTY6V9M8po0NC6ei?usp=drive_link
Best Model Checkpoint: https://drive.google.com/drive/folders/1FKMGAxda-YUsdkg8fb5l9zDaTNTlI-HN?usp=sharing

6 Contribution Breakdown

Patrick: Implemented the base model, including data processing, the training loop, and necessary codebase modifications. Suggested the experiments and ablation study which were adopted. Took over and completed ablation study when a teammate experienced an emergency.
Bolong: Began ablation study, including background research and data probing. Implemented a data ingestion pipeline that allows digesting the Pile dataset without downloading through streaming.
Anderson: Conducted hyperparameter search experiments and visualized the results. Scaled up to train a model on 27000 sequences for comparison. Provided some data unzip implementation. Modified `example.py` to get an intuitive generator interface in terminal.
Matthew: Conducted, visualized, and wrote about epoch frenzy experiment.

Put together Github repository as well as the rest of the codebase section. Wrote reflection section.

7 Reflection

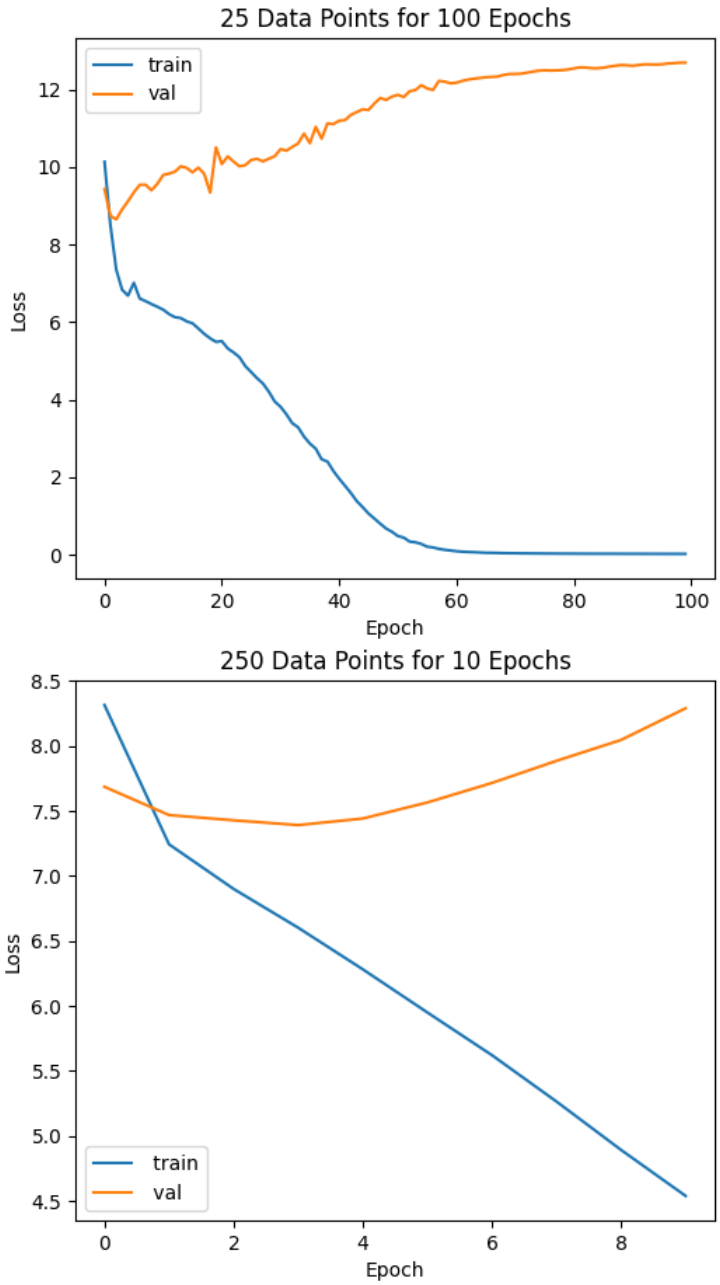
What we found least challenging was getting the report portion of the project put together once we all had our code and results. We were able to divide up the paper into parts and each write out our parts as well as help other members of the team as needed, and that process went smoothly. We also found coming up with ideas for experimentation to be a natural function of implementing the model, as this was each of ours' first time implementing an LLM.

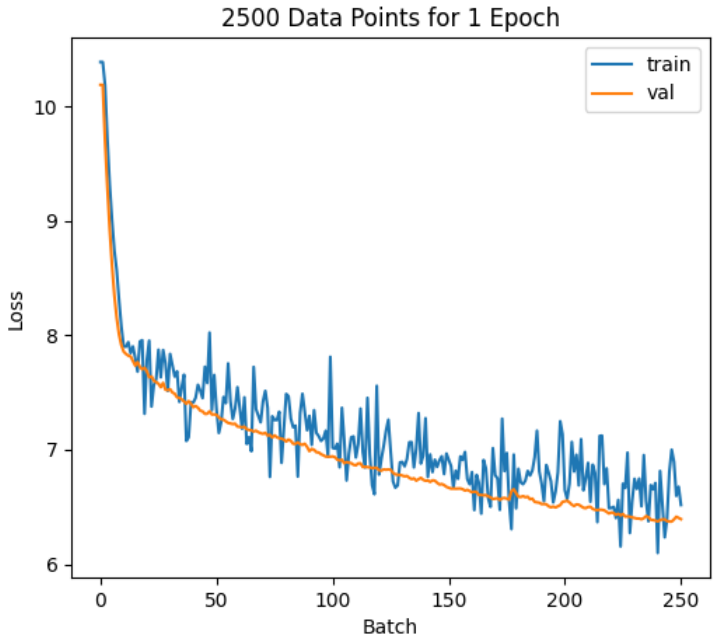
As product of the novelty, what we found most challenging was modifying the model to support training. Each member of the team attempted to get it to train using various methods until Patrick was able to put together a working model, although that still took a lot of troubleshooting. While the code is legible and cohesively written, there was little documentation to guide us. Another part that was more challenging than the rest of the project (but not as challenging as getting the model together initially) was modifying the code to support the experiments and ablation that we carried out. LLaMA is a deceptively intricate model, although we're all satisfied with the learning we gained from it.

Chatbot use disclosure

Class structures and plotting functionality were adapted from templates provided by ChatGPT.

A Plots for Epoch Frenzy





B Plots for Hyperparameter Selection

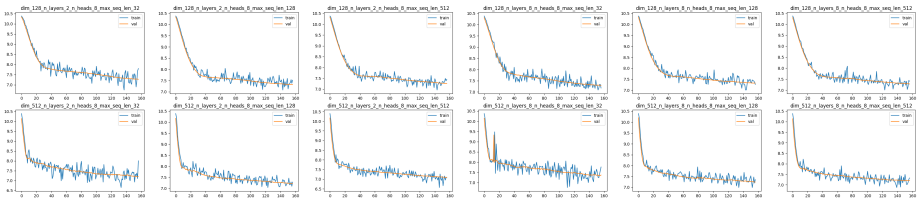


Figure 15: Hyperparameter Search Learning Curves Axis by dim, only n_heads = 8 is shown to reduce plot size

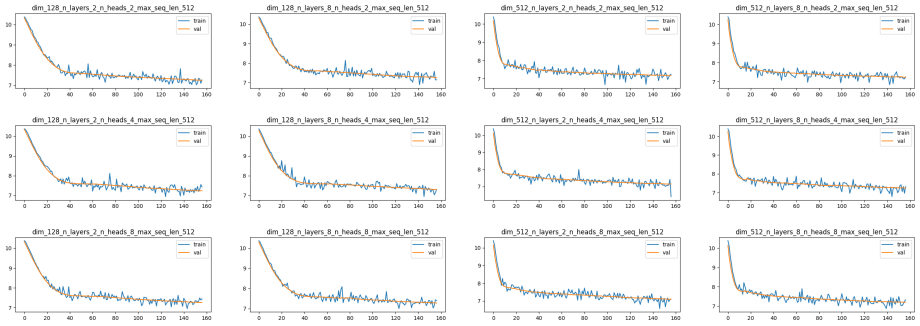


Figure 16: Hyperparameter Search Learning Curves Axis by n_heads, only max_seq_len = 512 is shown to reduce plot size

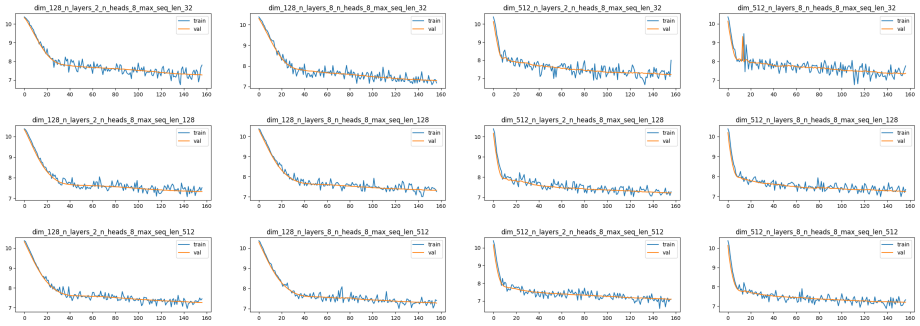


Figure 17: Hyperparameter Search Learning Curves Axis by max_seq_len, only n_heads = 8 is shown to reduce plot size

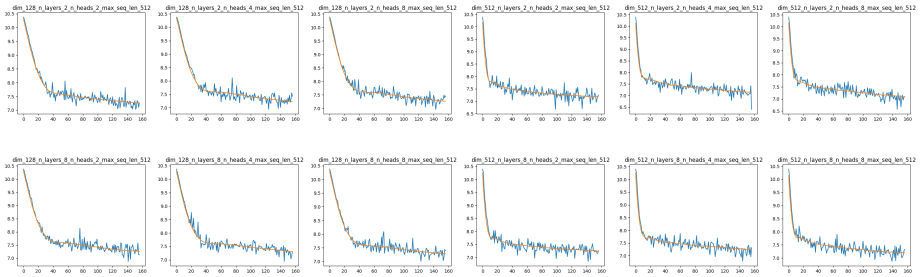


Figure 18: Hyperparameter Search Learning Curves Axis by n_layers, only max_seq_len = 512 is shown to reduce plot size

References

[1] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima,

Shawn Presser, and Connor Leahy. The pile: An 800gb dataset of diverse text for language modeling, 2020.