UW CSE 493S HW 2

Ming Yan, An Yan

Our code can be found at: `https://github.com/ming101266/CSE493_HW2/tree/main` along with some more of the graphs for our experiments that are not included.

Part 1: Infrastructure Setup

Logs of Experiments

EXP 1:

```
batch_size = 4
did not mask first three tokens
=== TRAINING TEXT ===
['I love machine learning.']
=== DataLoader Batches ===
X batch shape: torch.Size([3, 4])
Y batch shape: torch.Size([3, 4])
Attention Mask batch shape: torch.Size([3, 4])
  Batch 0:
    X (decoded, no pads): '<|endoftext|>I love machine'
    Y (decoded, no pads): 'I love machine learning'
    X (raw tokens): [50256, 40, 1842, 4572]
    Y (raw tokens): [40, 1842, 4572, 4673]
    Attention Mask: [True, True, True, True]
  Batch 1:
    X (decoded, no pads): 'I love machine learning'
    Y (decoded, no pads): ' love machine learning.'
    X (raw tokens): [40, 1842, 4572, 4673]
    Y (raw tokens): [1842, 4572, 4673, 13]
    Attention Mask: [True, True, True, True]
  Batch 2:
    X (decoded, no pads): ' love machine learning.'
    Y (decoded, no pads): ' machine learning.<|endoftext|>'
    X (raw tokens): [1842, 4572, 4673, 13]
    Y (raw tokens): [4572, 4673, 13, 50256]
    Attention Mask: [True, True, True, True]
number of parameters: 45.71M
num decayed parameter tensors: 6, with 45,714,432 parameters
num non-decayed parameter tensors: 3, with 2,304 parameters
using fused AdamW: False
Training loss: 0.0000
```

EXP 2:

Masked the first three tokens. During inference, the model would only output `learning`. This makes sense as the loss of the first three tokens was masked, and thus the model only learned to predict "learning."

Model Checkpoint

Inside the checkpoint folder, currently configured to checkpoint every 100 epochs.

Modifications Made

- Returned optimizer in `configure_optimizers`.
- Added option for tokenization.
  It seems more convenient to have it separate in training and inference. However, I stored the tokenizer name in `config` so it can be reconstructed during inference.
- Added `config.yaml` to avoid redefining configurations every time.

Challenges Encountered

1. **Model Stopping Criteria**
   Initially, I overlooked the need for special tokens to signal the beginning and end of sequences. Without these, the model didn't know when to stop generating. To address this, I used the GPT-2 end-of-text (EOT) token as both the beginning-of-sequence (BOS) and end-of-sequence (EOS) markers, padding each training input with it appropriately.
2. **Tokenizer Handling**
   At first, I considered storing the full tokenizer object in the configuration, but this isn't feasible since objects like the tokenizer can't be serialized to JSON or YAML. Instead, I saved just the tokenizer name (`"gpt2"` for part 1 and a custom tokenizer for part 2) and reconstructed the tokenizer at runtime using `tiktoken` for part 1.
3. **Padding and Masking**
   Since I'm using a decoder-only causal model (like GPT), padding doesn't inherently interfere with the forward pass, as long as attention masking is handled correctly. For simplicity, in this small example, the attention mask was set to all `True`. However, proper padding and masking will be important when batching longer sequences, especially when `block_size` increases and padding tokens are likely to appear. Without masking, these could cause unintended attention behavior and the model will fail to run on larger block sizes.
4. **Learning Rate Scheduling**
   I implemented a learning rate scheduler to manage training stability and improve convergence. In particular, I tried to use a cosine schedule with warmup of 0. Empirically this helped the model to improve even when the epochs are high.
5. **Training Performance**
   Training was initially taking a considerable amount of time on my local machine (roughly an hour per run), likely due to limited hardware capabilities. So, to keep ourselves entertained, we made a loading bar.
   To actually decrease training time and increase performance, we implemented gradient clipping and AMP (Automatic Mixed Precision). After applying these changes and running the model on more powerful machines in the UW labs, we reduced training time to about 10 minutes per run.

   Then, we decided to change our tokenizer form char-level to number & operator-level. This allowed us to reduce our block size from 15 to 5 and reduced our training time even more. Then, we discovered that instead of 1000 epochs, we we have been using so far, we would only need 100 for the model to converge, significantly speeding up the training process again.

   At the current moment, we only need a few seconds to run it on a UW server and just over 5 minutes on my local machine. We are very happy with how much we were able to improve our algorithm and this has been a very rewarding experience for all of us.

   Although this made our loading bar obsolete, we kept it to remind us of our struggles.

Part 2 - Algorithmic Task

Experiment Procedure

To reproduce the results, run the following commands in order:

```
python generate.py
python part2_train.py
python evaluate.py
```

Part 2: Algorithmic Tasks

Section 2.1: Data Generation

We generated 10,000 expressions of the form

$$a \text{ op } b = c \mod p$$

where $a, b \in [0, p-1]$, op $\in \{+, -\}$, and $c = (a \,\text{op}\, b) \bmod p$. These were shuffled and split as:

$$\text{Train: } 5000, \quad \text{Val: } 2500, \quad \text{Test: } 2500.$$

The configuration is specified in `part2config.yaml`:

```
generate:
  seed: 42
  op: '/'
  prime: 97
  num_samples: 10000
  data_dir: "ablation1/"
```

Samples are split into training, validation, and testing sets in proportions 50%, 25%, and 25%, and written to `train.txt`, `val.txt`, and `test.txt` in the experiment folder.

Section 2.2: Addition and Subtraction Experiments

Each experiment folder contains the full logs and results. We report summaries for the three random seeds: 42, 611, and 19683.

The model is recorded every 10 epochs, which is equivalent to 500 steps for our setup.

The training and validation loss is recorded at the end of every epoch and recorded in output.txt along with the final testing accuracy and loss.

The folders are named by seed_operator_prime_numSample_numLayer and all data relevant to the experiment including graphs, model checkpoints, data, and outputs.

Our models ran with the specs below with number of layer being 1 for half the experiments and 2 for the other half.

```
model:
  block_size: 5
  vocab_size: 16
  n_head: 4
  n_embd: 128
  dropout: 0.1
  bias: false
  tokenizer: "number-level"

train:
  batch_size: 100
  lr: 5e-4
```

```
  weight_decay: 0.05
  betas: [0.9, 0.95]
  epochs: 100
  save_interval: 10
  eval_interval: 1
  seed: 42
```

We have noticed that grokking occurs at almost all experiments except those with 113 as prime and with subtract operator.

Our model has difficulty learning these tasks with all 3 random seeds and achieved around accuracy around 35 percent each time.

The following graph is for seed 42, prime 97, 10k samples, and 1 layer we can clearly see grokking in this graph happening at around 25.

When grokking happened, there was a sudden and big change for validation accuracy to the point where it surpasses training accuracy.

This could be because we used dropout in our model and that there were less data for validation compared to training.

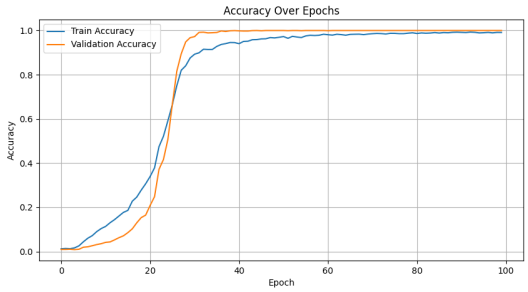


Figure 1: 1 Layer 42 seed 10k samples 97 as prime

We don't notice a big difference between having 1 or 2 layers in the graph but notice more of a difference depending on the task. Generally, 97 learned faster than 113 and addition learned faster than subtraction.

For the sake of keeping the project manageable, we chose to document only 20 experiments, each with at least 4 checkpoints placed at distinct epochs to ensure the experiment is captured.

Section 2.3: Grokking

With the following hyperparameters we were able to achieve perfect accuracy in modular division,

```
model:
  block_size: 5
  n_layer: 2
  n_head: 4
  n_embd: 128
  dropout: 0.1
  bias: false
  tokenizer: "number-level"

train:
  batch_size: 100
  save_dir: "grokking_checkpoints/"
  lr: 5e-4
  weight_decay: 0.05
  betas: [0.9, 0.95]
```

```
  epochs: 100
  save_interval: 10
  eval_interval: 1
  seed: 42

data:
  data_dir: "grokking_checkpoints/"

generate:
  seed: 42
  op: '/'
  prime: 97
  num_samples: 10000
  data_dir: "grokking_checkpoints/"
```

We first decided to run our model on the test set,
— Test Set Results —
Average Test Loss: 0.0154
Test Accuracy: 1.0000
Total Correct Predictions: 2500
Total Masked Tokens: 2500

Then, we got curious as to all our model would perform for all scenarios. Therefore, we tested it on mudulo division for all possible numbers, and we are delighted to report that it indeed evaluated all expressions correctly.
— Modulo Division (mod 97) Evaluation Results —
Total Expressions Evaluated: 9312
Total Correct Sequences (a/b=c): 9312
Accuracy: 1.0000

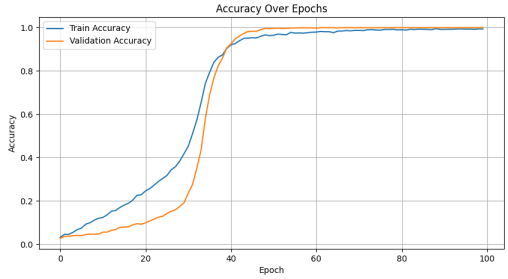


Figure 2: Accuracy for Grokking

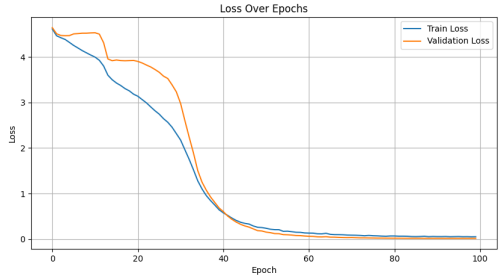


Figure 3: Loss for Grokking

From the graphs, we notice grokking at around 40 epochs. Since we have 50 steps per epoch, this is around 2000 steps, which is well within the 10^5 limit for this assignment.

Directions for inference with the model: Type

```
python3 part2_inference.py --prompt 2/1=
```

where 2/1 = should be replaced by the user's prompt.
Then, the model would output

```
number of parameters: 0.42M

=== Prompt ===
```

2/1=

=== Generated ===

2/1=2

for the user.

Section 2.4: Ablations

The Impact of Weight Decay on Grokking

Based on existing research and observations within the deep learning community, weight decay appears to be a critical hyperparameter that significantly influences if and how effectively a model exhibits grokking.

To quantify this influence, we'll conduct an experiment focusing on the grokking lag, the number of training steps between when the model achieves 100% training accuracy and when it subsequently reaches 100% validation accuracy. We will denote the grokking epoch as the first epoch when validation accuracy exceed training accuracy.

We will systematically evaluate this lag and grokking epoch by training our model with varying weight decay values: 0.01, 0.05, 0.1, and 0.5. All other hyperparameters will remain constant across these experimental runs to ensure that any observed differences can be directly attributed to the changes in weight decay.

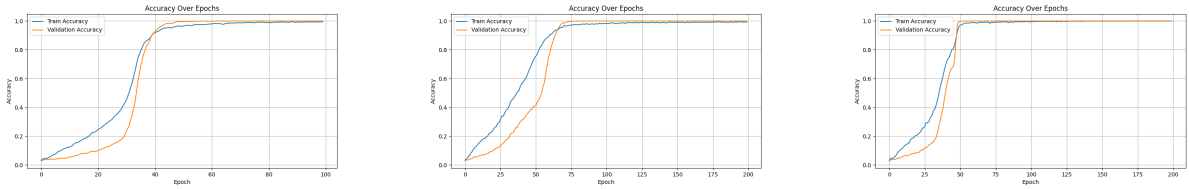
However, since increasing the weight decay also delays memorization, we have noticed that the model would take longer to train when weight decay is high. Thus, we decided to run the experiment for 200 epochs for weight decay is 0.5 and 0.1 instead of 100.

Weight Decay	**Grokking Lag (Steps)**	Grokking Epoch
0.01	-28	42
0.05	-38	40
0.1	-11	67
0.5	-13	49

Table 1: Grokking Lag and Epoch for Different Weight Decay Values

However, through the graph for 0.5 weight decay, we realized that the model did not grok at all. It generalized as soon as it memorized. Overall, the higher the weight decay, the more gradual the learning curve is. This confirms the claim that higher weight decay leads to models that generalizes better.

However, in this experiment, we noticed something very unusual: out model reaches 100% validation accuracy before training accuracy. We hypothesize that this might have to do with our utilization of gradient clipping.



(a) Weight Decay = 0.05 (b) Weight Decay = 0.1 (c) Weight Decay = 0.5

Figure 4: Training and Validation Performance for Different Weight Decay Values

To Add or Not to Add: That Is the Question

When a transformer model is trained on modular addition (e.g., 'a + b mod 97 → c'), it develops internal representations that associate pairs of input numbers with a specific output. These associations aren't based

on symbolic understanding of '+' or modular arithmetic rules, but instead arise from statistical pattern matching and correlation building.

So what happens when we present the same model with a subtraction prompt, like 'a - b =', even though it was never trained on subtraction?

The Core Mechanism

The model, having only seen 'a + b' during training, builds strong correlations between: - the pair of inputs '(a, b)' - and the result 'a + b mod 97'

It essentially learns: - "When I see input tokens 'a+b=', I expect output token 'c'" , where 'c = a + b mod 97' during training

Thus, when it it observes tokens 'a' and 'b', it learns to predict 'c'. It learns that these three numbers are closely related to each other.

Thus, when the model sees a subtraction prompt, e.g., '45 - 12 → ', since it has never encounted the operator '-', it guesses between the two groups which are associated with 12 and 45,

- '45 + 12 mod 97 = 57' - and - '12 + 33 mod 97 = 45'

Since both groups appear as valid, strongly associated outcomes for the same input pair '(a, b)', the model faces an ambiguous internal representation. Then, it essentially flips a coin between the two possible learned correlations.

This results in: Approximately 50% test accuracy on subtraction prompts even though the model was never trained on subtraction, which matches our observation.

Model is not guessing randomly

It's important to note: this isn't uniform random guessing over 97 outputs. That would yield 1% accuracy.

Instead, the model is guessing between two meaningful candidates and choosing one based on incomplete or misinterpreted operator signals.

This means the model has partially internalized the group structure of mod 97, but lacks a symbolic or semantic understanding of what the operator token '-' actually means,

Thus, in the absence of explicit subtraction training, the model reduces the decision to:

> To add or not to add? That is the question.

It knows the input pair is associated with something, but lacking clear operator context, it guesses between plausible meanings. That ambiguity produces the mysterious 50% accuracy on subtraction: a sign not of confusion, but of partial understanding constrained by training.

Further Suggestions

Evaluate Few-Shot Generalization from Addition to Subtraction

Investigate whether a model trained exclusively on modular addition can generalize to modular subtraction through 1-shot or few-shot learning. Specifically, test whether providing the model with a very small number of subtraction examples (e.g., 1–5 examples) enables it to correctly interpret and generalize the meaning of the '-' operator. If successful, this would suggest that the model's internal representations of modular arithmetic are structurally rich enough to support rapid task transfer across operations that share algebraic foundations.

Further Suggestions

Evaluate few-shot generalization from addition to subtraction: train on addition only, then introduce 1–5 examples of subtraction. Observe whether subtraction accuracy jumps, indicating latent group structure that supports rapid operator learning.

Appendix: Experiment Details and Results

Section 2.2 - Warmup

Layer	seed	Op	Prime	Train loss	Train acc	Test loss	Test acc
1	42	+	97	0.1548	0.9812	0.0149	0.9900
1	42	-	97	0.1855	0.9808	0.1613	0.9888
1	42	+	113	0.1618	0.9784	0.0151	0.9900
1	42	-	113	0.8557	0.7524	5.0707	0.2168
1	611	+	97	0.1480	0.9810	0.0123	0.9900
1	611	-	97	1.0640	0.8697	1.0755	0.8976
1	611	+	113	1.0688	0.8601	0.1034	0.9000
1	611	-	113	1.0347	0.8325	3.8304	0.2560
1	19683	+	97	1.0475	0.8386	0.1442	0.9000
1	19683	-	97	1.0673	0.8775	0.5996	0.7456
1	19683	+	113	1.0245	0.8375	0.1442	0.9000
1	19683	-	113	1.0274	0.8275	5.4187	0.2256
2	42	+	97	0.1405	0.9828	0.0049	1.0000
2	42	-	97	0.9660	0.9645	0.0613	0.9988
2	42	+	113	0.9662	0.9665	0.0051	1.0000
2	42	-	113	0.9667	0.9673	4.0707	0.3168
2	611	+	97	0.9414	0.9451	0.0023	1.0000
2	611	-	97	0.9640	0.9697	0.0755	0.9976
2	611	+	113	0.9688	0.9601	0.0034	1.0000
2	611	-	113	0.9347	0.9325	2.8304	0.3560
2	19683	+	97	0.1405	0.9828	0.0442	1.0000
2	19683	-	97	0.3021	0.9370	0.4996	0.8456
2	19683	+	113	0.1405	0.9814	0.0442	1.0000
2	19683	-	113	0.7798	0.8490	4.4187	0.3256

Table 2: Experiment Results