# HW2 Report

## Part 1: Infrastructure Setup

### Code Implementation

- `train.py`: Implements training loop with:
  - Number-level tokenization
  - Loss masking (only compute loss after '=' token)
  - Support for masking first N tokens (`-mask_first_n` flag)
  - Automatic saving of final model and loss/accuracy plots
- **inference.py**: Implements text generation with:
  - All the same number-level encoding and decoding.
  - Model loading from checkpoints
  - Temperature and top-k sampling
- run-experiment.py
  - Collectes all necessary experiments.
  - Option to run a single, a batch, and all experiments.
  - Automaticallly collects experiement log and trace status.

### Modifications from Original Codebase

1. **Custom Dataset Class**: `AlgorithmicDataset` with loss masking for equation format
2. **Number-level Tokenizer**: We find that character level tokenizer doesn't enable the model to grok for division tasks, therefore we implemented the number-level tokenization to strengthen the model capability to generalize. For sanity checks it falls back to character tokenization.
3. **Accuracy Metrics**: Added accuracy computation alongside loss
4. **Visualization**: Automatic generation of training curves

### Sanity Check Results

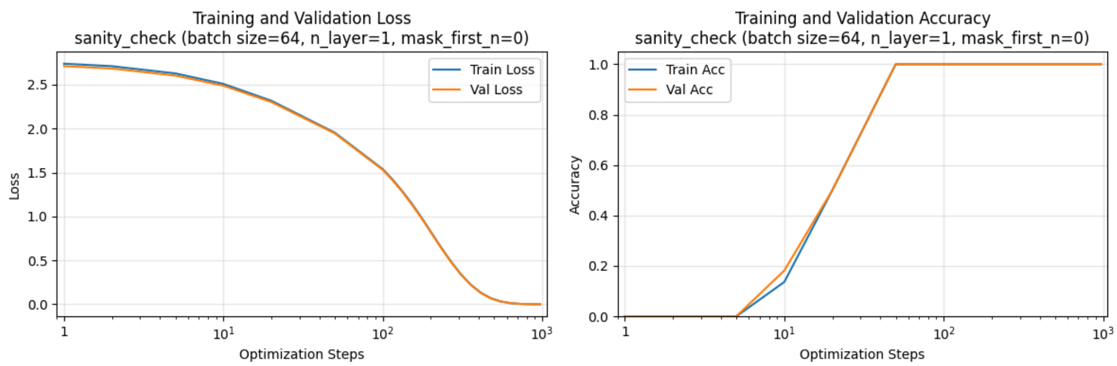
```
# Test 1: Basic memorization
python train.py --data_dir data/sanity_check --out_dir out/sanity_check --n_layer 1 --n_embd 32 --n_head 4 --max_steps 1000 --log_interval 100 --batch_size 64 --learning_rate 3e-4 --seed 42 --eval_points_per_decade 16

# Test 2: Masked first 3 tokens
python train.py --data_dir data/sanity_check --out_dir out/sanity_check_masked --n_layer 1 --n_em
```

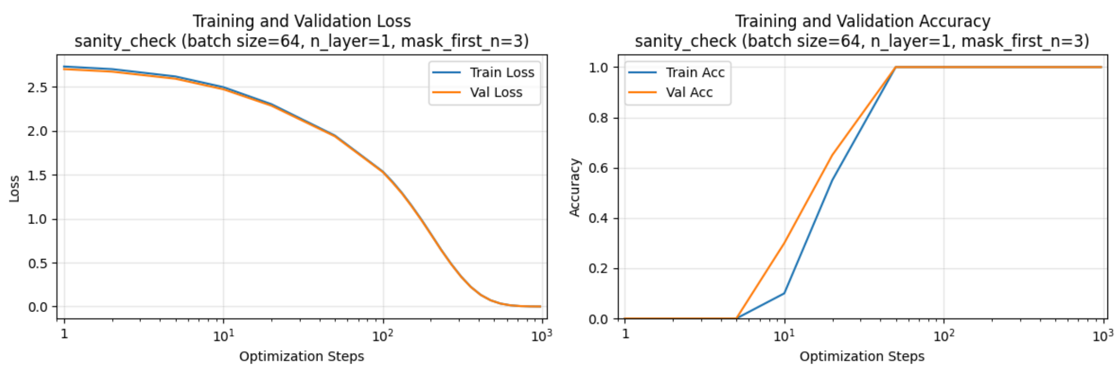
```
bd 32 --n_head 4 --max_steps 1000 --log_interval 100 --batch_size 64 --learning_rate 3e-4 --see
d 42 --eval_points_per_decade 16 --mask_first_n 3
```

Both tests successfully converged with train loss → 0. You can run the `inference.py` on the model checkpoint for two sanity checks.

Sanity Curves




Masked Sanity Curves




## Challenges Faced

- Debugging loss masking.

## Part 2.1: Data Generation

## Process Description

We generate complete datasets for modular arithmetic:

```
# For each operation and modulus p:
- Addition: a + b = c where c = (a + b) % p
- Subtraction: a - b = c where c = (a - b) % p
- Division: a / b = c where a = (b * c) % p
```

Division uses the reformulation where `a/b ≡ c (mod p)` means `a ≡ b×c (mod p)`.

## Dataset Statistics

| Task | Modulus | Total Examples | Train | Val | Test |
|---|---|---|---|---|---|
| Addition | 97 | 9,409 | 6,586 | 1,411 | 1,412 |
| Subtraction | 97 | 9,409 | 6,586 | 1,411 | 1,412 |
| Division | 97 | 9,312 | 6,518 | 1,396 | 1,398 |
| Addition | 113 | 12,769 | 8,938 | 1,915 | 1,916 |
| Subtraction | 113 | 12,769 | 8,938 | 1,915 | 1,916 |

## Part 2.2: Addition and Subtraction Experiments

## Configuration

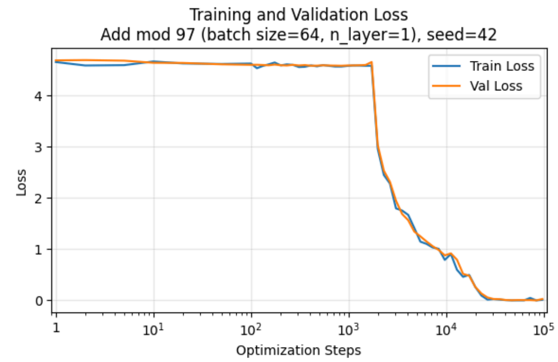
```
{
    'max_steps': 100000,
    'learning_rate': 1e-3,
    'log_interval': 1000,
    'n_embd': 128,
    'n_head': 4,
    'eval_points_per_decade': 16,
    'batch_size': 64
}
```

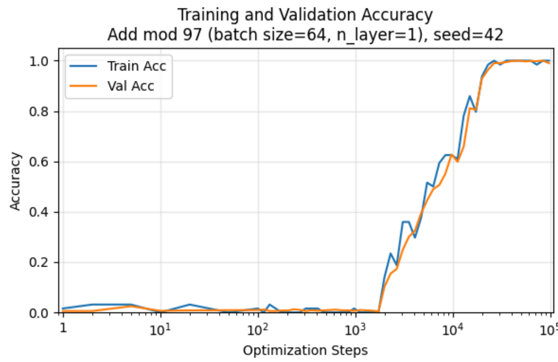
## Results (3 Random Seeds: 42, 123, 456)

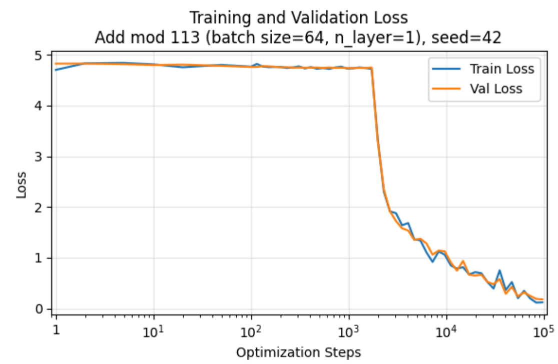
### 1-Layer Model Results

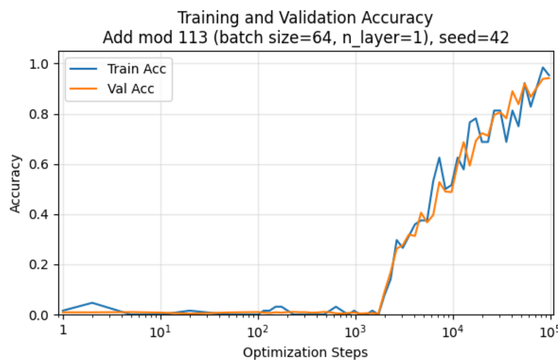
| Task | Metric | Seed 42 | Seed 123 | Seed 456 |
|---|---|---|---|---|
| Add p=97 | Final Train Loss | 0% | 0% | 0% |
| Add p=97 | Final Train Acc | 100% | 100% | 100% |
| Add p=97 | Final Test Acc | 99.04% | 99.36% | 99.07% |
| Add p=113 | Final Train Loss | 0% | 0% | 0% |
| Add p=113 | Final Train Acc | 100% | 100% | 100% |
| Add p=113 | Final Test Acc | 97.27% | 97.26% | 97.78% |
| Sub p=97 | Final Train Loss | 0% | 0% | 0% |
| Sub p=97 | Final Train Acc | 100% | 100% | 100% |
| Sub p=97 | Final Test Acc | 97.08% | 91.92% | 97.15% |
| Sub p=113 | Final Train Loss | 2% | 0% | 0% |
| Sub p=113 | Final Train Acc | 100% | 100% | 100% |
| Sub p=113 | Final Test Acc | 91.49% | 94.06% | 93.54% |

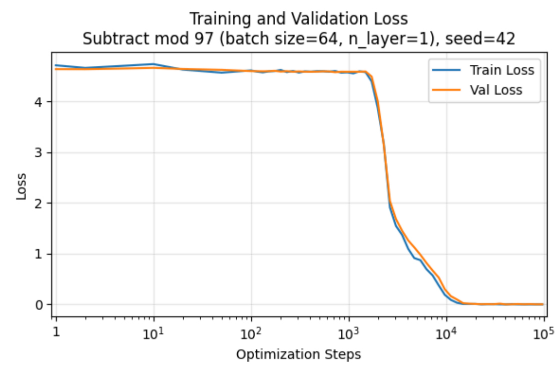
Plots for the experiments. We have also collected plots for the other 2 random seeds but only reporting the `42` as directed. Other plots can be found under the `out` directory. Params are given in the title:

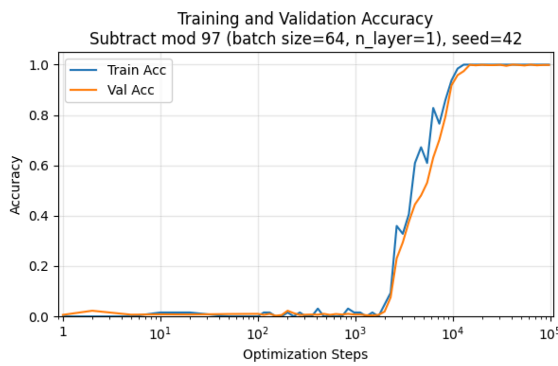


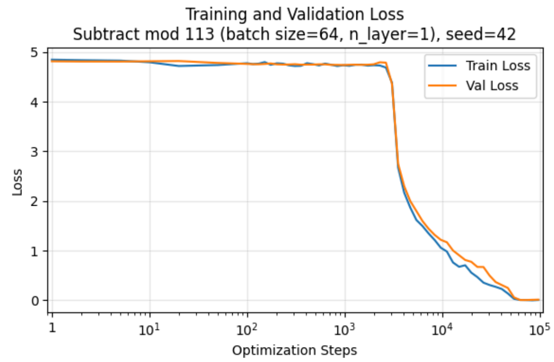



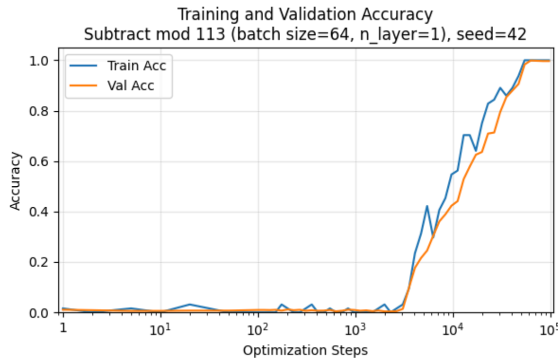



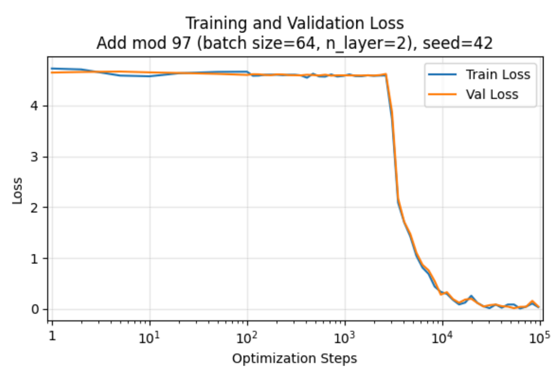


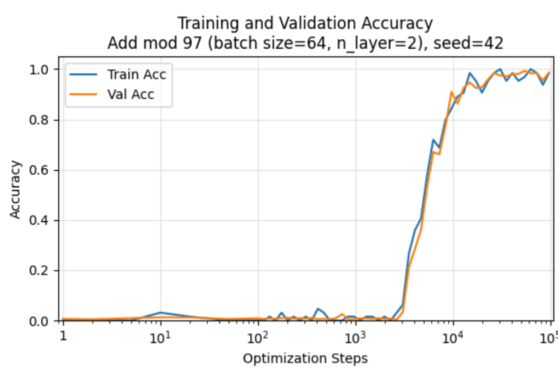



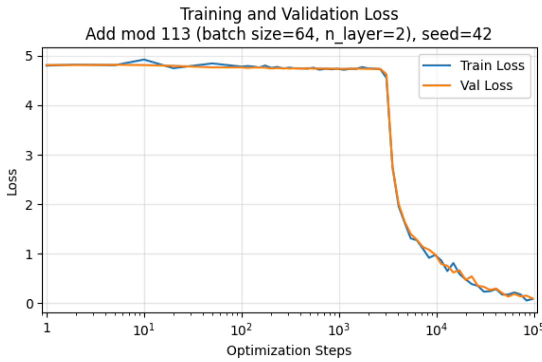

## 2-Layer Model Results

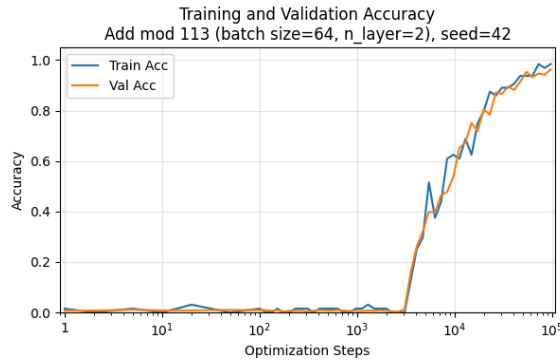
| Task | Metric | Seed 42 | Seed 123 | Seed 456 |
|---|---|---|---|---|
| Add p=97 | Final Train Loss | 0% | 0% | 0% |
| Add p=97 | Final Train Acc | 100.0% | 100.0% | 100.0% |
| Add p=97 | Final Test Acc | 99.78% | 99.96% | 99.78% |
| Add p=113 | Final Train Loss | 0% | 0% | 0% |
| Add p=113 | Final Train Acc | 100% | 100% | 100% |
| Add p=113 | Final Test Acc | 98.49% | 98.04% | 98.34% |
| Sub p=97 | Final Train Loss | 0% | 0% | 0% |
| Sub p=97 | Final Train Acc | 100% | 100% | 100% |
| Sub p=97 | Final Test Acc | 99.2% | 99.31% | 99.64% |
| Sub p=113 | Final Train Loss | 0% | 0% | 1.8% |
| Sub p=113 | Final Train Acc | 100% | 100% | 100% |
| Sub p=113 | Final Test Acc | 98.46% | 98.44% | 97.87% |

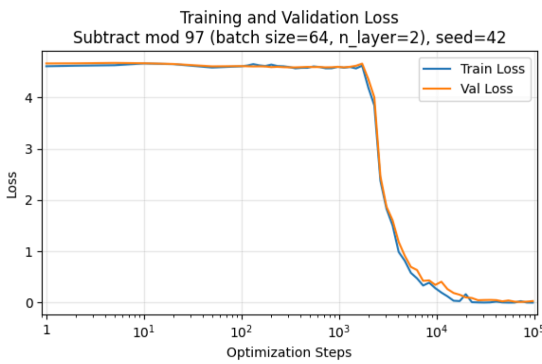
Plots:

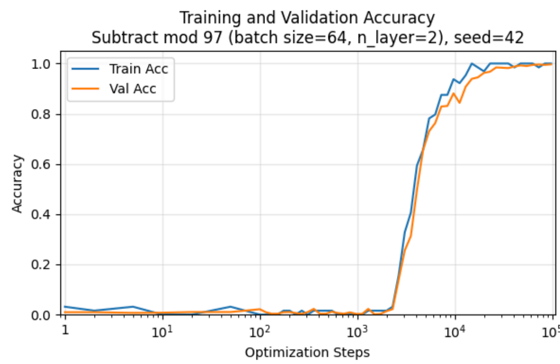


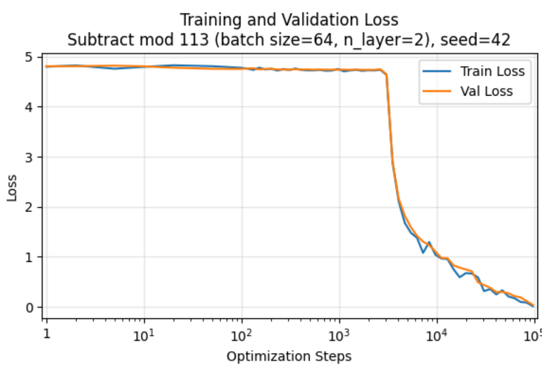



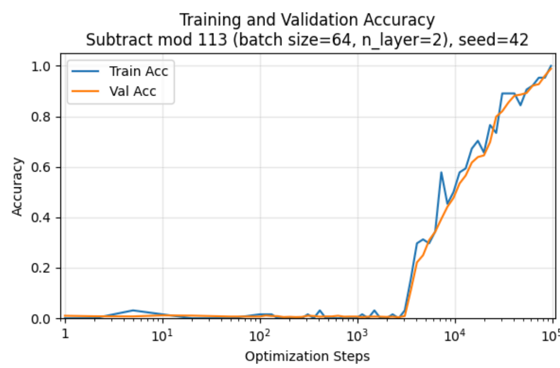


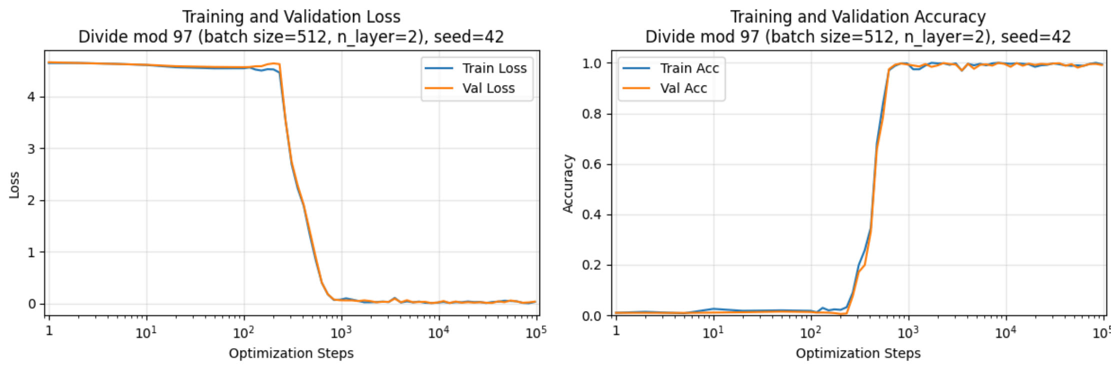




## Model Checkpoints

Final models saved at: `out/[task]_mod[p]_layer[n]_seed[n]/final_model.pt`

## Part 2.3: Grokking

We trained on the modulo division task for p = 97, and the training curves are as follows:




## Detailed training configurations:

```
{
    'n_layer': 2,
    'n_embd': 128,
    'n_head': 4,
    'batch_size': 512,
    'learning_rate': 1e-3,
    'weight_decay': 1.0,
    'beta_1': 0.9,
    'beta_2': 0.98
    'max_steps': 100000
}
```

### Grokking Results

- Final **validation loss:** 0.0972, **validation accuracy:** 0.9913.
- Final **training loss:** 0.0160, **training accuracy:** 0.9986

### Model Checkpoint

`out/divide_mod97_layer2_batch512/final_model.pt`

### Inference Instructions

```
python inference.py --checkpoint out/divide_mod97_layer2_seed42_batch512/final_model.pt --pro
mpts "91/7=" --max_new_tokens 1
```

## Part 2.4: Ablation Study

### Motivation

We investigated how batch size affects grokking speed and reliability, as batch size influences gradient noise and learning dynamics.

## Experimental Setup

- **Fixed**:

```
{
    'n_layer': 2,
    'n_embd': 128,
    'n_head': 4,
    'learning_rate': 1e-3,
    'weight_decay': 1.0,
    'beta_1': 0.9,
    'beta_2': 0.98
    'max_steps': 100000
    'p': 97
}
```

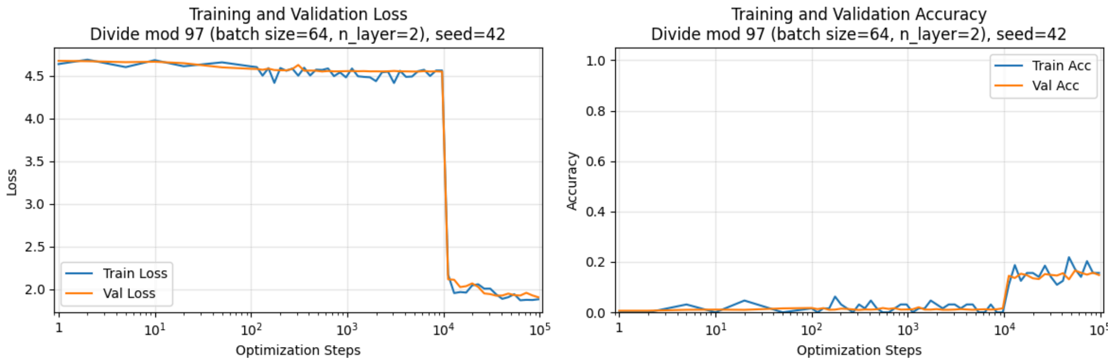
- **Varied**: batch_size ∈ {64, 128, 256, 512}
- We tested across three different seeds to ensure reliability. For simplicity we report only the seed=42 plots here, but plots for other seeds are available at `out/divide_mod97_layer2_<suffix>` . Result path to batch size mapping is as follows:

| Batch Size | Path |
| --- | --- |
| 64 | `out/divide_mod97_layer2_seed{seed}_batch{batch}` |
| 128 | `out/divide_mod97_layer2_seed{seed}_batch{batch}` |
| 256 | `out/divide_mod97_layer2_seed{seed}_batch{batch}` |
| 512 | `out/divide_mod97_layer2_seed{seed}_batch{batch}` |

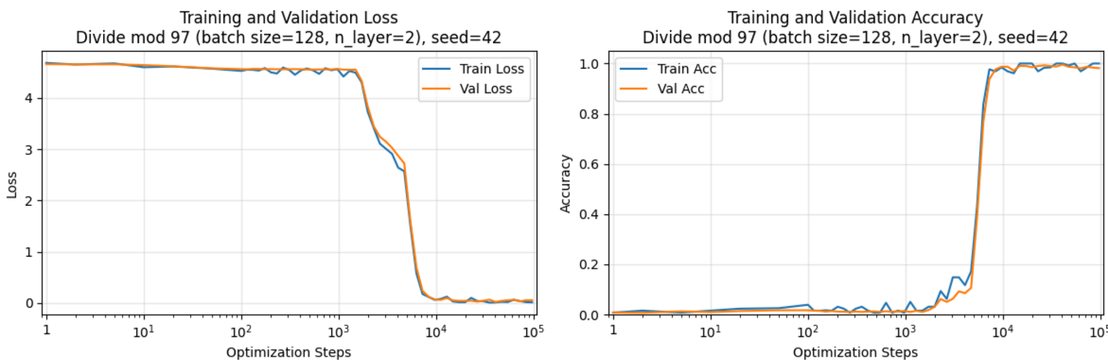
## Results

| Batch Size | Steps to Grok* | Final Training Loss | Final Training Accuracy | Final Test Accuracy |
| --- | --- | --- | --- | --- |
| 64 | Not Grokking** | 1.7755 | 18.25% | 17.06% |
| 128 | ~9000 | 0.0191 | 98.97% | 97.85% |
| 256 | ~10000 | 0.0032 | 99.83% | 99.24% |
| 512 | ~800 | 0.0048 | 99.62% | 99.57% |

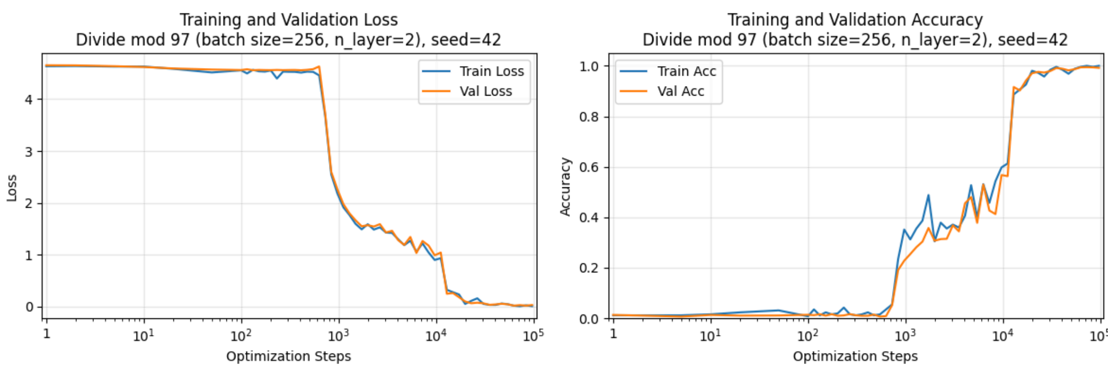
Batch size=64



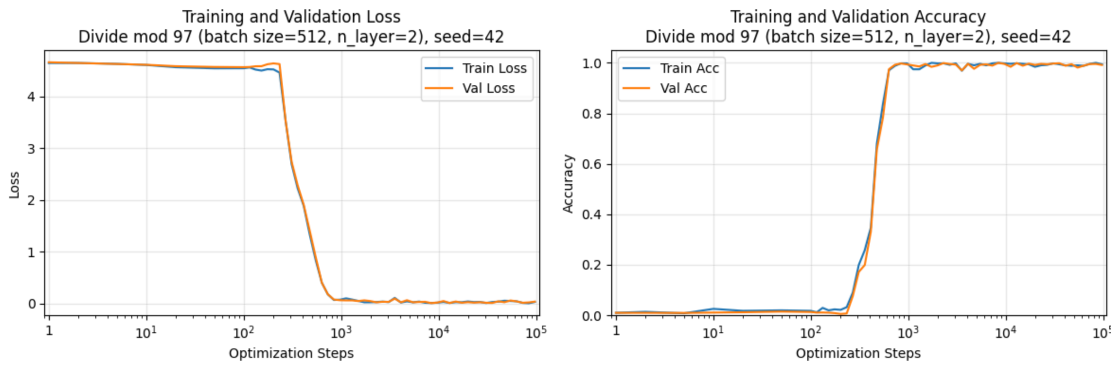

Batch size=128




Batch size=256




Batch size=512




*We mark the steps-to-grok as the number of steps the model takes to reach a generally high and stable test accuracy.

**We find that on batch size 64 there's no sign of grokking at all, i.e. the test accuracy keeps bouncing in a very low range and do not present signs of increasing.

As the conclusion, we find that batch size does affect the capability and timing of the model to grok. Larger batch size usually result in earlier grokking during training, and fewer batch size may actually cause the model unable to grok in a reasonable time.