# **Homework #2: Transformer Training on Algorithmic Tasks**

**CSE 493S/599S: Advanced Machine Learning**

Team Members: [Bohan Fang, Shanli Xing]

## **Overview**

This project implements transformer models for learning modular arithmetic operations (addition, subtraction, and division) and investigates the grokking phenomenon. We train small transformer models from scratch and analyze their learning dynamics on algorithmic tasks.

## **Project Structure**

.

├── model.py # Transformer model (adapted from nanoGPT)

├── train.py # Main training script

├── inference.py # Text generation and model evaluation

├── generate\_data.py # Data generation for modular arithmetic

├── run\_experiments.py # Script for running multiple seeds

├── data/ # Generated datasets

│ ├── sanity\_check/ # Sanity check data

│ └── algorithmic/ # Modular arithmetic datasets

│ ├── add\_mod97/

│ ├── subtract\_mod97/

│ ├── divide\_mod97/

│ └── ...

└── out/ # Model checkpoints and results

## **Setup and Requirements**

### **Dependencies**

pip install torch numpy matplotlib

* Python 3.8+
* PyTorch 2.0+
* CUDA-capable GPU (recommended but not required)

### **Running the Code**

1. **Generate Data:**

# Generate all modular arithmetic datasets

python generate\_data.py --operations add,subtract,divide --moduli 97,113 --output\_dir data/algorithmic

# Generate sanity check data

python generate\_data.py --sanity\_check --output\_dir data/sanity\_check

1. **Run Training:**

# Example: Train on addition mod 97

python train.py \

--data\_dir data/algorithmic/add\_mod97 \

--out\_dir out/add\_mod97 \

--n\_layer 1 \

--n\_embd 128 \

--n\_head 4 \

--batch\_size 64 \

--max\_steps 100000

1. **Run Inference:**

python inference.py --checkpoint out/add\_mod97/final\_model.pt --prompts "23+45=" --temperature 0.1

## **Part 1: Infrastructure Setup**

### **Modifications to Original Codebase**

1. **Custom Dataset Class**: Implemented AlgorithmicDataset with loss masking to only compute loss on answer tokens (after '=')
2. **Character-level Tokenizer**: Created simple tokenizer suitable for mathematical expressions
3. **Loss Masking**: Added support for masking first N tokens for sanity checks
4. **Metrics Tracking**: Added accuracy computation alongside loss
5. **Automated Plotting**: Generate training curves automatically

### **Sanity Checks**

We implemented two sanity checks:

1. **Basic memorization**: Train on "I love machine learning"

python train.py --data\_dir data/sanity\_check --out\_dir out/sanity\_check --n\_layer 1 --n\_embd 32 --max\_steps 1000

1. **Masked memorization**: Mask loss on first 3 tokens

python train.py --data\_dir data/sanity\_check --out\_dir out/sanity\_check\_masked --mask\_first\_n 3 --n\_layer 1 --n\_embd 32 --max\_steps 1000

Both tests passed - the model successfully memorized the string with train loss approaching 0.

## **Part 2: Data Generation**

### **Approach**

We generate all possible equations for each operation modulo p:

1. **Addition**: a + b = c (mod p) for all 0 ≤ a, b < p
2. **Subtraction**: a - b = c (mod p) for all 0 ≤ a, b < p
3. **Division**: a / b = c (mod p) for all 0 ≤ a < p, 1 ≤ b < p where gcd(b, p) = 1

**Implementation Details**

# Division requires computing modular inverse

def mod\_inverse(a, m):

# Extended Euclidean algorithm

m0, x0, x1 = m, 0, 1

while a > 1:

q = a // m

m, a = a % m, m

x0, x1 = x1 - q \* x0, x0

return x1 + m0 if x1 < 0 else x1

**Dataset Statistics**

For p = 97:

* Addition/Subtraction: 9,409 equations (97²)
* Division: 9,312 equations (excludes b=0)
* Train/Val/Test split: 70%/15%/15%

## **Part 2.2: Addition and Subtraction Results**

### **Configuration Used**

{

'n\_layer': 1 or 2,

'n\_embd': 128,

'n\_head': 4,

'batch\_size': 64,

'learning\_rate': 1e-3,

'max\_steps': 100000

}

### **Results**

### Both addition and subtraction were learned successfully with high test accuracy, indicating good generalization without grokking behavior.

## **Part 2.3: Division and Grokking**

### **Attempts and Challenges**

Our attempts at training on division failed to show grokking within 100,000 steps. The model achieved can’t learn the modular division.

### **Analysis of Failure**

We identified two main reasons for not observing grokking:

1. **Insufficient Training Steps**:  
   * Grokking on division often requires 200,000-500,000 steps
   * Our initial experiments only ran for 100,000 steps
   * The model needs extensive time to discover the modular inverse pattern
2. **Batch Size Too Small**:  
   * We used batch\_size=64 and 16, but literature suggests larger batches (512+) for division
   * Small batches create too much gradient noise for learning complex patterns
   * Division requires learning: modular inverse → multiplication → modular reduction

## **Part 2.4: Ablation Study**

### **Factor Studied: Batch Size Impact on Grokking**

We investigated how batch size affects the grokking phenomenon on the division task.

After adjusting parameters based on the literature:

{

'n\_layer': 2,

'n\_embd': 128,

'n\_head': 4,

'batch\_size': 256 or 512,

'learning\_rate': 1e-3,

'weight\_decay': 1.0,

'max\_steps': 100000

}

### With these settings, we observed that training accuracy starts to increase, but due to computation resources limits, we fail to go through the full training process of this config.

### **Key Findings**

1. **Larger batch sizes increase the chance of learning and grokking**: 256 and 512 batch sizes, the model begins to learn.
2. **Too small batches prevent grokking**: Batch sizes ≤64 didn't learn anything within 100k

### **Hypothesis**

Larger batch sizes provide:

* More stable gradients for learning complex modular inverse patterns
* Better signal-to-noise ratio for discovering algebraic structure
* Sufficient examples per update to recognize patterns across the full modular space

## **Challenges Faced**

1. **Debugging Division Training**: Initial runs showed no learning (stuck at ~10% accuracy)..
2. **Computational Constraints**: Full grokking experiments require 300k+ steps, taking several hours on GPU.
3. **Hyperparameter Sensitivity**: Division task is extremely sensitive to learning rate, weight decay, and batch size combinations.