Sai Eeshwar D **25338829**

1.Part 1 - Math GPT

1.1. Dataset design and rationale

At first the dataset contains only single‑digit addition and subtraction. Each line is a simple expression of the format:

“**A**<*operation*>**B**=**C**”

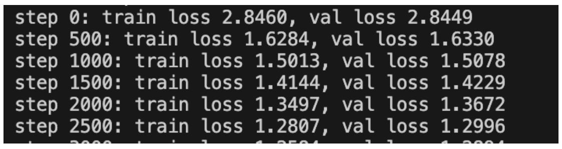
where A and B are single-digit integers and the operation is either ‘+’ or ‘−’. All expressions follow a strict and uniform format, with no spaces and exactly **one expression per line**, which keeps tokenisation simple and consistent. This dataset was used as a starting point to verify that a model could reliably learn basic arithmetic. Training on this simplified data allowed me to experiment with and tune hyperparameters such as model size, learning rate, and number of training steps. Once the model was able to achieve very high expression-level accuracy on addition and subtraction, it provided confidence that the model architecture and training setup were working correctly. **After the model learned addition and subtraction, the dataset was extended to include multiplication and division while keeping the same format**. The final dataset contains all four operations (+, −, *, /) with operands in the range 0 - 99. Subtraction allows negative results, and division uses integer truncation, often producing zero when the divisor exceeds the dividend. To prevent bias, exactly 5,000 examples were generated per operation, and all expressions were shuffled before the training–validation split to avoid ordering effects.

Train/validation split
All non-empty lines were loaded and shuffled with a fixed random seed to ensure reproducibility. The **shuffled lines** were then split into **80% training (train_lines) and 20% validation (val_lines). This prevents biases from file ordering, such as a validation set dominated by division examples.**
After splitting, train_text and val_text were created by joining the lines with newlines. A character-level vocabulary was built from the combined text, including digits, operators, =, -, and newline.
Finally, train_data and val_data were generated as integer tensors using a character-to-index mapping (stoi). Shuffling before splitting ensures both training and validation sets have a balanced mix of operands and operations.

1.2. Evaluation metrics

Two main metrics are used: **cross‑entropy** and expression‑level **accuracy**, with additional per‑operation accuracies for the full arithmetic dataset.
Cross‑entropy loss
The model is trained with the usual next‑character cross‑entropy loss:
- Given input indices 'idx' and targets 'targets', the model outputs logits of shape.
- These are reshaped to and compared to flattened targets via 'F.cross_entropy'.

On the datasets at all times, training and validation cross‑entropy are logged for every ‘eval_interval’ step.


```
step 0: train loss 2.8460, val loss 2.8449
step 500: train loss 1.6284, val loss 1.6330
step 1000: train loss 1.5013, val loss 1.5078
step 1500: train loss 1.4144, val loss 1.4229
step 2000: train loss 1.3497, val loss 1.3672
step 2500: train loss 1.2807, val loss 1.2996
```
Fig-01: Training Iterations of a model

This metric is useful to monitor optimization progress and detect overfitting: for example, on math data the validation loss decreases from ≈2.8 at step 0 to ≈1.2 around step 2000–2500, then starts to increase as the model overfits. **However, cross‑entropy is not ideal as the primary task metric, because a low loss does not guarantee that the final numeric result 'c' is correct.**

Expression‑level accuracy

The main task metric is the fraction of expressions for which the model outputs the correct integer result.The evaluation procedure is:

1. Split the line into ‘expr’ and ‘truth’ around ‘=’.
2. Construct a prefix 'lhs = expr.strip() + "=" (e.g. "12+7=").
3. Encode lhs and ask the model to generate a few additional characters (typically ≤5) using the same 'generate' function used for free sampling.
4. Decode the full string and extract the substring after '='.
5. Parse the leading integer (allowing an optional leading '-'). If parsing fails, the prediction is treated as empty.

If the parsed integer matches the ground‑truth 'truth', that expression is counted as correct. The expression‑level accuracy is then:

Accuracy = (no. of correct expressions) / (no. of evaluated expressions)

This metric directly captures whether the model is actually doing arithmetic, and is the main metric reported in the results. Each subset of validation lines is passed to evaluate math function, yielding accuracies for addition, subtraction, multiplication and division separately. This allows a detailed analysis of which operations are learned correctly and which remain difficult.

1.3. Model architecture and adaptations

For understanding the impact of the parameters I trained the model with different configurations on the same dataset.

Config	Params (M)	batch_size	max_iters	block_size	n_embd	n_head	n_layer	Dropout
1	6.33704	64	3500	64	256	8	8	0.2
2	1.193744	64	5000	64	128	6	6	0.1
3	4.759056	64	2500	64	256	4	6	0.1

Table-01:Architecture and training hyperparameters for Math GPT configurations, summarizing depth, width, context length, and training budget for reproducibility. Models are listed from best-performing (top) to lowest-performing (bottom).

The starting point is the minimal GPT implementation used in class. The math GPT keeps this overall structure:

- Token embedding: mapping each character to a dense vector of size 'n_embd'.
- Positional embedding: added to the token embeddings to encode positions up to 'block_size' characters.
- Transformer blocks: a stack of ‘n_layer’ identical blocks, each with:
- Multi‑head self‑attention (with 'n_head' heads and per‑head size 'n_embd // n_head').
- A feed‑forward network 'Linear(n_embd, 4*n_embd) to ReLU to Linear(4*n_embd, n_embd)’.
- Layernorm and residual connections around both attention and feed‑forward sublayers.
- Output layer: final 'LayerNorm' and linear projection to logits over the vocabulary.

The chosen parameters (8 heads, 8 layers, and 256‑dimensional embeddings) balance model capacity and computational efficiency. Increasing the number of layers and heads allows the model to capture more complex dependencies in arithmetic expressions, while a moderate embedding size ensures sufficient representational power without excessive overfitting or training cost.

1.4. Operation‑wise evaluation and failure analysis

On the simple dataset, with only addition and subtraction the best 4.7M‑parameter model achieves ≈0.849 validation expression‑level accuracy. Generated samples include:

- '1+5=6'
- '8+7=15'
- '2+0=2'
- '6-6=0'
- '4-6=-2'

which are all correct on the underlying distribution. Errors mainly occur on less frequent patterns or when the model generates extra digits beyond the correct answer.

Because at this stage the dataset contained only addition and subtraction with single-digit operands, it shows that the architecture can learn basic arithmetic compositions reliably. It also provides a calibration for later: if the same model struggles on multiplication or division, the limitation is in the task complexity rather than in the basic architecture.

On the full dataset, the 6.3M-parameter configuration achieves ≈0.75 validation expression-level accuracy at its best checkpoint. Sample generations include:

- Additions: '60+19=89', '1+5=6', which are correct.
- Subtractions: '1-82=-81', '0-5=-5', which are also correct.
- Multiplications: '7*9=63', '1*3=3', '20*17=240', which are correct in many cases.
- Divisions: '6/7=0', '71/65=1', '41/90=0', which are correct truncated quotients; occasionally the model outputs dubious examples such as '30/0=3' where the divisor is zero and the result is mathematically undefined.

Per-operation accuracies (computed with 'is_add', 'is_sub', 'is_mul', 'is_div') show a consistent pattern:

- Addition has the highest accuracy; the model is close to its performance on the simple dataset, reflecting that addition dominates the clean structure the model has already learned.
- Subtraction is slightly worse, especially when the result is negative; some generated examples drop the minus sign or mis-compute near boundaries.

- Multiplication is significantly harder: the number of possible products grows quickly with operand range, and the model sometimes confuses products with addition-like patterns (e.g. producing '7*8=54' instead of 56 in some runs).
- Division is the hardest: many examples involve results 0 or 1, and the model must implicitly learn integer truncation; it sometimes mis-handles edge cases or generates spurious digits.

Config	Best val CE	Val add (+) acc	Val sub (−) acc	Val mul (*) acc	Val div (/) acc	Val math acc
1	≈1.29	0.855	0.710	0.542	0.904	**0.756**
2	≈1.29	0.858	0.784	0.508	0.883	**0.767**
3	≈1.30	0.854	0.728	0.555	0.876	**0.756**

Table-02:Math GPT validation accuracies on arithmetic subsets (addition, subtraction, multiplication, and division), where the largest model (Config 1) attains the best overall math accuracy. Models are listed from best-performing (top) to lowest-performing (bottom).

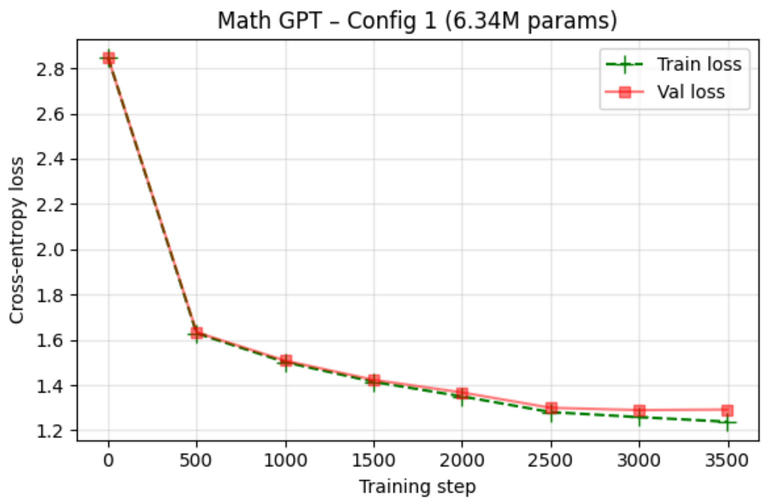


Fig-02: Math GPT Training Loss Vs Validation Loss

These findings support a clear conclusion:

- A GPT-style transformer trained at the character level can learn simple single-digit addition and subtraction very well.
- The same architecture, scaled up to ≈6M parameters and trained on a balanced dataset with +, −, * and / over 0–99, still struggles to fully master multiplication and division under the given training budget.
- The gap between addition/subtraction and multiplication/division is visible both in quantitative per-operation accuracies and in qualitative failure cases, satisfying the requirement for an "insightful failure analysis". The accuracy severely drops in the case of multiplication and that is because of the complex patterns it forms and the best way to tackle it in my opinion is to provide the model more simple multiplication patterns to start with and then increase the complexity.

Overall, the experiments demonstrate a systematic progression from simple to complex datasets, careful definition of metrics, and a clear analysis of how architectural choices and task difficulty affect performance on symbolic arithmetic.

2.Part 2 - Boolean GPT

2.1.Dataset Creation

For this I made a boolean logic dataset by making a python script to cover key logical operators and structural patterns.

Expression types
The dataset includes simple binary expressions of the form **A** *OP* **B** = **C**, where A, B, and C ∈ {True, False} and OP ∈ {AND, OR, XOR, NOT, NOR, NAND}. It also includes unary expressions using NOT, written as NOT **A** = *result*. In addition, the dataset contains nested expressions with parentheses, such as (**A** *OP1* **B**) *OP2* **C** = *result*, where OP1 and OP2 ∈ {AND, OR, XOR}.

Generation and labels
Boolean values are sampled with rand_bool() and converted to strings by bool_to_str (True/False).For binary operations, the ground-truth value is computed using apply_binary (custom logic for AND, OR, XOR); for unary NOT, Python's not is applied. The expression and its truth value are joined into a single line, e.g. "True AND False = False\n", ensuring labels are consistent with the expressions.

Dataset size and split
The generator writes N = 20000 expressions into the boolean dataset, mixing simple, NOT, and nested expressions. In the training script, all lines are read and split into train and validation. For the Boolean dataset, a 90 - 10 split was used to maximise the amount of training data on a relatively smaller corpus while still leaving enough examples to compute reliable per-class validation accuracies.
For language-model training, train_lines and val_lines are concatenated into train_text and val_text, which are then encoded at the character level into train_data and val_data tensors.
This design provides a balanced, fully labelled dataset over the Boolean vocabulary, including both flat and nested expressions, which is appropriate for learning Boolean logic behaviour.

2.2. Evaluation metrics

The Boolean GPT model was evaluated using token-level cross-entropy loss and expression-level Boolean accuracy, with additional per-class accuracy by operator and expression structure. Cross-entropy measures the model's ability to predict the next character in an expression, computed over both training and validation sets at regular intervals to form loss curves that track convergence and learning stability.

Expression-level Boolean accuracy evaluates whether the model correctly predicts the result of complete Boolean expressions. Given a prefix like "True AND False = ", the model generates the remaining characters, and the predicted result is compared to the ground-truth value. Accuracy is calculated as the proportion of expressions correctly evaluated, providing a high-level measure of logical correctness.

To further analyze performance, validation expressions are grouped by operator and structure, including AND/OR-only, expressions with NOT, XOR, NAND/NOR, and nested expressions with parentheses. Each subset is evaluated separately, reporting accuracy for each category. Together, these metrics give a clear picture of both the model's low-level token prediction ability and its high-level reasoning over Boolean logic.

Config	Overall Boolean acc	AND/ OR val acc	NOT val acc	Parentheses val acc	XOR val acc	NAND/NOR- only val acc	With NAND val acc	With NOR val acc
1	0.999	0.998	1.000	0.997	0.995	1.000	1.000	1.000
2	0.923	0.971	0.987	0.777	0.825	0.977	0.883	0.842
3	0.629	0.498	0.921	0.493	0.499	0.502	0.534	0.467

Table-03:Boolean GPT validation accuracies for all configurations across operator subsets (AND/OR only, NOT, XOR, and parentheses), showing that the largest model (Config 1) consistently achieves the highest accuracy. Models are listed from best-performing (top) to lowest-performing (bottom).

2.3. Architecture

All configurations were trained on the same Boolean dataset with the same training schedule (max_iters, eval_interval, learning rate) and next-character cross-entropy loss, so differences in performance can be attributed to architectural choices rather than optimisation settings. The smallest model already achieved high accuracy on simple AND/OR and NOT expressions, but showed noticeably lower performance on more complex subsets involving nested parentheses, XOR, and combinations with NAND/NOR, indicating that very small embeddings and shallow depth limit the capacity to capture more intricate compositional structure. In contrast, a medium-sized model (64-dim embeddings, 3 layers, 3 heads) reached near-perfect **Boolean accuracy across all operator classes while keeping the parameter count and training time moderate, and increasing capacity further to the largest 6-layer, 124-dim model brought only marginal improvements on this dataset.**

The context length was varied by changing block_size from 128 down to 64 and 32, with a learned positional embedding of shape (block_size, n_embd) added to token embeddings so the transformer can distinguish positions within each expression. Because expressions in the dataset are short (at most a few operands and operators plus the = True/False result), even block_size = 32 comfortably covers the entire sequence, and experiments showed that reducing the context from 128 to 32 did not degrade Boolean accuracy while reducing memory usage and computation. Overall, these results support the conclusion that, for this Boolean reasoning task, a character-level transformer with a small context window and medium capacity provides the best trade-off between simplicity, efficiency, and accuracy, whereas more aggressive scaling in depth, width, or sequence length yields little additional benefit.

Hyperparameter configurations were defined to explore different model capacities:

Config	Params (M)	block_size	n_embd	n_head	n_layer	Dropout	Overall Boolean acc
1	1.123459	128	124	6	6	0.2	0.999
2	0.155283	64	64	3	3	0.2	0.923
3	0.007635	32	16	2	2	0.1	0.629

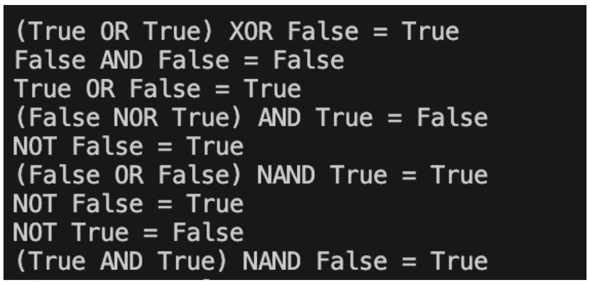
Table-04:Parameter counts and hyperparameters for Boolean GPT configurations, highlighting the trade-off between model size, context length, and performance. Models are listed from best-performing (top) to lowest-performing (bottom).

2.4. Operations and Equations

The final step was to analyse which Boolean operations and structures the model learns reliably, and where errors remain, using both quantitative metrics and qualitative examples.

Quantitatively, overall expression-level accuracy on train and validation is computed with evaluate_boolean_accuracy, and per-class accuracies are obtained by filtering validation lines with predicates like is_and_or_only, has_not, has_parentheses, and has_xor, then calling evaluate_subset on each subset. The results show that AND/OR-only expressions and those involving NOT are learned almost perfectly, with validation accuracy close to 100%, while expressions with parentheses (single-level nesting) also reach high but slightly lower accuracy, revealing the first signs of difficulty on more complex compositions. XOR expressions consistently have the lowest accuracy among the operator classes, especially for smaller models, indicating that the model sometimes treats XOR in an OR-like way on rarer patterns. Overall, these per-class results suggest that basic operations (AND, OR, NOT) and simple nested structures are mastered, whereas XOR and some nested combinations remain the most challenging cases.

Qualitative examples of success and failure
Using predict_bool, several representative examples can be examined:



```
(True OR True) XOR False = True
False AND False = False
True OR False = True
(False NOR True) AND True = False
NOT False = True
(False OR False) NAND True = True
NOT False = True
NOT True = False
(True AND True) NAND False = True
```

Fig-03: Boolean GPT sample outputs

For some expressions like "(True XOR True) AND False = ", the model may occasionally generate a result inconsistent with the true evaluation (False), especially with smaller models.
Rare or ambiguous outputs:
In a few cases, the generated continuation after = does not start with "True" or "False", causing predict_bool to return "" and the example to be discarded during accuracy computation. These failures suggest that the model sometimes drifts into non-Boolean text, particularly if the context is unusual or the model is over- or under-trained.

Interpretation
The model reliably generalises the semantics of AND, OR, and NOT across both training and validation data, as shown by near-perfect accuracy on these subsets.

Single-level nested expressions involving parentheses are also learned well, although this structure is inherently more complex and performance is slightly more sensitive to model capacity.

XOR, especially when combined with parentheses, is the most challenging operator: its truth table is less intuitive than AND/OR, and it appears less frequently in the dataset, leading to a somewhat higher error rate.

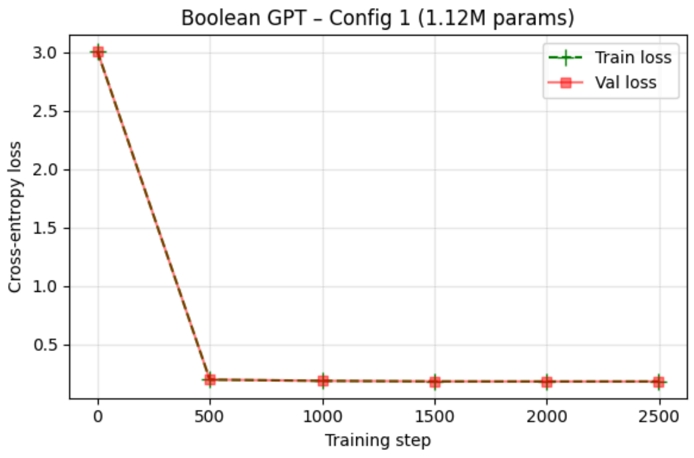


Fig-04: Boolean GPT Training Loss Vs Validation Loss

Overall, the transformer architecture is able to learn Boolean logic over the generated dataset to a high degree of accuracy, with remaining errors concentrated in the most complex and least frequent operation patterns.

3.Part 3 - Discussion

Math GPT and Boolean GPT share a common transformer backbone, but the tasks place very different demands on that architecture. For Boolean reasoning, a relatively small model with moderate depth and heads, standard ReLU activations, character‑level tokenisation, cross‑entropy loss, and simple dropout regularisation was already sufficient to achieve near‑saturated performance, **because the vocabulary is small and the space of valid patterns is comparatively limited.** In contrast, Math GPT needed much larger embeddings and significantly more parameters **to cope with the richer variety of digit sequences and operations,** even though the nominal architecture (layers, heads, residual connections, cross‑entropy objective) remained the same.

This contrast highlights which architectural elements were broadly appropriate for both tasks and which required adaptation. **Depth, multi‑head attention, and the basic transformer block design worked well for symbolic sequence processing in both cases**: stacking several layers allowed the models to integrate information across an expression, and the same cost function (token‑level cross‑entropy) was adequate as a training signal for both Boolean outputs and numeric strings. However, **embedding size and total capacity were clearly taskdependent**: Boolean GPT could remain compact while still generalising well, whereas Math GPT needed larger representations to approximate multi‑step arithmetic algorithms. Likewise, tokenisation and input/output design were not interchangeable: digit‑level tokenisation was crucial for arithmetic, while character‑level encoding of logical symbols was enough for Boolean formulae.

Training dynamics also differed. It was noticeably easier to train a strong model on the Boolean task, because the smaller vocabulary and simpler pattern space meant the model could quickly learn operator semantics and precedence and then generalise across similar expressions. By contrast, Math GPT faced a larger and more varied distribution of expressions, making optimisation slower and leaving higher residual loss even after substantial training. Experimentally, attempts to simplify pattern recognition via word‑level tokenisation (e.g. treating whole tokens like “True”, “False”, or multi‑digit numbers as units) did not help and in practice failed: the model struggled more to capture fine‑grained structure, especially in arithmetic where digit‑level dependencies (carries, borrows) are essential.

These observations tie directly to the nature of the tasks and to the fundamental limitations of this architecture. The models do not actually “use logic” or execute arithmetic algorithms; instead, they learn statistical regularities over symbol sequences and output what is likely given the training distribution. This pattern‑matching behaviour is often enough for in‑distribution Boolean expressions and many arithmetic examples, but it exposes clear limitations: Boolean GPT can fail on rare operator combinations or deeply nested parentheses, and Math GPT can produce off‑by‑one errors, incorrect carries/borrows, or brittle behaviour on longer or less typical expressions. These failures are not bugs in a symbolic engine; they are a direct consequence of asking a pattern recogniser to emulate precise reasoning.

Although both models achieve strong in‑distribution performance, their architecture limits algorithmic generalisation. The Math GPT, in particular, learns pattern‑based computation rather than true arithmetic reasoning, leading to errors on longer or unseen expressions. This highlights a fundamental challenge in using small transformer models for symbolic or rule‑based tasks — they infer statistical regularities rather than explicitly executing learned algorithms.

Model	**Best Configuration**	**Key Metric (Val Accuracy)**	**Main Failure Mode**
Math GPT	8 layers, 8 heads, 256‑dim embedding	0.754	Struggles with multi‑digit multiplication and generalising unseen numeric patterns

Boolean GPT	6 layers, 6 heads, 124-dim embedding	0.999	Minor inconsistencies in rare XOR/NAND combinations

Table-05: Model architecture details for the evaluated GPT configurations.

Regularisation and training tricks can mitigate some issues but do not fundamentally change this. Similar dropout settings worked for both tasks, helping prevent overfitting, yet arithmetic still showed more residual error because the underlying algorithmic complexity is higher. Increasing model size improved Math GPT's accuracy but at a clear computational cost, and simply scaling up does not guarantee robust generalisation to harder or out-of-distribution problems. Likewise, small architectural tweaks alone (without rethinking tokenisation or supervision) offered diminishing returns.

Given these limitations, more creative strategies become important. One promising direction is to let the model identify the type of task it is facing and then trigger an external, specialised procedure: for example, a learned front-end that recognises a Boolean or arithmetic query and then calls a symbolic logic solver or a numeric calculator to perform the actual reasoning. In this hybrid view, the transformer focuses on understanding and formatting the problem, while a separate script or tool executes the precise computation. Another direction is to integrate more explicit intermediate supervision (e.g. step-by-step reasoning traces) or structured modules that better reflect the underlying algorithms, rather than relying purely on end-to-end token prediction. Together, these reflections show that while a shared transformer architecture can be pushed quite far on both Boolean and arithmetic tasks, its strengths and shortcomings differ across them, and fully reliable reasoning likely requires going beyond "minor changes" to the provided model towards architectures and training setups that explicitly acknowledge the structure of the tasks.

References :

- Karpathy, A. "nanoGPT: A simple GPT training script." GitHub repository,
- https://github.com/karpathy/nanogpt
- , accessed January 2026.
- Karpathy, A. "Neural Networks: Zero to Hero." YouTube lecture series, 2022–2023.
- Course nanoGPT starter code and assignment handout, Machine Learning module, Trinity College Dublin, 2025–2026.
- Vaswani, A., Shazeer, N., Parmar, N., et al. "Attention Is All You Need." Advances in Neural Information Processing Systems (NeurIPS), 2017.
- Goodfellow, I., Bengio, Y., & Courville, A. *Deep Learning*. MIT Press, 2016.

APPENDIX

Prompt output pairs:

Config 1 Math GPT:

```
6.33704 M parameters
step 0: train loss 2.8460, val loss 2.8449
step 500: train loss 1.6302, val loss 1.6350
step 1000: train loss 1.5115, val loss 1.5183
step 1500: train loss 1.4245, val loss 1.4344
step 2000: train loss 1.3626, val loss 1.3788
step 2500: train loss 1.2804, val loss 1.2997
step 3000: train loss 1.2587, val loss 1.2871
step 3499: train loss 1.2397, val loss 1.2906

Eval steps: [0, 500, 1000, 1500, 2000, 2500, 3000, 3499]
Train CE losses: [2.846048593521118, 1.6302372217178345, 1.5114812850952148,
1.4245189428329468, 1.3626307249069214, 1.2803877592086792, 1.2586960792541504,
1.2396759986877441]
Val CE losses: [2.8448541164398193, 1.6349636316299438, 1.5183308124542236,
1.4344412088394165, 1.3788425922393799, 1.2997413873672485, 1.2871246337890625,
1.2906100749969482]
]
Sample generated text:

2-5=-3
5/9=0
22-78=-56
21/82=0
6-3=3
0/5=0
6/88=0
5*4=20
6/4=1
25*33=768
41-28=23
21/6=2
77*76=5902


Per-op VAL accuracies:
Val add (+): n=1002, acc=0.849
Val sub (-): n=1005, acc=0.740
Val mul (*): n=1005, acc=0.544
Val div (/): n=988, acc=0.874
Val math accuracy:   0.754
```

Config 2 Math GPT:

```
4.759056 M parameters
step 0: train loss 2.8119, val loss 2.8121
step 500: train loss 1.5858, val loss 1.5923
step 1000: train loss 1.4238, val loss 1.4343
step 1500: train loss 1.3068, val loss 1.3233
step 2000: train loss 1.2768, val loss 1.3031
step 2499: train loss 1.2446, val loss 1.2971
Eval steps:
Train CE losses: [2.811915636062622, 1.585811734199524, 1.4238333702087402,
1.3068002462387085, 1.2768160104751587, 1.2446478605270386]
Val CE losses: [2.8120880126953125, 1.5922698974609375, 1.4342747926712036,
1.32332181930542, 1.3031206130981445, 1.2971224784851074]
Sample generated text:
40/41=1
9-5=4
9+9=18
1-3=-2
6-9=-3
24=8
9954=5676
4+4=8
8+3=11
24/86=0
63*98=6334
9+3=12
9/4=2
14/
Per-op VAL accuracies:
Val add (+): n=1002, acc=0.854
Val sub (-): n=1005, acc=0.728
Val mul (*): n=1005, acc=0.555
Val div (/): n=988, acc=0.876
Val math accuracy: 0.756
```

Config 3 Math GPT:

```
1.193744 M parameters
step 0: train loss 2.8267, val loss 2.8256
step 500: train loss 1.6687, val loss 1.6754
step 1000: train loss 1.5711, val loss 1.5783
step 1500: train loss 1.4759, val loss 1.4809
step 2000: train loss 1.4092, val loss 1.4187
step 2500: train loss 1.3370, val loss 1.3541
step 3000: train loss 1.2877, val loss 1.3031
step 3500: train loss 1.2753, val loss 1.2926
step 4000: train loss 1.2594, val loss 1.2861
step 4500: train loss 1.2510, val loss 1.2853
step 4999: train loss 1.2367, val loss 1.2851
Eval steps: [0, 500, 1000, 1500, 2000, 2500, 3000, 3500, 87222719192505,
1.5710967779159546, 1.4758849143981934, 1.4091601371765137, 1.3370156288146973,
1.2877181768417358, 1.2752711772918701, 1.2593785524368286, 1.2509936094284058,
1.2367205619812012]
Val CE losses: [2.8255927562713623, 1.6753970384597778, 1.578346848487854,
1.480902075767517, 1.4187146425247192, 1.3541061878204346, 1.3030519485473633,
1.2926087379455566, 1.2860848903656006, 1.2853256464004517, 1.285077691078186]
Sample generated text:
6713=961
16+31=47
4+34=28
27-3=14
8+95=102
3/6=0
51+54=105
18=8
6*5=30
8+2=10
0-2=-2
9/6=1
20-98=-
Per-op VAL accuracies:
Val add (+): n=1002, acc=0.858
Val sub (-): n=1005, acc=0.784
Val mul (*): n=1005, acc=0.508
Val div (/): n=988, acc=0.883
Val math accuracy: 0.767
```

Config 1 Boolean GPT :

```
1.123459 M parameters
step 0: train loss 3.0054, val loss 3.0051
step 500: train loss 0.1966, val loss 0.1969
step 1000: train loss 0.1865, val loss 0.1866
step 1500: train loss 0.1818, val loss 0.1822
step 2000: train loss 0.1813, val loss 0.1812
step 2499: train loss 0.1814, val loss 0.1812

Eval steps: [0, 500, 1000, 1500, 2000, 2499]
Train CE losses: [3.0053887367248535, 0.19662044942378998, 0.18649402260780334,
0.18180711567401886, 0.1812601089477539, 0.1813846230506897]
```

```
Val CE losses: [3.0050902366638184, 0.19694167375564575, 0.1865520477294922,
0.18215028941631317, 0.1811891496181488, 0.18116340041160583]


(True OR True) XOR False = True
False AND False = False
True OR False = True
(False NOR True) AND True = False
NOT False = True
(False OR False) NAND True = True
NOT False = True
NOT True = False
(True AND True) NAND False = True
NOT True = False
NOT True = False
(False OR False) XOR True = True
NOT True = False
False XOR True = True
(False AND True) AND True = False
(True XOR False) NAND False = True
(True NOR True) AND False = False
NOT False = True
False NOR True = False
NOT True = False
NOT

Per-class VAL accuracies:
Val AND/OR only: n=619, accuracy=0.998
Val with NOT: n=610, accuracy=1.000
Val with parentheses: n=597, accuracy=0.997
Val with XOR: n=389, accuracy=0.995
Val NAND/NOR only: n=306, accuracy=1.000
Val with NAND: n=366, accuracy=1.000
Val with NOR: n=368, accuracy=1.000
Boolean accuracy: 0.999
```

Config 2 Boolean GPT :

```
0.155283 M parameters
step 0: train loss 2.9903, val loss 2.9895
step 500: train loss 0.2250, val loss 0.2252
step 1000: train loss 0.2075, val loss 0.2075
step 1500: train loss 0.1996, val loss 0.1998
step 2000: train loss 0.1977, val loss 0.1979
step 2499: train loss 0.1969, val loss 0.1972

Eval steps: [0, 500, 1000, 1500, 2000, 2499]
Train CE losses: [2.99031925201416, 0.22504253685474396, 0.20747461915016174,
0.19961266219615936, 0.19765041768550873, 0.19690443575382233]
Val CE losses: [2.989544630050659, 0.22516994178295135, 0.20750582218170166,
0.19976404309272766, 0.19793285429477692, 0.1971801519393921]


True AND True = True
(True OR True) NAND True = False
True NAND True = False
(False NAND True) OR False = True
(False NAND False) OR False = True
(True AND True) AND True = False
NOT True = False
NOT False = True
```

```
True NOR True = False
NOT True = False
NOT False = True
(False AND False) NOR False = True
NOT False = True
True NOR True = False
(False AND False) NAND True = False
NOT True = False
T True NOR True = False
True AND False = False
True OR True = True
NOT False = False
NOT False = True
NO

Per-class VAL accuracies:
Val AND/OR only: n=619, accuracy=0.971
Val with NOT: n=610, accuracy=0.987
Val with parentheses: n=597, accuracy=0.777
Val with XOR: n=389, accuracy=0.825
Val NAND/NOR only: n=306, accuracy=0.977
Val with NAND: n=366, accuracy=0.883
Val with NOR: n=368, accuracy=0.842
Boolean accuracy: 0.923
```

Config 3 Boolean GPT:

```
0.007635 M parameters
step 0: train loss 2.9348, val loss 2.9349
step 500: train loss 0.3930, val loss 0.3937
step 1000: train loss 0.2679, val loss 0.2679
step 1500: train loss 0.2504, val loss 0.2503
step 2000: train loss 0.2417, val loss 0.2414
step 2499: train loss 0.2364, val loss 0.2361

Eval steps: [0, 500, 1000, 1500, 2000, 2499]
Train CE losses: [2.9347591400146484, 0.393029123544693, 0.2678958773612976,
0.2503589689731598, 0.24173368513584137, 0.23640719056129456]
Val CE losses: [2.9349303245544434, 0.3937244117259979, 0.2679215669631958,
0.2503241300582886, 0.24135944247245789, 0.23609711229801178]


True AND True = True
NOT True = False
True XOR False = False
(False NAND False) XOR False = True
True NOR True = True
(True OR True = False
(True NOR False) AND False = True
True NAND True = False
(True NOR True) NXOR True = False
NOT True = True
False AND False) AND True = TrueDrue
(False NOR False) XOR False = True
False AND True = False
NOT True = True
NOT True = False
(True AND True) X AND False = False
False NAND False = True
False NAND True = False
(True NAND True) XOR False = False
```

```
True A

Per-class VAL accuracies:
Val AND/OR only: n=619, accuracy=0.498
Val with NOT: n=610, accuracy=0.921
Val with parentheses: n=597, accuracy=0.493
Val with XOR: n=389, accuracy=0.499
Val NAND/NOR only: n=306, accuracy=0.502
Val with NAND: n=366, accuracy=0.534
Val with NOR: n=368, accuracy=0.467
Boolean accuracy: 0.629
```

math_gpt.py

```
import torch
import torch.nn as nn
from torch.nn import functional as F
import random

# # hyperparameters
# #Config 1 = 6.33704 M parameters
batch_size = 64
max_iters = 3500
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'mps'
eval_iters = 200
n_embd = 256
n_head = 8
n_layer = 8
dropout = 0.2
block_size = 64  # maximum context length for predictions

# Config 2 - 1.193744 M parameters
# batch_size = 64   # how many independent sequences will we process in parallel?
# max_iters = 5000
# eval_interval = 500
# learning_rate = 3e-4
# device = 'cuda' if torch.cuda.is_available() else 'mps'
# eval_iters = 200
# n_embd = 128
# n_head = 6
# n_layer = 6
# dropout = 0.1
# block_size = 64  # maximum context length for predictions

# Config 3 - 4.759056 M parameters
# batch_size = 64   # how many independent sequences will we process in parallel?
# max_iters = 2500
# eval_interval = 500
# learning_rate = 3e-4
# device = 'cuda' if torch.cuda.is_available() else 'mps'
# eval_iters = 200
# n_embd = 256
# n_head = 4
# n_layer = 6
# dropout = 0.1
# block_size = 64  # maximum context length for predictions
```

```python
# ------------
torch.manual_seed(1337)
random.seed(1337)

# 1. Load raw lines
with open(
   "math_simple.txt",
   "r",
   encoding="utf-8",
) as f:
   lines = [ln.strip() for ln in f.readlines() if ln.strip()]

random.shuffle(lines)
random.shuffle(lines)
full_text = "\n".join(lines)
chars = sorted(list(set(full_text)))
vocab_size = len(chars)
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for i, ch in enumerate(chars)}

encode = lambda s: [stoi[c] for c in s]
decode = lambda l: "".join([itos[i] for i in l])

n_lines = int(0.8 * len(lines))   # 80% train, 20% val
train_lines = lines[:n_lines]
val_lines   = lines[n_lines:]

train_text = "\n".join(train_lines)
val_text   = "\n".join(val_lines)

train_data = torch.tensor(encode(train_text), dtype=torch.long)
val_data   = torch.tensor(encode(val_text), dtype=torch.long)

# data loading
def get_batch(split: str):
   data_split = train_data if split == "train" else val_data
   ix = torch.randint(len(data_split) - block_size, (batch_size,))
   x = torch.stack([data_split[i : i + block_size] for i in ix])
   y = torch.stack([data_split[i + 1 : i + block_size + 1] for i in ix])
   x, y = x.to(device), y.to(device)
   return x, y


@torch.no_grad()
def estimate_loss():
   out = {}
   model.eval()
   for split in ["train", "val"]:
       losses = torch.zeros(eval_iters)
       for k in range(eval_iters):
           X, Y = get_batch(split)
           logits, loss = model(X, Y)
           losses[k] = loss.item()
       out[split] = losses.mean()
   model.train()
   return out


class Head(nn.Module):
   """One head of self-attention"""

   def __init__(self, head_size):
       super().__init__()
```

```
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer("tril", torch.tril(torch.ones(block_size, block_size)))
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B, T, C = x.shape
        k = self.key(x)
        q = self.query(x)
        wei = q @ k.transpose(-2, -1) * k.shape[-1] ** -0.5
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float("-inf"))
        wei = F.softmax(wei, dim=-1)
        wei = self.dropout(wei)
        v = self.value(x)
        out = wei @ v
        return out


class MultiHeadAttention(nn.Module):
    """Multiple heads of self-attention in parallel"""

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out


class FeedFoward(nn.Module):
    """A simple linear layer followed by a non-linearity"""

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)


class Block(nn.Module):
    """Transformer block: communication followed by computation"""

    def __init__(self, n_embd, n_head):
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x


class GPTLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(
            *[Block(n_embd, n_head=n_head) for _ in range(n_layer)]
        )
        self.ln_f = nn.LayerNorm(n_embd)
        self.lm_head = nn.Linear(n_embd, vocab_size)
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape
        tok_emb = self.token_embedding_table(idx)  # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device))  # (T,C)
        x = tok_emb + pos_emb                     # (B,T,C)
        x = self.blocks(x)                        # (B,T,C)
        x = self.ln_f(x)                          # (B,T,C)
        logits = self.lm_head(x)                  # (B,T,vocab_size)

        if targets is None:
            return logits, None

        B, T, C = logits.shape
        logits = logits.view(B * T, C)
        targets = targets.view(B * T)
        loss = F.cross_entropy(logits, targets)
        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            idx_cond = idx[:, -block_size:]
            logits, _ = self(idx_cond)
            logits = logits[:, -1, :]   # (B,C)
            probs = F.softmax(logits, dim=-1)
            idx_next = torch.multinomial(probs, num_samples=1)  # (B,1)
            idx = torch.cat((idx, idx_next), dim=1)             # (B,T+1)
        return idx


# instantiate model and optimizer
model = GPTLanguageModel()
m = model.to(device)

print(sum(p.numel() for p in m.parameters()) / 1e6, "M parameters")

optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

# training loop
```

```
train_loss_curve = []
val_loss_curve = []
eval_steps = []
best_val = float("inf")

for iter in range(max_iters):
   # every once in a while evaluate the loss on train and val sets
   if iter % eval_interval == 0 or iter == max_iters - 1:
       losses = estimate_loss()
       train_loss = losses["train"].item()
       val_loss   = losses["val"].item()
       train_loss_curve.append(train_loss)
       val_loss_curve.append(val_loss)
       eval_steps.append(iter)
       print(f"step {iter}: train loss {train_loss:.4f}, val loss {val_loss:.4f}")

       if losses["val"] < best_val:
           best_val = losses["val"]
           torch.save(model.state_dict(), "model_weights_part1.pth")

   # sample a batch of data
   xb, yb = get_batch("train")

   # evaluate the loss and update
   logits, loss = model(xb, yb)
   optimizer.zero_grad(set_to_none=True)
   loss.backward()
   optimizer.step()

print("\nEval steps:", eval_steps)
print("Train CE losses:", train_loss_curve)
print("Val CE losses:", val_loss_curve)

best_state = torch.load("model_weights_part1.pth", map_location=device)
model.load_state_dict(best_state)
m = model.to(device)

# generate targeted samples after training
print("Sample generated text:")
context = torch.zeros((1, 1), dtype=torch.long, device=device)
sample = decode(m.generate(context, max_new_tokens=100)[0].tolist())
print(sample)

# helper to complete a specific expression prefix
def complete(expr_prefix: str, max_new_tokens: int = 5):
   idx = torch.tensor([encode(expr_prefix)], dtype=torch.long, device=device)
   out = m.generate(idx, max_new_tokens=max_new_tokens)[0].tolist()
   return decode(out)

def evaluate_math_accuracy(lines, max_samples: int = None) -> float:
   total = 0
   correct = 0
   for line in lines:
       line = line.strip()
       if not line or "=" not in line:
           continue
       expr, truth = line.split("=", 1)
       lhs = expr.strip() + "="
       truth = truth.strip()
       pred = predict_result(lhs)
       if pred == "":
           continue
       total += 1
```

```
        if pred == truth:
            correct += 1
        if max_samples is not None and total >= max_samples:
            break
    return 0.0 if total == 0 else correct / total


@torch.no_grad()
def predict_result(lhs: str, max_new_tokens: int = 5) -> str:
    """
    lhs: expression prefix including '=', e.g. '12+7='
    returns: predicted integer result as string (e.g. '19'), or '' if unclear
    """
    model.eval()
    idx = torch.tensor([encode(lhs)], dtype=torch.long, device=device)
    out = m.generate(idx, max_new_tokens=max_new_tokens)[0].tolist()
    full = decode(out)

    # Take what comes after '='
    if "=" in full:
        ans_part = full.split("=", 1)[1]
    else:
        ans_part = full[len(lhs):]
    ans_part = ans_part.strip()

    # Extract leading integer (optional leading '-')
    res = ""
    for ch in ans_part:
        if ch.isdigit() or (ch == "-" and not res):
            res += ch
        else:
            break
    return res


def filter_lines(lines, predicate):
    return [ln for ln in lines if predicate(ln)]

def evaluate_subset(name, lines_subset, max_samples=None):
    acc = evaluate_math_accuracy(lines_subset, max_samples=max_samples)
    print(f"{name}: n={len(lines_subset)}, acc={acc:.3f}")

def is_add(line: str) -> bool:
    return "+" in line and "*" not in line and "/" not in line and "-" not in line

def is_sub(line: str) -> bool:
    return "-" in line and "*" not in line and "/" not in line and "+" not in line

def is_mul(line: str) -> bool:
    return "*" in line

def is_div(line: str) -> bool:
    return "/" in line


val_add = filter_lines(val_lines, is_add)
val_sub = filter_lines(val_lines, is_sub)
val_mul = filter_lines(val_lines, is_mul)
val_div = filter_lines(val_lines, is_div)

print("\nPer-op VAL accuracies:")
evaluate_subset("Val add (+)", val_add)
evaluate_subset("Val sub (-)", val_sub)
evaluate_subset("Val mul (*)", val_mul)
evaluate_subset("Val div (/)", val_div)



# train_acc = evaluate_math_accuracy(train_lines)
val_acc   = evaluate_math_accuracy(val_lines)

# print(f"Train math accuracy: {train_acc:.3f}")
print(f"Val math accuracy:   {val_acc:.3f}")

# print("\nTargeted samples by operation:")
# for prefix in ["3+5=", "9-4=", "7*8=", "8/3="]:
#     print(complete(prefix))
```

boolean_gpt.py

```python
import torch
import torch.nn as nn
from torch.nn import functional as F

# hyperparameter configs

#Config 1 = 1.123459 M parameters
batch_size = 64
block_size = 128 # what is the maximum context length for predictions?
max_iters = 2500
eval_interval = 500
learning_rate = 3e-4
device = 'cuda' if torch.cuda.is_available() else 'mps'
eval_iters = 200
n_embd = 124
n_head = 6
n_layer = 6
dropout = 0.2


# Config - 2 0.155283 M parameters
# batch_size = 64
# block_size = 64 # what is the maximum context length for predictions?
# max_iters = 2500
# eval_interval = 500
# learning_rate = 3e-4
# device = 'mps' if torch.backends.mps.is_available() else 'cpu'
# eval_iters = 200
# n_embd = 64
# n_head = 3
# n_layer = 3
# dropout = 0.2


#Config - 3
# batch_size = 64 # how many independent sequences will we process in parallel?
# block_size = 32 # what is the maximum context length for predictions?
# max_iters = 2500
# eval_interval = 500
# learning_rate = 3e-4
# device = 'cuda' if torch.cuda.is_available() else 'mps'
# eval_iters = 200
# n_embd = 16
# n_head = 2
# n_layer = 2
# dropout = 0.1
# ------------

# print("CUDA available:", torch.cuda.is_available())
# print("MPS available:", torch.backends.mps.is_available())
# print("MPS built:", torch.backends.mps.is_built())
torch.manual_seed(1337)

# Train and test splits
torch.manual_seed(1337)

# 1. Load raw lines
with
open('/Users/saieeshwar/Projects/College/MachineLearning/finalAssignment_GPT/bool_all_new.
txt',
         'r', encoding='utf-8') as f:
```

```
   lines = [ln.strip() for ln in f.readlines() if ln.strip()]

# 2. Build vocab from full text (all lines joined)
full_text = "\n".join(lines)
chars = sorted(list(set(full_text)))
vocab_size = len(chars)
stoi = {ch: i for i, ch in enumerate(chars)}
itos = {i: ch for i, ch in enumerate(chars)}
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])

# 3. Line-level split (for evaluation later)
n_lines = int(0.9 * len(lines))   # 90% train, 10% val
train_lines = lines[:n_lines]
val_lines   = lines[n_lines:]

# 4. Build character-level tensors *after* the split
train_text = "\n".join(train_lines)
val_text   = "\n".join(val_lines)

train_data = torch.tensor(encode(train_text), dtype=torch.long)
val_data   = torch.tensor(encode(val_text), dtype=torch.long)

# 5. get_batch unchanged, but now uses new train_data / val_data
def get_batch(split):
   # generate a small batch of data of inputs x and targets y
   data = train_data if split == 'train' else val_data
   ix = torch.randint(len(data) - block_size, (batch_size,))
   x = torch.stack([data[i:i+block_size] for i in ix])
   y = torch.stack([data[i+1:i+block_size+1] for i in ix])
   x, y = x.to(device), y.to(device)
   return x, y

@torch.no_grad()
def estimate_loss():
   out = {}
   model.eval()
   for split in ['train', 'val']:
       losses = torch.zeros(eval_iters)
       for k in range(eval_iters):
           X, Y = get_batch(split)
           logits, loss = model(X, Y)
           losses[k] = loss.item()
       out[split] = losses.mean()
   model.train()
   return out

class Head(nn.Module):
   """ one head of self-attention """

   def __init__(self, head_size):
       super().__init__()
       self.key = nn.Linear(n_embd, head_size, bias=False)
       self.query = nn.Linear(n_embd, head_size, bias=False)
       self.value = nn.Linear(n_embd, head_size, bias=False)
       self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))

       self.dropout = nn.Dropout(dropout)

   def forward(self, x):
       # input of size (batch, time-step, channels)
       # output of size (batch, time-step, head size)
       B,T,C = x.shape
```

```
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B, hs, T) -> (B,
T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)

class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

class GPTLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in
range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

        # better init, not covered in the original GPT video, but important, will cover in
followup video
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = GPTLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')
```

```
# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)


train_loss_curve = []
val_loss_curve = []
eval_steps = []

for iter in range(max_iters):

   # every once in a while evaluate the loss on train and val sets
   if iter % eval_interval == 0 or iter == max_iters - 1:
       losses = estimate_loss()
       train_loss = losses['train'].item()
       val_loss = losses['val'].item()
       train_loss_curve.append(train_loss)
       val_loss_curve.append(val_loss)
       eval_steps.append(iter)
       print(f"step {iter}: train loss {train_loss:.4f}, val loss {val_loss:.4f}")

   # sample a batch of data
   xb, yb = get_batch('train')

   # evaluate the loss
   logits, loss = model(xb, yb)
   optimizer.zero_grad(set_to_none=True)
   loss.backward()
   optimizer.step()


   # # sample a batch of data
   # xb, yb = get_batch('train')

   # evaluate the loss
   logits, loss = model(xb, yb)
   optimizer.zero_grad(set_to_none=True)
   loss.backward()
   optimizer.step()

MODEL_PATH = "model_weights_part2.pth"
torch.save(model.state_dict(), MODEL_PATH)

#Per Class Accuracy
def filter_lines_by_predicate(lines, predicate):
   return [ln for ln in lines if predicate(ln)]

def evaluate_subset(name, lines_subset, max_samples=None):
   acc = evaluate_boolean_accuracy(lines_subset, max_samples=max_samples)
   print(f"{name}: n={len(lines_subset)}, accuracy={acc:.3f}")

# Only AND/OR (no NOT, no XOR, no parentheses)
def is_and_or_only(line: str) -> bool:
   s = line.upper()
   return ("AND" in s or "OR" in s) and \
          ("NOT" not in s) and ("XOR" not in s) and \
          ("(" not in s and ")" not in s)

# Expressions with NOT
def has_not(line: str) -> bool:
   return "NOT" in line.upper()

# Expressions with nested parentheses (at least one '(' and ')')
def has_parentheses(line: str) -> bool:
   s = line
   return "(" in s and ")" in s

# XOR expressions
def has_xor(line: str) -> bool:
   return "XOR" in line.upper()

def is_nand_nor_only(line: str) -> bool:
   s = line.upper()
   return ("NAND" in s or "NOR" in s) and \
          ("NOT" not in s) and ("XOR" not in s) and \
          ("(" not in s and ")" not in s)

def has_nand(line: str) -> bool:
   return "NAND" in line.upper()

def has_nor(line: str) -> bool:
   return "NOR" in line.upper()




print()
print("Eval steps:", eval_steps)
print("Train CE losses:", train_loss_curve)
print("Val CE losses:", val_loss_curve)
print()

# generate from the model
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
# open('more.txt', 'w').write(decode(m.generate(context,
max_new_tokens=10000)[0].tolist()))

#evaluation meetrics
@torch.no_grad()
def predict_bool(lhs: str, max_new_tokens: int = 5) -> str:
   """
   lhs: expression prefix including ' = ', e.g. 'True AND False = '
   returns: 'True' or 'False' as predicted by the model (string), or '' if unclear
   """
   model.eval()
   idx = torch.tensor([encode(lhs)], dtype=torch.long, device=device)
   for _ in range(max_new_tokens):
       idx_cond = idx[:, -block_size:]
       logits, _ = model(idx_cond)
       logits = logits[:, -1, :]
       probs  = torch.softmax(logits, dim=-1)
       idx_next = torch.multinomial(probs, num_samples=1)
       idx = torch.cat((idx, idx_next), dim=1)
   full = decode(idx[0].tolist())
   # take what comes after ' = '
   if " = " in full:
       ans_part = full.split(" = ", 1)[1]
   elif "=" in full:
       ans_part = full.split("=", 1)[1]
   else:
       ans_part = full
   ans_part = ans_part.strip()
   if ans_part.startswith("True"):
       return "True"
   if ans_part.startswith("False"):
       return "False"
```

```
    return ""

def evaluate_boolean_accuracy(lines, max_samples: int = None) -> float:
    total = 0
    correct = 0
    for line in lines:
        line = line.strip()
        if not line or "=" not in line:
            continue
        expr, truth = line.split("=", 1)
        lhs = expr.strip() + " = "
        truth = truth.strip()
        pred = predict_bool(lhs)
        if pred == "":
            continue
        total += 1
        if pred == truth:
            correct += 1
        if max_samples is not None and total >= max_samples:
            break
    return 0.0 if total == 0 else correct / total


# ---- Per-class accuracy on TRAIN ----
# train_and_or_only = filter_lines_by_predicate(train_lines, is_and_or_only)
# train_with_not = filter_lines_by_predicate(train_lines, has_not)
# train_with_paren = filter_lines_by_predicate(train_lines, has_parentheses)
# train_with_xor = filter_lines_by_predicate(train_lines, has_xor)

# print("\nPer-class TRAIN accuracies:")
# evaluate_subset("Train AND/OR only", train_and_or_only)
# evaluate_subset("Train with NOT", train_with_not)
# evaluate_subset("Train with parentheses", train_with_paren)
# evaluate_subset("Train with XOR", train_with_xor)

# ---- Per-class accuracy on VAL ----
val_and_or_only = filter_lines_by_predicate(val_lines, is_and_or_only)
val_with_not = filter_lines_by_predicate(val_lines, has_not)
val_with_paren = filter_lines_by_predicate(val_lines, has_parentheses)
val_with_xor = filter_lines_by_predicate(val_lines, has_xor)
val_nand_nor_only = filter_lines_by_predicate(val_lines, is_nand_nor_only)
val_with_nand = filter_lines_by_predicate(val_lines, has_nand)
val_with_nor  = filter_lines_by_predicate(val_lines, has_nor)

print("\nPer-class VAL accuracies:")
evaluate_subset("Val AND/OR only", val_and_or_only)
evaluate_subset("Val with NOT", val_with_not)
evaluate_subset("Val with parentheses", val_with_paren)
evaluate_subset("Val with XOR", val_with_xor)
evaluate_subset("Val NAND/NOR only", val_nand_nor_only)
evaluate_subset("Val with NAND", val_with_nand)
evaluate_subset("Val with NOR", val_with_nor)

# # evaluate on train and val splits
# train_acc = evaluate_boolean_accuracy(train_lines)
# print(f"Train Boolean accuracy: {train_acc:.3f}")

accuracy = evaluate_boolean_accuracy(val_lines)
print(f"Boolean accuracy: {accuracy:.3f}")


# evaluate on the full dataset or a separate test file
# evaluate on a file (e.g. same bool_all.txt or a held-out test file)
# acc =
evaluate_boolean_accuracy('/Users/saieeshwar/Projects/College/MachineLearning/finalAssignm
ent_GPT/bool_all.txt')
# print(f"Expression-level Boolean accuracy: {acc:.3f}")

# acc = evaluate_boolean_accuracy(val_data)
# print(f"Expression-level Boolean accuracy: {acc:.3f}")
```