Transformer Addition

Jenna Kainic

Task Setup

Problem formulation

Aim: to study whether a small Transformer can learn multi-digit integer addition from input–output examples, and how this knowledge generalizes beyond the training distribution.

We can frame the task as a causal language modeling problem, where the target sum is a continuation of the character sequence representing the summation problem ("a + b ="). Given this representation, we can use an autoregressive (causal) Transformer as the model, training with next-token prediction. This avoids unnecessary extra machinery (e.g. encoder-decoder) and has direct comparisons to other work on algorithmic tasks. I chose to default to k=3 digits as the smallest setting that reliably includes complexity like multi-step carry while keeping the space of possible inputs reasonable (the space of inputs grows exponentially in k). However, this is configurable: the architecture supports arbitrary k.

Model architecture

I used a GPT-2–style causal Transformer implemented using the HuggingFace transformers library.
Details:

- Layers: 6
- Hidden size: 256
- Attention heads: 8
- Context length: 64 tokens
- Dropout: 0.1
- Total parameters: ~6M

The model was chosen to be quickly trainable on a free Colab T4 GPU. I decided to start with a smaller model within the suggested range (6 rather than 12 layers), to first test if a smaller model was expressive enough to represent the aim. Especially with a model that can be quickly trained and evaluated, it is easy enough to add further complexity if needed. And ultimately the goal is to get a picture of small transformer capabilities, not to maximize performance.

Tokenization

We use a character-level tokenizer with a small vocabulary: <pad>, <bos>, <eos>, "0"–"9", " ", "+", "=". Addition is inherently digit-wise; this way we get explicitly positional reasoning and avoid ambiguities from subword tokenization. Model stops either at <eos> token or max_new_tokens which is set to k+4 to ensure we capture relevant failure modes (terminating early, off-by-some length errors, wrong digit predictions). However, it is worth noting that this will not capture degenerate looping; if we noticed many outputs using the max allotted tokens, this would be worth exploring.

Data Design

Training Distribution

The training data consists of 50k pairs of uniformly sampled 3-digit integers (plus 2k samples for validation). I considered a non-uniform distribution to get a more balanced representation of likely failure modes (especially carry terms). However, the uniform distribution is simpler to generate, and with enough samples we still get good coverage of possible failure

modes. Plus, we can design our evaluation scheme to interrogate particular failure modes, potentially giving us a better understanding of the system's inherent limitations.

Test Distributions

We define three independent test sets, each with its own distribution from which 7.5k pairs are generated.

1. In distribution (3 digits, uniform distribution, but generated separately from the train/val sets): this is a sanity check that the model has actually learned the trained task
2. Length generalization (4 digits, uniform distribution): this tests whether the model has learned a length-invariant algorithm, isolating extrapolation in sequence length
3. Carry distribution shift (3 digits, generated to ensure carry at every position): this tests robustness to different arithmetic structures, separating memorization from generalization by maximally stressing the dependency chain
    a. Note: these are randomly generated but constructed explicitly to ensure that for each position there will be carry, such that the distribution is not uniform over all possible pairs that meet this constraint.

NOTE: It arguably would have been a better idea to generate data to ensure no overlap between train/validation/test sets (relevant for the in-distribution test set). With 50k training pairs and 7.5k test pairs drawn from ~810k possibilities, expected overlap is roughly 50k/810k × 7.5k ≈ 460 samples, or about 6% of the test set. This probably not enough to significantly inflate accuracy, but still worth keeping in mind when evaluating results.

Training Details

Training uses a standard next-token cross-entropy loss, but with loss masking on tokens corresponding to the prompt, ensuring that the model is trained to produce correct answers without wasting capacity modeling the prompt. Data is pre-generated, and all random number generation is seeded to ensure reproducible results. Validation set used for monitoring purposes.

Optimization Details

Hyperparameters chosen for stability for small transformers, fast convergence on algorithmic tasks, and GPU constraints:

- Optimizer: AdamW
- Learning rate: 3e-4
- Warmup: 200 steps
- Effective batch size: 256
- Max steps: 8k
- Precision: FP16

Training runs in <10 minutes, though convergence plots plateau about halfway through, so early stopping would suffice.

Evaluation

We evaluate using free-running autoregressive generation rather than teacher-forced loss, because generation errors expose structured failure modes (e.g., carry-position mistakes, early termination) that loss alone can obscure. This is especially important here because errors in arithmetic tend to propagate; we want to see failure for what it is.

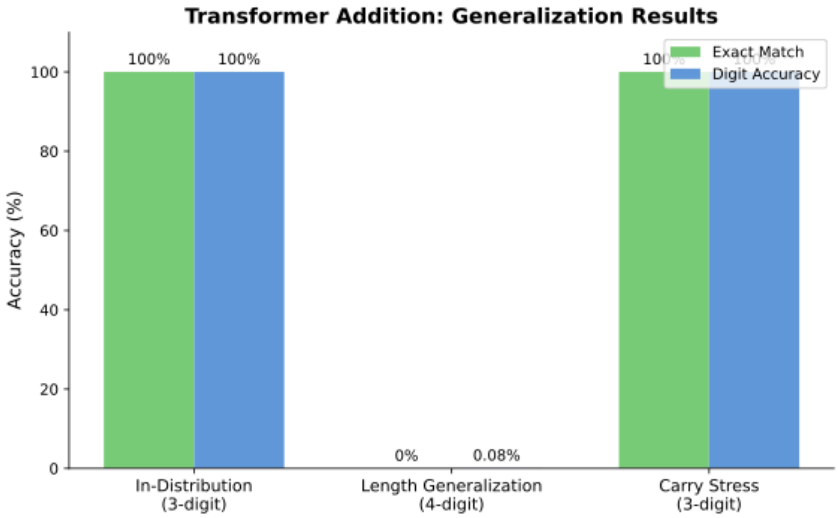
Metrics

1. Exact match accuracy: The fraction of examples where the generated answer string exactly matches the correct sum. This is the strictest metric and corresponds to task success.

2. Digits are aligned by least significant digit (right-aligned), and accuracy is computed per position. This can provide insight into where errors occur (e.g. carry positions).

Both metrics were computed on all of the previously described test sets.

Results



Interpretation

The model has learned addition, including carry propagation, within the trained sequence length. The carry-stress result rules out the hypothesis that it's relying on statistical shortcuts or memorization—if it were, the all-carry cases would be the first to break. Instead it handles the hardest arithmetic cases perfectly. We've also ruled out failure modes like early termination or degenerate looping, at least for this fixed length.

But it has learned a length-specific solution, not a length-invariant algorithm. The 4-digit collapse isn't "carries are hard at scale", because the carry test proves carries aren't the bottleneck. It's that the model has baked positional structure into its circuits in a way that doesn't extrapolate.

Looking more closely at 100 examples (a subset) of the length errors, we note:

- Failure mode: length collapse. Almost every prediction (94%) is exactly 2 digits (with the remaining being empty strings).
- Failure mode: garbage digits. E.g. 5454 + 7843 = 13297, but we get prediction "23" — not the first digits, last digits, or any obvious slice of the correct answer. This is a pattern that persists.

The predicted digits themselves show no systematic relationship to the correct answer: they do not correspond to any consistent slice of the target, nor to the inputs. This pattern suggests the model has learned a solution deeply entangled with length and thus likely with absolute positional information

Remaining Mysteries

- Why specifically 2 digits? The model almost always produces exactly 2-digit outputs on 4-digit inputs. This is unlikely to be a coincidence, but the mechanism isn't clear. One hypothesis: with 4-digit operands, the "=" token

appears 2 positions later than during training. If the model learned when to emit <eos> as an absolute position, the shifted input could cause it to start late and stop early, leaving a ~2 token output window.

- If the problem is absolute position, why does the model not have trouble with the max-carry test? Clearly there's *some* flexibility in output length. What's different about the 4-digit input case? My hypothesis: the position-specificity is primarily in input parsing, not output termination.
- Is there any structure in the wrong digits? I spot-checked and found no obvious pattern (not first digits, last digits, or sums of digit pairs). But a systematic analysis could reveal partial computation.
- Does the model generalize to shorter sequences? We tested k+1 but not k-1. The model might succeed, or might fail in an illuminating way (e.g. over-generation rather than truncation). Either outcome would help characterize the positional dependence, and whether failures are symmetric.
- How sharp is the transition? We compared k=3 (100%) to k=4 (0%). Is there a "graceful" degradation in between here, or is it binary? Testing k=3 inputs with one operand padded with a leading zero (e.g., "0123 + 456") or one 3-digit with one 4-digit number ("1230 + 456") might probe whether the model is sensitive to input length, digit alignment, or both.

Additional Experiments

- Test 2-digit inputs, mixed-length inputs, and padded inputs to map exact failure conditions.
- Train a variant where the answer is emitted least-significant-digit-first. This aligns carry propagation with autoregressive generation order, which could help length generalization.
- Train on a mixture of lengths and test whether this induces better length generalization.

Why is this so hard?

The difficulty isn't arithmetic per se: the carry-stress results prove the model can learn addition. It's also not in variable output length (for the same reason). The difficulty appears to be generalization in input parsing. Our architecture uses learned absolute positional embeddings. During training on fixed-format inputs, the model learns associations like "the units digit of the first operand is at position 2." This yields perfect in-distribution performance but creates brittle dependencies: when input format changes, every position-specific circuit misfires.

What's notable is that output length *isn't* the bottleneck. The model successfully produces 4-digit outputs on 3-digit inputs that overflow, so it has learned something like "keep emitting digits until the sum is complete." But this flexibility doesn't extend to input parsing.

Additionally, the task requires information to flow right-to-left (carries propagate from units toward higher places), but autoregressive generation flows left-to-right. Therefore, the full sum must effectively be resolved before the first token is emitted. This implicit computation could very well depend on reading operands from expected positions; once parsing fails, everything downstream fails.

How could this be improved?

Some modifications that could plausibly help:

- Relative positional encodings could help the model learn "the + sign separates operands" rather than "operand 1 ends at position 2," enabling format generalization.
- Reversed output would reflect natural computation, which may reduce dependence on position-indexed "working memory."
- Variable length training is probably the simplest intervention. It would force format-invariant parsing.