CS310 Progress Report:
Self-proving machine learning models with per-input verification

Howard Cheung
5514611

Supervised by Dr. Matthias C. Caro

January 2026

Contents

1 Introduction **2**
2 Literature review **2**
2.1 Interactive proof systems . . . 2
2.2 Self-proving models . . . 3
2.3 Transcript Learning (TL) . . . 3
2.4 Annotated Transcript Learning (ATL) . . . 4
2.5 Reinforcement Learning from Verifier Feedback (RLVF) . . . 4
3 Reproducing the results of Amit et al. **4**
3.1 Initial training attempts . . . 5
3.2 Hyperparameter analysis . . . 5
3.3 Successful training on DCS compute . . . 7
3.4 Results, inference testing, and distribution analysis . . . 10
3.5 Other reproduction choices . . . 13
3.5.1 Choosing the base of representation . . . 14
3.5.2 Implementing Reinforcement Learning from Verifier Feedback . . . 14
4 Tournament greatest common divisor **14**
4.1 Why tournament GCD? . . . 14
4.2 Ground-truth generator . . . 15
4.3 Linear verifier . . . 15
4.4 Number of inputs . . . 16
4.5 Training data distribution . . . 16
4.6 Annotation strategies for tournament ATL . . . 17
5 Project management **17**
5.1 Reflections on Term 1 . . . 17
5.2 Plans for Term 2 . . . 17
6 References **20**
A Project specification **21**
A.1 Problem statement . . . 21
A.1.1 Motivation . . . 21
A.1.2 Prior work . . . 21
A.1.3 The self-proving framework . . . 22
A.1.4 Gap and project direction . . . 22
A.2 Objectives . . . 22
A.2.1 Theoretical objectives . . . 22
A.2.2 Practical objectives . . . 23
A.3 Methods & methodology . . . 24
A.3.1 Development methodology . . . 24
A.3.2 Data generation . . . 25
A.3.3 Analysis and evaluation . . . 25
A.3.4 Version control and transparency . . . 26
A.4 Timetable . . . 26
A.5 Resources & risks . . . 28
A.5.1 Software . . . 28
A.5.2 Hardware . . . 28
A.6 Legal, social, ethical and professional issues . . . 29
A.7 References . . . 30

1 Introduction

Self-proving models are neural networks trained to output verifiable proofs alongside their answers, transforming average-case correctness guarantees into per-input certificates [Amit et al., 2024]. This project aims to reproduce and extend the self-proving framework to new mathematical problems beyond the two-input GCD case studied in the original paper.

During Term 1, Figure 2 from Amit et al. [2024] was successfully reproduced using the DCS Batch Compute System, with improved training stability (smaller error bars across seeds). A key finding emerged: models trained on log-uniform input distributions fail to generalise to uniformly sampled inputs, despite achieving high verifiability on in-distribution data. This distribution mismatch raises questions about whether verifiability is truly learned or merely memorised for a specific data distribution.

The main extension proposed is *tournament GCD*, which computes $\gcd(x_1, \ldots, x_n)$ for $n > 2$ using a binary tree structure. The ground-truth generator, verifier, and annotation strategies have been designed and are ready for implementation. In Term 2, the distribution question will be investigated, new tournament models will be trained, and further extensions to problems in NP will be explored.

2 Literature review

This section summarises the formal mathematical framework from Amit et al. [2024] on self-proving models, which was studied during the first five weeks of Term 1.

2.1 Interactive proof systems

Self-proving models are based on *Prover-Verifier Games* (PVGs) [Anil et al., 2021]. In a PVG, a computationally limited verifier V interacts with a powerful prover P to check the correctness of P's outputs. The verifier is trusted but cannot solve the problem itself; the prover can solve the problem but is untrusted.

A verifier V must satisfy two guarantees:

- **Completeness**: an honest prover P^* can convince V of any correct answer with probability 1.
- **Soundness**: any prover (even adversarial) cannot convince V of an incorrect answer except with small probability s (the soundness error).

In this work, the verifier is manually defined rather than learned. This limits generality—the "Bitter Lesson" [Sutton, 2019] argues that hand-crafted components tend to plateau while learned approaches scale better in the long term. However, a manually defined verifier is appropriate for tasks with clear correctness criteria, such as mathematical problems. This project therefore explores whether self-proving models can learn verifiable computation for mathematical problems and leaves more general settings for future work.

2.2 Self-proving models

Traditional machine learning optimises for *correctness*: how often does the model output the right answer? For a model F_θ, ground-truth F^*, and input distribution μ:

$$\text{Correctness:} \quad \Pr_{x\sim\mu,\, y\sim F_\theta(x)}[y = F^*(x)] \geq \alpha$$

The distribution μ captures which inputs we care about. For example, in English text, μ would assign higher probability to grammatically correct sentences than to random gibberish. Correctness is about being right on average over inputs weighted by μ, not about every possible input.

This framework assumes each input has exactly one correct output. Tasks like poetry generation, where many outputs are valid, require extensions not covered here. For mathematical problems like GCD, this assumption holds: $\gcd(12, 18) = 6$ and nothing else.

The problem is that α-correctness provides no guarantee for any individual input. A model that is 99% correct on average could still be wrong on the specific input you care about.

Self-proving models instead optimise for *verifiability*: how often can the model prove its answer to the verifier?

$$\text{Verifiability:} \quad \Pr_{x\sim\mu,\, y\sim P_\theta(x)}[\langle V, P_\theta\rangle(x, y) \text{ accepts}] \geq \beta$$

The key insight proved by Amit et al. [2024] is that β-verifiability implies $(\beta - s)$-correctness, i.e. when the verifier accepts, you know the answer is correct with high probability bounded by the soundness error s. This transforms average-case training into per-input guarantees at inference time.

2.3 Transcript Learning (TL)

A *transcript* π is the complete record of a prover-verifier interaction: the model's output y plus all query-response pairs $(q_1, a_1, \ldots, q_R, a_R)$ over R rounds.

The toy problem (in P) used by Amit et al. [2024] is the greatest common divisor (GCD) of two positive integers. The Euclidean algorithm computes $\gcd(a, b)$ by repeatedly replacing the larger number with the remainder of dividing one by the other until the remainder is zero [Euclid, 1908]. A key result from number theory is *Bézout's identity* [Bézout, 1779]:

> For any integers $a, b \in \mathbb{Z}$, not both zero, there exist integers $u, v \in \mathbb{Z}$ such that
>
> $$au + bv = \gcd(a, b).$$

The extended Euclidean algorithm computes both the GCD and these coefficients [Wikipedia contributors].

For GCD, the proof is non-interactive ($R = 1$): the model outputs the answer $y = \gcd(a, b)$ along with Bézout coefficients (u, v), and the verifier checks that $au + bv = y$ and that y divides both a and b. More complex problems may require multiple rounds.

Transcript Learning (TL) trains the model to generate accepting transcripts alongside the outputs. TL requires a dataset of transcripts from an honest prover P^* that the verifier accepts.

The model learns by imitating these "perfect demonstrations", similar to behavioural cloning in reinforcement learning [Pomerleau, 1988].

The training objective is an *agreement function* $A(\theta)$ that measures how often the model's transcript matches the honest one. We cannot optimise verifiability directly because it is discrete (a binary accept/reject signal has no gradient). The agreement function $A(\theta)$ is continuous and differentiable, and Lemma B.4 in Amit et al. [2024] proves that $A(\theta)$ lower-bounds verifiability.[1] Therefore, maximising agreement indirectly maximises verifiability.

2.4 Annotated Transcript Learning (ATL)

Annotated Transcript Learning (ATL) extends TL by including intermediate algorithm steps in the training data. For the GCD problem, the *Euclidean depth* of an input pair (a, b) is the number of iterations needed by the extended Euclidean algorithm. The *annotation cutoff* T controls how many steps are shown during training.

If the model only memorises the training data without learning the underlying algorithm, its verifiability is capped at the fraction of inputs with depth $\leq T$ (the memorisation bound). When verifiability exceeds this bound, it indicates that the model has learned to generalise. Figure 4 in Section 3 shows that the reproduced models exceed their respective memorisation bounds, providing evidence of generalisation.

2.5 Reinforcement Learning from Verifier Feedback (RLVF)

A challenge with training directly from verifier feedback is that verifiers provide only sparse rewards: a single accept/reject bit at the end, with no guidance on intermediate steps. This is the "exploration problem" in RL terms. If a model starts with 0% verifiability, there are no accepted transcripts to learn from.

RLVF addresses this with two-stage training: first use TL to achieve baseline verifiability (around 30–40%), then use rejection sampling on the model's own outputs. As described in a talk by the authors (paraphrased): "If the baseline level is 30%, then we expect 30% accepted transcripts after TL/ATL. Then we can simply continue with RLVF on this." This can push verifiability to 80%+ [Amit et al., 2024]. However, RLVF was not implemented in this project (see Section 3.5).

3 Reproducing the results of Amit et al.

[2]The goal was to reproduce Figure 2 from Amit et al. [2024], which shows verifiability at different Euclidean depths for each annotation level T. Verifiability measures how often the model's output includes a valid proof transcript that the verifier accepts. More precisely, model outputs are evaluated at three levels:

[1]Technically, there are two caveats that need to be handled before differentiating $A(\theta)$: (1) $A(\theta)$ involves an expectation over $x \sim \mu$ and $\pi^* \sim \mathcal{T}^*(x)$, and (2) computing the gradient requires the **log-derivative trick** because $A(\theta)$ involves the probability of an autoregressive model generating a specific sequence.

[2]This was called "Baseline reproduction" in the project specification. To avoid confusion with the *baseline dataset* (where the model outputs only the answer y without the proof transcript π), the term "baseline" is avoided unless referring to that specific dataset.

- **Output**: the GCD k only. The question being answered is "is the final answer correct?"
- **Transcript**: the GCD plus Bézout coefficients (k, u, v). The question being answered is "can the proof be verified?"
- **Annotated**: the intermediate Euclidean steps up to annotation level T. The question being answered is "is the reasoning trace fully correct?"

The "transcript" level corresponds to *verifiability* as defined by Amit et al. [2024].

3.1 Initial training attempts

Following the repository's README, which states

> "Once you obtain data, you can train models on the datasets to reproduce the experimental section of the paper,"

a custom training script was written to train models across all annotation levels on a personal computer (MacBook M2 Pro). However, the paper's appendix and codebase did not clearly specify the exact training hyperparameters upfront. Conservative defaults were therefore chosen:

- Batch size 64 (smaller batches are safer for memory-limited hardware)
- 1 epoch (a quick validation run before committing to longer training)
- Learning rate 4×10^{-4} (a conservative choice)
- Gradient clipping at 1.0 (a standard default)

The results were disappointing: verifiability was approximately 20% lower than the paper across all annotation levels, with no error bars since only one seed was used. Figure 1 shows the results. The dashed lines represent *Euclidean depth bounds* for each annotation level T, which are the theoretical upper limits on verifiability if the model merely memorises rather than generalises. Euclidean depth is the number of iterations required by the Euclidean algorithm for a given input pair. For annotation level T, the model sees at most T steps of the algorithm during training. If the model only memorises these examples without learning the underlying algorithm, it can only prove inputs whose Euclidean depth is at most T. The bound for each T is the fraction of the training distribution with depth $\leq T$.

Critically, annotation levels $T = 5, 6, 7$ all fell *below* their respective Euclidean depth bounds (68.4% vs 75.6%, 72.9% vs 84.9%, and 71.9% vs 91.4%). This indicated that the models were not even memorising effectively, let alone generalising. A fundamental issue with the training configuration was evident.

3.2 Hyperparameter analysis

A closer inspection of the GitHub repository revealed a dedicated reproduction script, `/runs/annot_len.sh`, which contained the exact hyperparameters used by the authors. Table 1 compares the initial settings with those of Amit et al. [2024].

The most significant difference was total data exposure. The number of samples seen during training is:

$$\text{samples seen} = \text{iterations} \times \text{batch size}$$

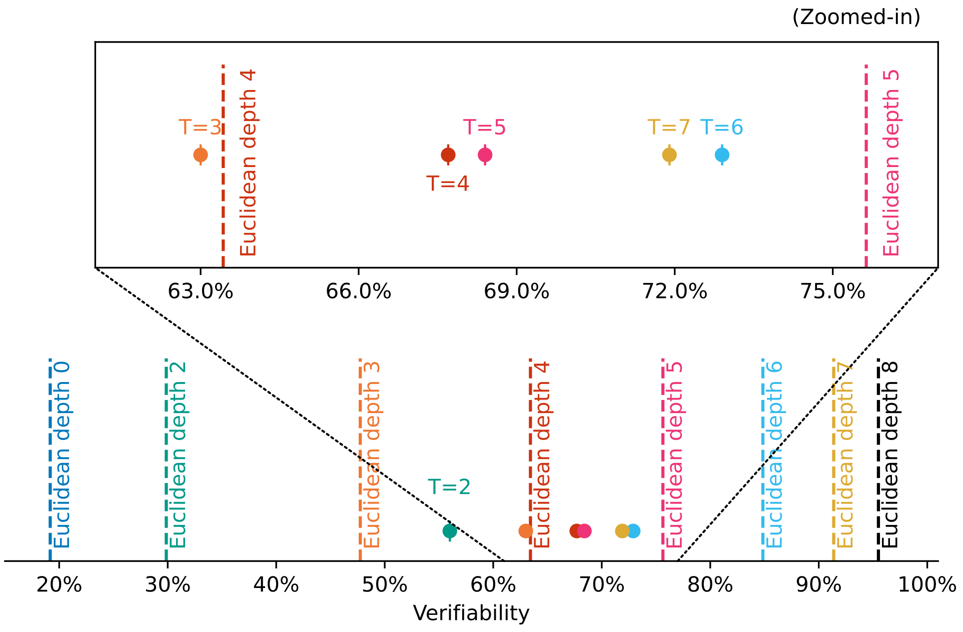


Figure 1: Verifiability from initial local training, with Euclidean depth bounds shown as dashed lines.

Parameter	**Initial**	**Paper**	**Impact on my training**
Batch size	64	1024	16× smaller ⇒ noisier gradients
Epochs	1	10	10× less data exposure
Learning rate	0.0004	0.0007	43% lower ⇒ slower convergence
LR decay	None	10%	No fine-tuning at end
Gradient clip	1.0	2.0	More aggressive clipping
Seeds	1	3	No error bars

Table 1: Comparison of initial training settings versus the paper's reproduction script.

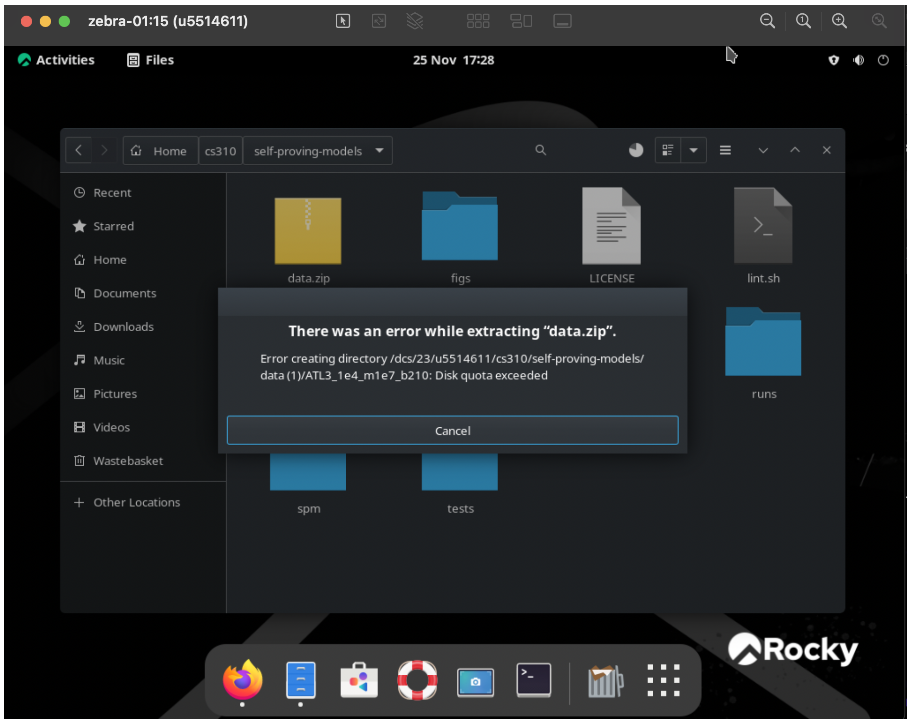
With the initial settings: 160,000 $\times$ 64 $=$ 10,240,000 samples. With the paper's settings: 100,000 $\times$ 1,024 $=$ 102,400,000 samples. This represents a **10× difference** in data exposure, explaining the substantial performance gap. Combined with the lower learning rate and lack of learning rate decay, the model had insufficient training to learn the algorithm properly.

The conclusion was clear: retraining with the correct hyperparameters was necessary.

3.3 Successful training on DCS compute

The retraining required significant computational resources: with 10× more data and 3 seeds per dataset instead of 1, the total would be $10 \times 3 \times \sim 38$[3] $= \sim 1{,}140$ hours (47.5 days) if trained sequentially on the same MacBook. This was infeasible without access to dedicated compute infrastructure.

The DCS Batch Compute System (BCS) provided the necessary resources: Nvidia A10 GPUs with 24GB of VRAM, accessible through the SLURM scheduler. However, the authors' training script `./runs/annot_len.sh` could not be used directly, as it trains all 24 model configurations (8 datasets × 3 seeds) sequentially in a single process, which would exceed the BCS 48-hour time limit per job. Instead, the training was restructured into separate jobs that could be submitted individually and chained using SLURM dependencies. The training dataset (`data.zip`) is 2.9GB compressed but expands to approximately 15GB when extracted. However, with the rest of the project files included, this exceeds the 25GB disk quota for DCS home directories, making it impossible to permanently store the extracted data, as shown in Figure 2.


There was an error while extracting "data.zip".

Error creating directory /dcs/23/u5514611/cs310/self-proving-models/data (1)/ATL3_1e4_m1e7_b210: Disk quota exceeded

Figure 2: Error message when attempting to extract `data.zip` on DCS home directory (25GB quota exceeded)

Following the official DCS dataset guide, my initial approach used `fuse-zip` to mount the compressed ZIP dataset as a virtual filesystem, which avoids extracting it entirely:

[3] Approximately 38 hours were spent on the initial local training.

```
# Mount dataset read-only (no disk quota used)
mkdir $TMPDIR/data
fuse-zip -r data.zip $TMPDIR/data

# Access files normally
ls $TMPDIR/data/Baseline_1e4_m1e7_b210/

# Unmount when done
fusermount3 -u $TMPDIR/data
```

The DCS guide recommends this approach as “100× faster than expanding on network drives” for *typical* file access patterns. Machine learning workloads, however, have very different I/O characteristics, as training involves millions of random I/O operations to load training batches. Each file operation incurs context switching between kernel and userspace.

Consequently, training iterations with `fuse-zip` took approximately 1200ms each—only marginally faster than training on the local machine.

training/logs/old/job_1218535_task_1.err

```
2025-11-26 00:50:06,568: it 0: loss 5.4415, dt 3680.33ms, ep 0.00, lr 7.00e-04
2025-11-26 00:52:06,022: it 100: loss 0.2714, dt 1195.73ms, ep 0.01, lr 7.00e-04
2025-11-26 00:54:05,877: it 200: loss 0.1093, dt 1203.44ms, ep 0.02, lr 7.00e-04
2025-11-26 00:56:05,864: it 300: loss 0.0703, dt 1198.82ms, ep 0.03, lr 7.00e-04
...
slurmstepd: error:
*** JOB 1218535 ON gecko-03 CANCELLED AT 2025-11-26T01:14:53 ***
```

At this rate:

- time per seed: 100,000 iterations $\times$ 1.2s = 33 hours,
- time per dataset (3 seeds): 99 hours,
- all 24 jobs: 24 $\times$ 33 = 792 hours.

The full training campaign would take approximately 33 days of wall-clock time. The job was cancelled after seeing this performance. Despite having access to the GPU cluster, the overhead from `fuse-zip` eliminated any performance gains. The solution (from the same guide) was to extract the dataset to the temporary local storage `$TMPDIR` on each compute node, at the start of each training job:

```
training/train_single.sh
# NEW: Extract data.zip to TMPDIR for faster I/O (avoids FUSE overhead)
echo "Extracting $DATA_ZIP to $MOUNT_POINT..."
unzip -q "$DATA_ZIP" -d "$MOUNT_POINT"

# Verify extraction succeeded
if [ ! -d "$MOUNT_POINT/data" ]; then
    echo "ERROR: Extraction failed - data directory not found"
    exit 1
fi

echo "Extraction successful. Data available at: $MOUNT_POINT/data"
```

This approach is effective because `$TMPDIR` points to local storage on the compute node itself rather than the network-mounted home directory. Consequently, I/O operations could access the native filesystem directly, bypassing all FUSE overhead. Additionally, there are typically 150GB+ available in `$TMPDIR`, so file size is not a constraint. The main downside is that the extracted data is cleaned up when the job finishes, requiring re-extraction for each training run. However, the ∼5-minute extraction time is negligible compared to the hours saved during training. Since data does not need to persist between runs, this trade-off is acceptable. This extraction approach achieved a **35.3× speedup** over `fuse-zip` for the baseline model, which has the shortest sequences (11 tokens). The longer ATL sequences take more time per iteration, as shown in Table 3.

```
training/logs/job_1218541_task_1.err
2025-11-26 01:16:35,700: it 100: loss 0.2034, dt 33.68ms, ep 0.01, lr 7.00e-04
2025-11-26 01:16:39,057: it 200: loss 0.0993, dt 33.91ms, ep 0.02, lr 7.00e-04
2025-11-26 01:16:42,418: it 300: loss 0.0656, dt 33.79ms, ep 0.03, lr 7.00e-04
...
2025-11-26 02:15:07,677: it 99900: loss 0.0018, dt 34.08ms, ep 9.99, lr 7.00e-05
2025-11-26 02:15:12,085: it 100000: loss 0.0016,
                                             dt 1090.60ms, ep 10.00, lr 7.00e-05
```

For reference, Amit et al. [2024] reported ∼45ms per iteration on an NVIDIA A10G GPU. The 34ms achieved on the BCS NVIDIA A10 GPU is slightly faster, which validates that the training setup was working correctly.

Method	**ms/iteration**	**Time per 100k iterations**
`fuse-zip` mounting	∼1200ms	∼33 hours
Extracted to `$TMPDIR`	∼34ms	∼1 hour

Table 2: I/O performance comparison between mounting and extraction approaches

A three-layer (Bash) script hierarchy manages the training runs. The top layer has three submission scripts:

`submit_baseline.sh`, `submit_training_part1.sh`, and `submit_training_part2.sh`. Each one submits jobs to SLURM and chains them with dependencies so they run one after another. These call `train_job.sbatch`, the SLURM batch script that requests 1 GPU (Gecko), 12 CPU threads, and 32GB RAM, and figures out which dataset and seed to train based on the array task ID. The innermost script, `train_single.sh`, performs the core operations: extracting the data, setting the hyperparameters, and running `python train.py`.

`submit_*.sh` → **`sbatch`** `train_job.sbatch` → **`bash`** `train_single.sh` → **`python`** `train.py`

Figure 3: Training script hierarchy

There were 24 training jobs to run (8 datasets, 3 seeds each), but the BCS only allows 16 jobs per array submission. The jobs were therefore split into three batches. First, the baseline dataset (3 jobs) was submitted to verify that the extraction approach functioned correctly on the SLURM system before committing to the rest. Once those finished without errors, Part 1 was submitted: TL, ATL2, ATL3, and ATL4 (12 jobs). Part 2 came last: ATL5, ATL6, and ATL7 (9 jobs). Part 2 contains fewer datasets intentionally, since higher annotation levels lead to longer sequences and slower training; this ensures the wall-clock time is roughly balanced across batches.

Dataset	**Sequence Length**	**ms/iteration**	**Time per seed**
Baseline	11 tokens	∼34ms	∼1 hour
TL	20 tokens	∼40ms	∼1.1 hours
ATL4	56 tokens	∼94ms	∼2.6 hours
ATL7	89 tokens	∼138ms	∼3.8 hours

Table 3: Longer annotations mean longer sequences and slower iterations, from the logs

The whole training campaign took about 58 hours of wall-clock time, running from November 26 to 28 (Term 1 Week 8). Jobs ran sequentially using SLURM's dependency chaining, meaning each job waits for the previous one to finish. This is necessary because running multiple jobs in parallel would have caused the data symlinks to overwrite each other. All logs are available in `training/logs/` for verification of exact timings.

The authors acknowledge the use of the Batch Compute System in the Department of Computer Science at the University of Warwick, and associated support services, in the completion of this work.

3.4 Results, inference testing, and distribution analysis

With training complete, the results as seen in Figure 4 below can be compared against Amit et al. [2024]'s Figure 2. The key metric is *verifiability*: what percentage of the model's outputs come with a proof transcript π that the verifier accepts?

The results closely match those in Figure 2 of Amit et al. [2024]. One limitation is that a direct overlay was not possible because the authors do not publish their raw training logs

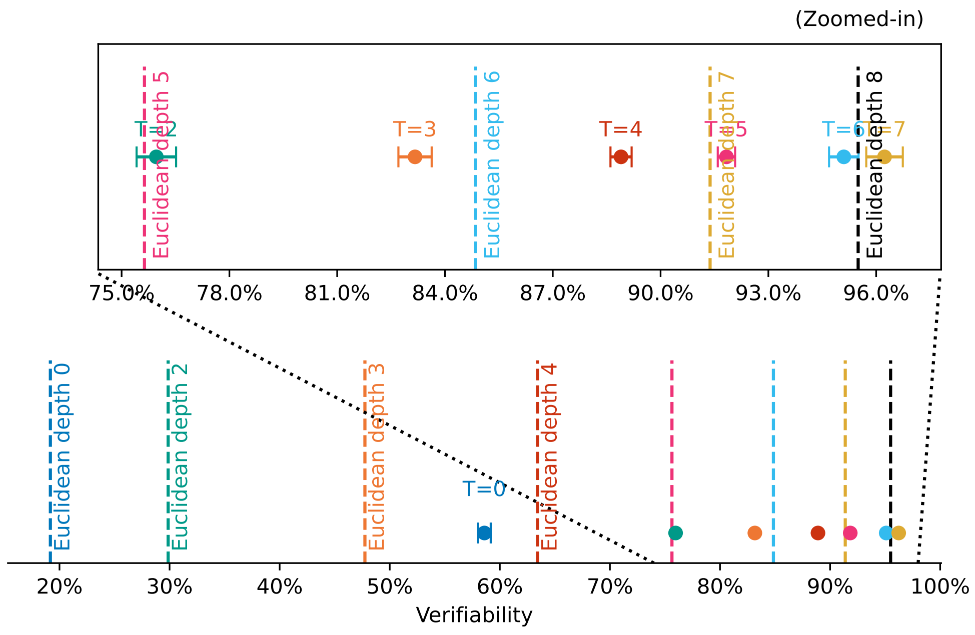


Figure 4: Reproduced verifiability vs Euclidean depth diagram for each annotation level T. Higher T means more intermediate steps are shown during training, leading to higher verifiability. Error bars show standard deviation across 3 seeds.

(the codebase, Section 5.3, and Appendix F of their paper were checked), so this is a visual comparison. One notable difference: the reproduced training is more stable. The error bars (standard deviation across 3 seeds) are noticeably smaller than what the authors show. At $T = 0$ (TL), the reproduced results show 58.6% ± 1% verifiability, while their figure shows a much wider spread. Euclidean depth 2 is also included in the plot, which the authors skipped. Their `annotation.py` script had the depth values hardcoded; this was replaced with a call to `euclidean_depths()` from `samplers.py` so the depths are computed dynamically.

Notably, all annotation levels exceed their respective Euclidean depth bounds, indicating that the models have learned to generalise beyond mere memorisation.

To evaluate the trained models, an inference script was written to test on the held-out evaluation set (`eval.npz`). Table 4 shows the results across 1000 samples from the log-uniform distribution.

The evaluation uses mask-based verification, following the method by Amit et al. [2024]. An earlier approach parsed the output tokens back into integers and checked Bézout's identity mathematically, but parsing errors accumulated and proved difficult to resolve.

The fix was to compare tokens directly against the ground truth using pre-computed masks. For each of the three verification levels (Output, Transcript, Annotated) defined in Section 3, the model's generated tokens are element-wise multiplied by a binary mask that selects only the relevant positions. The result is then compared against the pre-masked ground truth; if all

Model	Output	Transcript	Annotated
Baseline	$99.6 \pm 6.3\%$	N/A	N/A
TL	$98.0 \pm 14.0\%$	$57.0 \pm 49.5\%$	N/A
ATL2	$99.6 \pm 6.3\%$	$76.5 \pm 42.4\%$	$76.0 \pm 42.7\%$
ATL3	$99.7 \pm 5.5\%$	$84.9 \pm 35.8\%$	$84.5 \pm 36.2\%$
ATL4	$99.7 \pm 5.5\%$	$89.0 \pm 31.3\%$	$88.2 \pm 32.3\%$
ATL5	$98.9 \pm 10.4\%$	$91.8 \pm 27.4\%$	$90.8 \pm 28.9\%$
ATL6	$98.9 \pm 10.4\%$	$94.4 \pm 23.0\%$	$93.7 \pm 24.3\%$
ATL7	$99.8 \pm 4.5\%$	$96.9 \pm 17.3\%$	$96.7 \pm 17.9\%$

Table 4: Inference results on held-out evaluation data (log-uniform distribution).

masked positions match exactly, the output is considered correct for that level.

When testing the models interactively with hand-picked inputs like gcd(1024, 2048) or gcd(512, 256), performance dropped significantly to around 20% success versus 80-98% on the evaluation set. The issue stems from the distribution of training data.

spm/data/samplers.py:110-114

```python
def sample_a_b(self, num_samples):
    log_ubound = np.emath.logn(self.base, self.ubound)
    log_a, log_b = np.random.uniform(0, log_ubound, size=(2, num_samples))
    a, b = fcast(np.power(self.base, log_a)), fcast(np.power(self.base, log_b))
    return a, b
```

Both training and evaluation data use **log-uniform sampling** over $[1, 10^4]$: numbers are drawn as round($10^{U(0,4)}$). Figure 5 illustrates the difference between log-uniform and uniform sampling.

The log-uniform distribution heavily favours smaller values, meaning numbers below 100 appear far more often than numbers above 1000. The handpicked test examples (clean powers of 2, bigger numbers) were out of distribution. Although Figure 4 shows that the model learned to solve GCDs, it does so only for the specific distribution it trained on, not for GCDs in general.

To quantify this distribution mismatch, a new evaluation dataset with **uniform sampling** over $[1, 10^4]$ was generated and the same trained models were tested. Table 5 shows the results.

The results reveal a significant performance gap. While output accuracy remains high (92-99%), verifiability (transcript accuracy) drops dramatically: from 57% to 2% for TL, from 97% to 63% for ATL7, and similarly for other annotation levels. The models learned to compute the GCD correctly, but their ability to produce valid proofs **does not generalise beyond the log-uniform distribution** they were trained on, a phenomenon similar to the issues identified in the analysis of behavioural cloning by Ross et al. [2011].

This raises a question for the tournament extension in the next section: should the model be

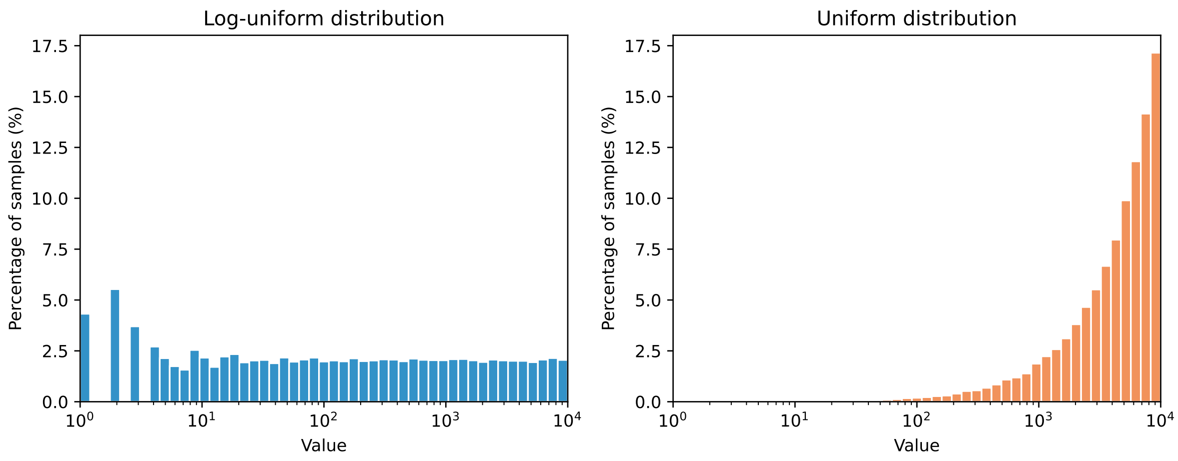


Figure 5: Comparison of log-uniform (left) and uniform (right) sampling over $[1, 10^4]$ on a logarithmic x-axis. **In log-space,** log-uniform sampling produces a flat distribution, while uniform sampling concentrates samples at higher values.

Model	Output	Transcript	Annotated
Baseline	$98.6 \pm 11.7\%$	N/A	N/A
TL	$92.5 \pm 26.3\%$	$2.0 \pm 14.0\%$	N/A
ATL2	$98.6 \pm 11.7\%$	$5.9 \pm 23.6\%$	$5.7 \pm 23.2\%$
ATL3	$98.5 \pm 12.2\%$	$10.2 \pm 30.3\%$	$9.8 \pm 29.7\%$
ATL4	$97.4 \pm 15.9\%$	$14.6 \pm 35.3\%$	$14.1 \pm 34.8\%$
ATL5	$95.2 \pm 21.4\%$	$31.5 \pm 46.5\%$	$29.6 \pm 45.6\%$
ATL6	$92.2 \pm 26.8\%$	$42.0 \pm 49.4\%$	$39.0 \pm 48.8\%$
ATL7	$97.8 \pm 14.7\%$	$62.9 \pm 48.3\%$	$59.9 \pm 49.0\%$

Table 5: Inference results on uniformly sampled evaluation data. The same models from Table 4 were tested on 1000 uniformly sampled pairs from $[1, 10^4]$.

trained on log-uniform data (for a fair comparison with the reproduced results, i.e., the case where $n = 2$) or on uniform data (to test whether the architecture can learn more general arithmetic)? This will be investigated in Term 2.

3.5 Other reproduction choices

Two experiments from the original paper were not reproduced: the base of representation ablation (Figure 3) and Reinforcement Learning from Verifier Feedback (RLVF).

3.5.1 Choosing the base of representation

Figure 3 in Amit et al. [2024] tests how the number of unique prime divisors $\omega(B)$ in the base of representation B affects verifiability. Charton [2023] had previously shown that transformers correctly compute GCDs when the integers are products of primes dividing the base; Amit et al. [2024] found that this also holds for verifiability.

Reproducing this experiment would require generating many datasets (one per base configuration) and training with 20 seeds each, as specified in the repository's README. After discussing with my supervisor, the decision was to prioritise the annotation length experiments instead. The annotation results (Figure 2) are more relevant to the tournament GCD extension, which will use annotated transcripts. Investigating tokenisation effects is not directly relevant to the project's goal of extending Self-Proving models to other mathematical problems.

3.5.2 Implementing Reinforcement Learning from Verifier Feedback

RLVF was not implemented, as Amit et al. [2024] reported (Appendix F) that their RLVF training took approximately one month. As a rough estimate, the NVIDIA A10 GPU used in the BCS showed performance similar to that reported by Amit et al. [2024] for the reproduction process detailed above, so training using RLVF was not feasible. In addition, the BCS cluster imposes a 48-hour time limit per job, and RLVF's continuous workflow, which involves online generation and rejection sampling, does not fit a batch job model. Transcript Learning and Annotated Transcript Learning are sufficient for reproducing the paper's main results and for the tournament extension.

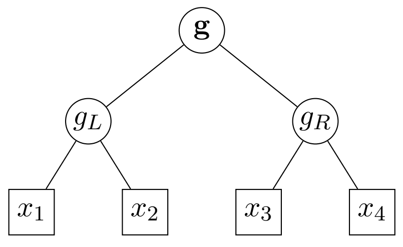
4 Tournament greatest common divisor

The results reproduced in Section 3 address the two-input case ($n = 2$). A natural question arises: can self-proving models scale to computing $\gcd(x_1, x_2, \ldots, x_n)$ for $n > 2$? This section describes design decisions for extending the approach using a **tournament structure**. Extending to tournament GCD requires three components: (1) a ground-truth generator that produces training data with the full reasoning trace, (2) a verifier that checks the model's output, and (3) a model (prover) trained to generate the complete execution trace.

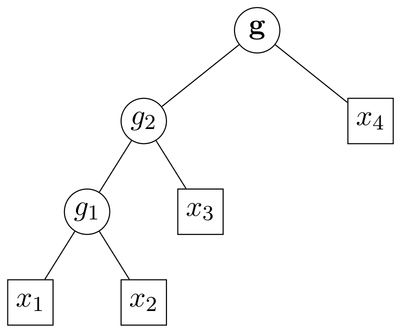
4.1 Why tournament GCD?

GCD is *associative*: $\gcd(\gcd(a, b), c) = \gcd(a, \gcd(b, c))$ [Wikipedia contributors], meaning the GCD of n numbers can be computed by combining pairwise GCDs in any order, and the result is the same. Figure 6 illustrates two approaches for computing $\gcd(x_1, x_2, x_3, x_4)$.

Both approaches require $n - 1$ pairwise GCD operations. The difference is the *depth*: the longest chain of dependent operations. Sequential computation has depth $n - 1$, while tournament structure has depth $\lceil \log_2 n \rceil$. This matters because transformers struggle with long-range dependencies; shorter dependency chains are easier to learn [Vaswani et al., 2017][Merrill et al., 2025].



Tournament: depth $\log_2 n$



Sequential: depth $n-1$

Figure 6: Tournament (left) vs sequential (right) computation of $\mathbf{g} = \gcd(x_1, x_2, x_3, x_4)$. Both perform $n-1 = 3$ pairwise GCDs, but the tournament has depth $\log_2 4 = 2$ while the sequential chain has depth 3.

4.2 Ground-truth generator

For the tournament to produce a valid proof, the model must output n Bézout coefficients $(c_1, \ldots, c_n)$ satisfying

$$\sum_{i=1}^{n} c_i \cdot x_i = \mathbf{g}$$

where $\mathbf{g} = \gcd(x_1, \ldots, x_n)$. These coefficients are computed by combining local Bézout coefficients at each level of the tournament. For example, with $n = 4$:

- Compute $g_L = \gcd(x_1, x_2) = u_1 x_1 + v_1 x_2$ and $g_R = \gcd(x_3, x_4) = u_2 x_3 + v_2 x_4$
- Compute $\mathbf{g} = \gcd(g_L, g_R) = A \cdot g_L + B \cdot g_R$
- Substituting: $\mathbf{g} = A(u_1 x_1 + v_1 x_2) + B(u_2 x_3 + v_2 x_4) = (Au_1)x_1 + (Av_1)x_2 + (Bu_2)x_3 + (Bv_2)x_4$

By induction, this extends to any n, requiring $\lceil \log_2 n \rceil$ rounds and $n-1$ pairwise GCDs. Figure 7 shows the algorithm in pseudocode.

4.3 Linear verifier

The verifier performs two checks, both in $O(n)$ time:

Check 1 (Bézout's identity): Does $\sum_{i=1}^{n} c_i \cdot x_i = y$? This confirms y is a linear combination of the inputs, so the true GCD G must divide y, implying $G \leq y$.

Check 2 (common divisor): Does $y \mid x_i \ \forall i \in [n]$? This confirms y is a common divisor. Combined with Check 1, we have $G \leq y$ and $y \leq G$, proving $y = G$.

For annotated transcripts, the verifier also checks that intermediate values match the ground truth using the same mask-based approach from Section 3.4.

```
def tournament_gcd(inputs):
    """
    Inputs: A list of integers inputs = [x_1, ..., x_n]
    Outputs:
       - final_gcd: The GCD of all inputs
       - final_coeffs: A list of n integers [c_1, ..., c_n]
         such that sum(c_i * x_i) = final_gcd
    """
    n = len(inputs)
    # Each element tracks (value, coefficient_vector)
    layer = [(x, one_hot(i, n)) for i, x in enumerate(inputs)]

    while len(layer) > 1:
        next_layer = []
        for i in range(0, len(layer), 2):
            if i + 1 >= len(layer):
                next_layer.append(layer[i]) # Odd element passes through
            else:
                left, right = layer[i], layer[i+1]
                g, u, v = extended_gcd(left.value, right.value)
                new_coeffs = u * left.coeffs + v * right.coeffs
                next_layer.append((g, new_coeffs))
        layer = next_layer

    return layer[0] # (final_gcd, [c_1, ..., c_n])
```

Figure 7: Tournament GCD algorithm to generate the ground-truths (pseudocode). The coefficient vectors are combined at each level.

4.4 Number of inputs

The tournament extension uses fixed-n with powers of 2: specifically $n = 4$, 8, and 16. Powers of 2 produce clean binary tournament trees with no odd elements to carry forward between rounds. Fixed-n also keeps the transformer's block size constant and enables a "verifiability vs tournament size" plot analogous to Figure 4.

Variable-n is left for later work. Supporting arbitrary n would require either multiple rounds of interaction ($R > 1$ in the interactive proof framework from Section 2) or a model-picker architecture that selects the appropriate fixed-n model, which raises meta-learning questions.

4.5 Training data distribution

The choice of input distribution remains an open question for Term 2. With log-uniform sampling, as n increases, the GCD of n random numbers quickly converges to 1 (since most pairs of random integers are coprime) [Nymann, 1972]. Two options are being considered: (1) log-uniform to enable fair comparison with the reproduced $n = 2$ results, or (2) uniform to test whether the architecture can learn more general arithmetic[4].

[4]This will be investigated before generating the tournament datasets, to be done immediately after this document is submitted.

4.6 Annotation strategies for tournament ATL

Three annotation strategies are under consideration:

Option A: Show only pairwise GCD results at each round, without Euclidean sub-steps. This tests pure tournament reasoning by isolating the tournament structure from the within-pair computation.

Option B: Show Euclidean algorithm steps within each pairwise GCD. This provides more supervision but produces longer sequences and more complex masking.

Option C: Use pre-trained ATL models recursively, where each pairwise GCD is handled by an already-trained model. This is left as exploratory future work, as it raises the question of whether we are simply shifting the problem to meta-learning.

The plan is to start with Option A to establish baseline tournament performance, then test Option B to measure whether additional supervision improves verifiability.

5 Project management

5.1 Reflections on Term 1

The original timetable in the **Project Specification** was too ambitious for three reasons. **First,** the specification was written in Week 2 of Term 1 when the workload from other modules was still light; it was not anticipated how much time coursework deadlines would consume later in the term:

- CS352 Project Management: 2 essays, 2 peer reviews, 1 group presentation
- CS342 Machine Learning: programming coursework due Tuesday Week 11 (16 December)
- CS325 Compiler Design: programming coursework due Monday Week 8 (24 November)

Second, the timeline was intentionally aggressive to see how much progress could be made. **Third,** several graduate school applications needed to be completed during the term, which took significant time and energy.

Project management could have been better. The **Project Specification** timetable allocated Week 10 for cleaning up the repository and preparing the progress report write-up, but this could not be done due to coursework load, and an unforeseen family situation during the Christmas break meant no progress could be made over the holiday. Additionally, because training required the DCS Batch Compute System while development happened locally on my MacBook, two separate workspaces were effectively maintained. This made the codebase too messy to commit to GitHub regularly.

5.2 Plans for Term 2

The immediate next steps are (in this order):

① Consolidate the two codebases into a single, clean repository. With the 100GB storage allocation recently received from DCS, there is no longer a need to work around disk quotas with `.gitignore`'s, SSH file transfers, `$TMPDIR`, or Google Drive. Regular commits and pushes to the GitHub repository will maintain a clear development trace.

② Investigate the distribution question described in Section 4.

③ Implement and train fixed-n tournament GCD models for $n = 4, 8, 16$.

④ Explore further extensions: either variable-n tournament GCD, or a new mathematical problem, ideally NP-complete, to demonstrate the generality of self-proving models. For NP-complete problems, verification is efficient while solution-finding is hard, so the ground-truth generator would produce solutions that the model learns to construct with proofs. *This is exploratory work.*

An online *Kanban* was initially considered to make the workflow more transparent (an idea from a friend), but his project involved building an application with parallel workstreams. After discussion with the supervisor, it was agreed that a simple to-do list and fortnightly meetings would suffice for this sequential, empirical research project.

The general principle for Term 2 is to front-load work in the first half, leaving time for report writing and other module deadlines: CS331 Neural Computing has a coursework due in the second half of term (date TBA), and CS356 Approximation and Randomised Algorithms has a deadline on Monday of Week 7 (23 February). The exact plan is detailed in the timetable below.

Time	**Focus/work**	**Official deadline**
* * * * *	**Start of Term 2**	* * * * *
Week 1 Jan 12 – Jan 16	Discuss progress and next steps with supervisor. Prepare and review the first draft of the **Progress Report**.	—
Week 2 Jan 19 – Jan 23	Finalise and submit this document (**Progress Report**).	**Progress Report** Thursday, Jan 22
Week 3 Jan 26 – Jan 30	Investigate the distribution question (log-uniform vs uniform training). Consolidate and upload the clean repository to GitHub.	—
Week 4 Feb 2 – Feb 6	Implement and train fixed-n tournament GCD models for $n = 4, 8, 16$: generate datasets, implement ground-truth generator and verifier, train models, and evaluate verifiability.	—
Week 5 Feb 9 – Feb 13		—
Week 6 Feb 16 – Feb 20		—
Week 7 Feb 23 – Feb 27	Explore further extensions: variable-n tournament GCD or a new mathematical problem in NP.	—
Week 8 Mar 2 – Mar 6	Consolidate all experimental results. Finalise code and documentation. Prepare key findings and visualisations for the **Final Report**.	—

Time	**Focus/work**	**Official deadline**
Week 9 Mar 9 – Mar 13	Make the *Final Report Plan*, outlining the structure, key arguments, and results to be included.	*Final Report Plan* Friday, Mar 13
Week 10 Mar 16 – Mar 20	Meet with supervisor to review the *Final Report Plan* and discuss the scope of the **Project Management Report** before the Easter break.	—
∗ ∗ ∗ ∗ ∗ **End of Term 2** ∗ ∗ ∗ ∗ ∗		
Easter break Mar 21 – Apr 26	Finish the write-up of the **Project Management Report** and the **Final Report**.	**PM Report** & **Final Report** Thursday, Apr 9
∗ ∗ ∗ ∗ ∗ **Start of Term 3** ∗ ∗ ∗ ∗ ∗		
Early weeks Apr 27 onwards Week 2 – 3	Prepare for the **Viva**: create presentation slides, prepare demo, and review all project work and key literature for the Q&A session.	**Viva** Monday, May 4 – Friday, May 15

6 References

Amit, Noga, Shafi Goldwasser, Orr Paradise, and Guy N. Rothblum [2024]. “Models That Prove Their Own Correctness”. In: *ICML 2024 Workshop on Theoretical Foundations of Foundation Models*. URL: https://openreview.net/forum?id=yCm00lPmBo.

Anil, Cem, Guodong Zhang, Yuhuai Wu, and Roger Grosse [2021]. *Learning to Give Checkable Answers with Prover-Verifier Games*. arXiv: 2108.12099 [cs.LG]. URL: https://arxiv.org/abs/2108.12099.

Sutton, Richard S. [2019]. *The Bitter Lesson*. Online essay, Incomplete Ideas. Accessed from http://www.incompleteideas.net/IncIdeas/BitterLesson.html.

Euclid [1908]. *The Thirteen Books of the Elements. Books III–IX*. Ed. by Thomas L. Heath. Vol. 2. Originally written c. 300 BC; English translation and commentary by T. L. Heath. Book VII contains the classical Euclidean algorithm for the greatest common divisor. Cambridge: Cambridge University Press.

Bézout, Étienne [1779]. *Théorie générale des équations algébriques*. Paris, France: Ph.–D. Pierres.

Wikipedia contributors [n.d.(a)]. *Extended Euclidean algorithm*. https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm. Accessed: 21 January 2026.

Pomerleau, Dean A. [1988]. “ALVINN: An Autonomous Land Vehicle in a Neural Network”. In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 1. Morgan Kaufmann, pp. 305–313.

Ross, Stéphane, Geoffrey Gordon, and Drew Bagnell [2011]. “A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning”. In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS)*, pp. 627–635.

Charton, François [2023]. “Learning the greatest common divisor: explaining transformer predictions”. In: *arXiv preprint arXiv:2308.15594v2*. Revised March 14, 2024.

Wikipedia contributors [n.d.(b)]. *Greatest common divisor*. Accessed: 21 January 2026. URL: https://en.wikipedia.org/wiki/Greatest_common_divisor#:~:text=The%5C%20GCD%5C%20is%5C%20an%5C%20associative%5C%20function.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin [2017]. “Attention is all you need”. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. NIPS’17. Long Beach, California, USA: Curran Associates Inc., pp. 6000–6010. ISBN: 9781510860964.

Merrill, William and Ashish Sabharwal [2025]. *A Little Depth Goes a Long Way: The Expressive Power of Log-Depth Transformers*. arXiv: 2503.03961 [cs.LG]. URL: https://arxiv.org/abs/2503.03961.

Nymann, J. E. [1972]. “On the probability that n positive integers are relatively prime”. In: *Journal of Number Theory* 4.5, pp. 469–473. DOI: 10.1016/0022-314X(72)90038-8.

A Project specification

As required by the module webpage, a copy of the original **Project Specification** has been included as an appendix to this **Progress Report**.

CS310 Project Specification:

Self-proving machine learning models with per-input verification

Howard Cheung
5514611

Supervised by Dr. Matthias C. Caro

October 2025

A.1 Problem statement

A.1.1 Motivation

Modern machine learning models and Large Language Models (LLMs) achieve outstanding *average* accuracy across benchmarks [Artificial Analysis, 2025], providing empirical evidence that they are *usually* right. However, they offer no formal guarantee that any *particular* output is correct for a specific input, due to the risk of subtle errors or "hallucinations" [Huang et al., 2025]. This poses a serious reliability gap in high-stakes domains such as healthcare, finance, mathematics, and law, where even a single incorrect answer can have severe consequences.

In this work we ask the following question:

Can a model provide not only an answer but also a formal proof that the answer is correct for the specific input at hand?

If we can solve this, we move from "usually right" to "provably right". We no longer have to bet on probabilities when we cannot afford to take any chances, and trustworthy AI could be deployed in more critical fields.

A.1.2 Prior work

Many strategies have been proposed to verify the reasoning of LLMs. One approach is to prompt models to explain their reasoning in natural language; however, the model may instead attempt to persuade the user of an incorrect answer [Turpin et al., 2023]. Another is to ask the LLM to defend its responses, but even when the answer is correct, it may still fail to produce a logically convincing proof [Wang et al., 2023]. Using chain-of-thought (CoT) prompting to reveal intermediate reasoning steps [Wei et al., 2022] can improve interpretability and sometimes accuracy, as demonstrated in many mainstream reasoning LLMs, such as `o1` [OpenAI, 2024], `Claude Sonnet 4.5` [Anthropic, 2025], and `Gemini 2.5 Pro` [Google DeepMind, 2025]. Even so, subsequent studies have shown that CoT reasoning *itself* can be fabricated or internally inconsistent, producing justifications that sound plausible but are not genuinely correct [Cheng et al., 2025]. Ultimately, these approaches may improve user trust or interpretability, but none can formally prove correctness *on a per-input basis*.

A.1.3 The self-proving framework

A recent framework proposed by Amit et al. [2024] introduces the concept of **self-proving models** to address this lack of per-input verification using interactive proofs [Goldwasser et al., 1985]. Instead of merely generating an output y from an input x, a self-proving model P_θ (the prover) produces an interactive proof transcript π alongside the output y through a series of queries and responses with a fixed verifier V; the transcript π is then checked by V, which accepts only if it is convinced that the answer y is correct and rejects otherwise[5].

A.1.4 Gap and project direction

However, the study by Amit et al. [2024] primarily served as a *proof of concept*, focusing on a simple arithmetic task—the greatest common divisor (GCD) problem using Bézout's identity—and left open the broader question of how these ideas might generalise to more complex reasoning tasks.

The notion of per-input verification could, in principle, be extended to language-based tasks like legal or policy summarisation, where both correctness and comprehensive coverage matter. However, unlike mathematics, correctness in such domains is often subjective and context-dependent, making formal verification far less straightforward. If mathematical problems can be generalised robustly, this line of work could eventually inspire verification methods for more ambiguous settings, but that remains an extension beyond the scope of this project.

Our project therefore begins with **mathematical** domains where correctness can be defined unambiguously. In particular, we apply the self-proving framework to areas of mathematics beyond number theory, providing both provable guarantees and implementations of self-proving models for at least one new mathematical problem. Moreover, this opens up some interesting possibilities for evaluation of the results, which will be elaborated in Section A.3.3.

A.2 Objectives

The overarching aim is to extend the self-proving models framework beyond its original toy setting (GCD) and explore whether such per-input verification can hold for harder mathematical problems. The objectives can be divided into two categories:

- theoretical: developing the mathematical foundations and formal understanding of the framework;
- practical: implementing the theory in code and producing a functional demonstration.

A.2.1 Theoretical objectives

① Familiarisation with literature and foundation building.

[5]More precisely, any incorrect y passes only with negligible probability.

- **Description:** Develop a thorough understanding of the mathematical and formal knowledge of the self-proving framework introduced by Amit et al. [2024]. This includes studying the theory of *interactive proofs*, formal definitions of *verifiers* and *provers*, and the three training methods—*Transcript Learning* (TL), *Reinforcement Learning from Verifier Feedback* (RLVF), and *Annotated Transcript Learning* (ATL). Supplementary resources like the talk on YouTube will be utilised.
- **Deliverable:** Study notes (typed in `Markdown` using Obsidian) summarising key definitions, intuition behind the theory, and relationships between core concepts. These notes will serve both as reference material for supervisor meetings and as a foundation for the Literature Review in the final report.

(2) Extension to a more complex mathematical task.

- **Description:** Select and define a new mathematical problem to apply the self-proving framework beyond the GCD example. One potential candidate is *Carathéodory's Theorem* for convex set containment, which states that any point inside a convex hull in $\mathbb{R}^d$ can be expressed as a convex combination of at most $d+1$ points. This property naturally lends itself to a prover-verifier setup, where the prover must supply such a combination and the verifier checks its validity efficiently. The goal is to determine how correctness and verifiability can be clearly defined for this problem while keeping it simple enough to implement in the later demo.
- **Deliverable:** A short write-up summarising the chosen problem, its formal definition, and an outline of how a prover-verifier setup could be built for it. This will be discussed with the supervisor to assess feasibility and guide the practical stage.

(3) Exploratory work.

- **Description:** Reflect on insights gained from applying the framework to the new mathematical problem and explore whether similar principles could extend to other domains like combinatorics (e.g. graph-colourability or connectivity proofs), or linear algebra (e.g. matrix inversion verification). These would be pursued only in a best-case scenario as exploratory directions once the primary mathematical task has been completed and they involve identifying structural patterns that make a task suitable for self-proving models. If time permits, connections to quantum-verifiable models could be explored with the supervisor, but this is a stretch goal that will only happen if the main objectives are completed smoothly ahead of schedule.
- **Deliverable:** To be defined in consultation with the supervisor. Possible outcomes may include a short conceptual note or meeting summary that outlines promising directions to be refined into follow-up objectives on new problems or a generalisation.

A.2.2 Practical objectives

(1) Reproduction of the baseline system.

- **Description:** Fork the official repository by Amit et al. [2024] to set up a local development environment. Reproduce the original GCD-based experiment using the same framework built using nanoGPT. The goal is to understand the code structure, verify that the results match those reported in the paper, and gain practical experience with the prover-verifier interaction, implementation of proof generation, training configuration, and evaluation metrics. Once the baseline runs smoothly, I plan to experiment with scaling the setup to larger open-source/weight models (such as GPT-OSS) to see how the framework behaves at a bigger scale.

- **Deliverable:**
  - A forked and functioning GitHub repository successfully replicating the GCD experiment.
  - Notes summarising the repository structure, key functions (proof generation and verification), system behaviour, and training results, along with thoughts on extending the setup to larger models.

② Implementation of the new problem.

- **Description:** Implement and train self-proving models for the new mathematical problem defined earlier. This will include programming a verifier algorithm, preparing the datasets, and applying the training methods described by Amit et al. [2024]: TL, RLVF, and ATL.
- **Deliverable:** A functional implementation of the self-proving framework applied to the chosen mathematical problem in a GitHub repository, with clear documentation explaining the verifier design, dataset setup, and training outcomes, which will serve as the basis for evaluation and analysis in the next stage of the project.

③ Evaluation and analysis.

- **Description:** Evaluate the implemented system by comparing correctness and verifiability between the baseline (GCD) and the new mathematical task quantitatively. Analyse failure cases, model behaviour, and scalability to assess the strengths and limitations of the framework.
- **Deliverable:** A document summarising experimental results, key metrics, and insights into model performance and generalisability, which will be included in the progress and final report.

As this is a new and fast-moving area of research, the precise scope and direction are expected to develop over time. The objectives above mark the early and tangible milestones.

A.3 Methods & methodology

A.3.1 Development methodology

This project will adopt an iterative workflow. Each iteration has three stages:

1. Define or refine a theoretical component (e.g. verifier design or training objective).
2. Implement and run experiments to test it.
3. Analyse outcomes and update the plan based on results.

Findings and milestones will be reviewed during fortnightly supervision meetings, so subsequent work builds on validated results and insights.

As mentioned, the project will start by reproducing the baseline GCD experiment from Amit et al. [2024] using their repository. If the fork integrates cleanly, further work will build on it; otherwise (e.g. if there are dependency issues, limited modularity, or incompatibility with any extensions), a standalone implementation will be developed.

A.3.2 Data generation

The project will generate all training data programmatically rather than relying on external datasets. Each sample represents an interaction between a *prover* (the model) and a *verifier* (a deterministic checking algorithm).

For a given input x, the ground-truth output $F^*(x)$ is computed using a trusted algorithm, and a valid proof transcript π is produced by the model. The resulting triplets (x, y, π) form the training data for TL. In ATL, intermediate reasoning steps are inserted as annotations within π. In RLVF, data are generated online as the model interacts directly with the verifier and receives binary accept/reject feedback.

All datasets will be created with fixed random seeds and clear documentation of the generation parameters, so experiments can be reproduced.

A.3.3 Analysis and evaluation

The evaluation strategy largely follows the framework introduced by Amit et al. [2024], with additional analyses specific to this project. Model performance will be assessed quantitatively along two core dimensions:

- **Correctness:** the proportion of model outputs y matching the ground-truth $F^*(x)$.
- **Verifiability:** the proportion of outputs whose proofs π are accepted by the verifier V.

Results will be reported as averages over multiple seeds with standard deviations and plotted as learning curves showing correctness and verifiability over training iterations.

In addition to the standard evaluation, three complementary analyses will be performed to gain a deeper understanding of model behaviour:

- **Sample efficiency:** how quickly each training method improves verifiability with increasing data or training steps, indicating how efficiently the model learns from verifier feedback.
- **Proof-level statistics:** examination of typical proof length, structure, and interaction complexity (e.g. number of rounds), which are expected to vary depending on the mathematical task and verifier design. This helps identify how the complexity of the task influences the difficulty of achieving high verifiability percentage.
- **Cross-task comparison:** comparison between the baseline GCD experiment and the new mathematical task to evaluate whether the three training methods remain effective as proof structures and verifier complexity change, and how well the self-proving framework generalises to different problem structures.

A.3.4 Version control and transparency

All code, notes, experiment configurations, and results (including metrics, visualisations, and summaries) will be version-controlled using `Git` and hosted on a GitHub repository accessible to the supervisor, with branches representing major iterations.

A.4 Timetable

Time	Focus/work	Official deadline
* * * * *	**Start of Term 1**	* * * * *
Week 1 Oct 6 – Oct 10	Project initiation; kick-off meeting with the supervisor.	—
Week 2 Oct 13 – Oct 17	Write-up and submission of this document (**Project Specification**).	**Project Spec.** Thursday, Oct 16
Week 3 Oct 20 – Oct 24	Literature review on interactive proofs, provers/verifiers, TL/RLVF/ATL; produce `markdown` notes.	—
Week 4 Oct 27 – Oct 31	Reproduce GCD baseline experiment from Amit et al. [2024]. Fork the repo, set up the development environment, install dependencies, and study the code structure, focusing on data generation and verifier implementation.	—
Week 5 Nov 3 – Nov 7	Complete the baseline reproduction; execute the full training and evaluation scripts, debugging any runtime issues. Check metrics match paper; prepare short internal summary.	—
Week 6 Nov 10 – Nov 14	Choose a mathematical problem suitable for self-proving setup; formalise its definition and correctness condition[6]. Draft pseudocode for both prover's expected output and verifier's checking algorithm.	—
Week 7 Nov 17 – Nov 21	Implement the verifier algorithm and dataset generation scripts for the new mathematical task.	—
Week 8 Nov 24 – Nov 28	Train the self-proving model on the new mathematical task using TL, ATL, and RLVF.	—

[6]just like Bézout's identity for the GCD problem

Time	Focus/work	Official deadline
Week 9 Dec 1 – Dec 5	Evaluate the model's performance on the new task; measure correctness and verifiability, and analyse initial results using the three metrics from Section A.3.3.	—
Week 10 Dec 8 – Dec 12	Clean repo, summarise results, and prepare short internal write-up for **Progress Report**.	—
* * * * *	**End of Term 1**	* * * * *
Christmas break Dec 13 – Jan 11	Buffer time; start first draft of the **Progress Report**.	—
* * * * *	**Start of Term 2**	* * * * *
Week 1 Jan 12 – Jan 16	Discuss progress and next steps with supervisor. Prepare and review the first draft of the **Progress Report**.	—
Week 2 Jan 19 – Jan 23	Finalise and submit the **Progress Report**.	**Progress Report** Thursday, Jan 22
Week 3 Jan 26 – Jan 30	*Exploratory work phase.* Possible directions include: (1) extending the framework to additional mathematical problems identified in Term 1; (2) generalising findings and developing theoretical insights about which classes of mathematical problems are naturally suited for self-proving models. Specific objectives will be defined based on Term 1 findings and **Progress Report** discussions.	—
Week 4 Feb 2 – Feb 6		—
Week 5 Feb 9 – Feb 13		—
Week 6 Feb 16 – Feb 20		—
Week 7 Feb 23 – Feb 27		—
Week 8 Mar 2 – Mar 6	Consolidate all experimental (and/or theoretical) results. Finalise code, make sure it is well-documented and the repo is clean. Prepare a summary of key findings and visualisations for the **Final Report**.	—
Week 9 Mar 9 – Mar 13	Make the *Final Report Plan*[7], outlining the structure, key arguments, and results to be included.	*Final Report Plan* Friday, Mar 13

[7]This *might* be optional, but it is recommended for additional feedback.

Time	Focus/work	Official deadline
Week 10 Mar 16 – Mar 20	Meet with supervisor to review the *Final Report Plan* and discuss the scope of the **Project Management Report** before the Easter break.	—
∗ ∗ ∗ ∗ ∗ **End of Term 2** ∗ ∗ ∗ ∗ ∗		
Easter break Mar 21 – Apr 26	Finish the write-up of the **Project Management Report** and the **Final Report**.	**PM Report** & **Final Report** Thursday, Apr 9
∗ ∗ ∗ ∗ ∗ **Start of Term 3** ∗ ∗ ∗ ∗ ∗		
Early weeks Apr 27 onwards Week 2 – 3	Prepare for the **Viva**: create presentation slides, prepare demo, and review all project work and key literature for the Q&A session.	**Viva** Monday, May 4 – *Friday, May 14*[8]

A.5 Resources & risks

A.5.1 Software

The software stack will consist of free tools.

- The programming language of choice is `Python` for its readability, extensive machine-learning ecosystem, and native support for GPU acceleration via PyTorch.
- Version control will be managed using `Git` and GitHub.
- Obsidian will be used for maintaining notes and meeting summaries.
- Overleaf/LaTeX will be used for preparing official reports.
- Development will be carried out in VSCode.

All datasets are generated programmatically, so no external data is required. Key references, including Amit et al. [2024] and supporting literature on interactive proofs and verifier-feedback training, are downloaded for local access.

A.5.2 Hardware

The project requires computation for training and verifying small-to-medium-scale ML models.

Development and light experimentation will be conducted on a MacBook Pro (M2 Pro, 32GB RAM, 1TB storage), which provides sufficient performance for small-scale prototyping.

For heavier computation, the project will make use of the DCS compute infrastructure, which includes both Compute Nodes and the Batch Compute System:

[8] Exact date TBC. At the time of writing, the website says `Friday 14th May 2026`, but 14th May 2026 is a Thursday. It is therefore unclear whether the intended date is *Thursday 14th* or *Friday 15th May 2026*.

- Compute Nodes: e.g. `stone-01 - 04` (GTX 1650 4 GB), `cobra-01 - 02` (RTX 3050 8 GB), `panda-01 - 02` (GTX 1080 8 GB). These are suitable for mid-sized GPU workloads and debugging GPU-dependent code.
- Batch Compute System: uses the SLURM scheduler, providing access to high-end nodes such as `gecko` (A10 GPUs 24 GB) and `falcon` (A5000 GPUs 24 GB), as well as `tiger` CPU partitions with 64-core EPYC processors. This system is designed for long-running or resource-intensive jobs.

Both are accessible to third-year students and support PyTorch workloads.

Note. If the Batch Compute System is used, then the following acknowledgement will be included as per the request on the webpage:

"The authors acknowledge the use of the Batch Compute System in the Department of Computer Science at the University of Warwick, and associated support services, in the completion of this work."

Risk	**Impact**	**Mitigation strategy**
Hardware failure or device loss	Medium	All project materials are backed up daily to the private GitHub repository whenever changes are made. In case of device failure, work can continue on the DCS machines and the compute nodes or the batch compute system if necessary.
Limited access to high-performance compute resources	Medium	If the DCS GPU queues are temporarily full, jobs will be rescheduled or left in the queue. If waiting times grow too long, cloud-based alternatives like Google Colab will be used instead. Running lighter prototype versions locally on the MacBook will serve as a fallback option.
Dependency conflicts or library version changes	Low	A dedicated Python virtual environment and a `requirements.txt` file will be maintained to keep dependencies consistent across machines.
Scope creep or time overrun	High	Progress and priorities will be reviewed fortnightly with the supervisor. The timetable is optimistic, with the winter break reserved for delays.

A.6 Legal, social, ethical and professional issues

This project does not raise any direct legal, social, or ethical concerns. All training data is generated programmatically—since no external datasets or human participants are used, there are no privacy issues.

From a broader ethical perspective, this project *contributes positively* to AI safety by developing models that can verify the correctness of their own outputs, aligning with efforts to make AI systems more reliable and transparent.

A.7 References

Artificial Analysis [2025]. *LLM Leaderboard: Comparison of Over 100 Models.* Accessed: 2025-10-10. URL: https://artificialanalysis.ai/leaderboards/models.

Huang, Lei, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu [2025]. "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions". In: *ACM Transactions on Information Systems* 43.2. ISSN: 1558-2868. DOI: 10.1145/3703155. URL: https://doi.org/10.1145/3703155.

Turpin, Miles, Julian Michael, Ethan Perez, and Samuel R. Bowman [2023]. "Language models don't always say what they think: unfaithful explanations in chain-of-thought prompting". In: *Proceedings of the 37th International Conference on Neural Information Processing Systems.* NIPS '23. New Orleans, LA, USA: Curran Associates Inc.

Wang, Boshi, Xiang Yue, and Huan Sun [Dec. 2023]. "Can ChatGPT Defend its Belief in Truth? Evaluating LLM Reasoning via Debate". In: *Findings of the Association for Computational Linguistics: EMNLP 2023.* Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, pp. 11865–11881. DOI: 10.18653/v1/2023.findings-emnlp.795. URL: https://aclanthology.org/2023.findings-emnlp.795/.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter brian, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou [2022]. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems.* Ed. by S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh. Vol. 35. Curran Associates, Inc., pp. 24824–24837. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/9d5609613524ecf4f15af0f7b31abca4-Paper-Conference.pdf.

OpenAI [2024]. *OpenAI o1 System Card.* arXiv: 2412.16720 [cs.AI]. URL: https://arxiv.org/abs/2412.16720.

Anthropic [Sept. 2025]. *Claude Sonnet 4.5: Announcement and System Card.* System card: https://www.anthropic.com/claude-sonnet-4-5-system-card. Accessed: 2025-10-14. URL: https://www.anthropic.com/news/claude-sonnet-4-5.

Google DeepMind [2025]. *Gemini 2.5: Pushing the Frontier with Advanced Reasoning, Multimodality, Long Context, and Next Generation Agentic Capabilities.* Tech. rep. Accessed: 2025-10-14. Google DeepMind. URL: https://storage.googleapis.com/deepmind-media/gemini/gemini_v2_5_report.pdf.

Cheng, Jiahao, Tiancheng Su, Jia Yuan, Guoxiu He, Jiawei Liu, Xinqi Tao, Jingwen Xie, and Huaxia Li [2025]. "Chain-of-Thought Prompting Obscures Hallucination Cues in Large Language Models: An Empirical Evaluation". In: *arXiv preprint arXiv:2506.17088v3.* Accepted to EMNLP 2025 Findings (to appear). URL: https://arxiv.org/abs/2506.17088v3.

Amit, Noga, Shafi Goldwasser, Orr Paradise, and Guy N. Rothblum [2024]. "Models That Prove Their Own Correctness". In: *ICML 2024 Workshop on Theoretical Foundations of Foundation Models.* URL: https://openreview.net/forum?id=yCm00lPmBo.

Goldwasser, S, S Micali, and C Rackoff [1985]. "The knowledge complexity of interactive proof-systems". In: *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing.* STOC '85. Providence, Rhode Island, USA: Association for Computing Machinery, pp. 291–304. ISBN: 0897911512. DOI: 10.1145/22145.22178. URL: https://doi.org/10.1145/22145.22178.