Task 3 — Thinking Deeper (Sequential Computation)

This is my writeup for **Task 3**. I describe the model, how I trained it, what "thinking longer" means in my implementation, and what I could (and could not) demonstrate from the runs I managed to save.

How I model "thinking longer" (sequential computation)

Task 3 asks for a model where "thinking longer" literally means running more sequential computation. I implemented:

$$z_{t+1} = f(z_t)$$

where f is a shared reasoning block (weights shared), applied for K steps. Increasing K is my compute budget knob.

Code: `models/task3/model.py` (`ThinkingMinePredictor`).

Model structure (more detail + why I chose it)

I wanted a model where "thinking longer" is not a metaphor — it should literally be "run the same computation more times". That pushes me toward a **shared-weight** iterative block, not a deeper one-off network.

My model has three parts:

- **Board encoder (spatial features):** I start the same way as Task 1: encode the visible board as a small vocabulary, run a conv stem + residual conv blocks, and then flatten to HW tokens. Convs are doing the short-range clue geometry work here.
- **Positional embeddings:** I use learned row/col embeddings so the Transformer knows which token came from which cell.
- **Thinking loop (sequential computation):** I apply *one* TransformerEncoderLayer repeatedly for K steps. The weights are shared, so increasing K is increasing compute budget, not increasing parameter count.

The key point is that the only knob at inference time is K. If K is larger, the model literally runs more iterations of the same reasoning block before producing logits.

Model snippet (shared-weight loop)

```
# models/task3/model.py (ThinkingMinePredictor.forward)
tokens = self._encode_tokens(x_int8)     # (B,HW,d)
per_step = []
for _ in range(steps_i):
    tokens = self.think_block(tokens)    # shared weights each iteration
    logits = self.head(tokens_to_grid(tokens)).squeeze(1)
    per_step.append(logits)
```

Inputs/outputs and data

Task 3 uses the same supervision as Task 1:

- input: visible board
- output: per-cell mine logits
- mask: loss only on unrevealed cells

I reuse the cached Task 1 datasets (teacher LogicBot trajectories), rather than regenerating data.

How I trained the model (what I actually ran)

I trained Task 3 in Colab using `notebooks/04_train_task3_colab.ipynb` and trained one model per difficulty (easy/medium/hard). My training setup mirrors Task 1, with the key change that the model has an explicit `steps` loop and I apply deep supervision:

- train/val split: random split over samples, `val_frac=0.1`
- epochs: per-difficulty caps (easy 25, medium 35, hard 40) with early stopping on validation mine-class F1 (patience = 6)
- loss: masked BCE-with-logits on unrevealed cells only, with `pos_weight` for imbalance
- deep supervision: compute a masked loss at every think step (later steps weighted more), with normalized weights so the overall loss scale stays stable

- efficiency/reliability: AMP + TF32 (when available); I intentionally do *not* use `torch.compile` because some Colab builds crash with TorchDynamo/FakeTensor (`DataDependentOutputException`), so I hard-disable Dynamo and run eagerly

Training strategy (making more steps matter)

If I only trained on the final step, early steps could stay bad. So I used **deep supervision**:

- run K steps during training
- compute masked BCE loss at every step
- weight later steps more (linear ramp)

I also use early stopping on validation mine-class F1, `pos_weight` for imbalance, and AdamW with weight decay.

On this second run, +5 epochs doesn't seem to push train/test loss and accuracy much further; we are stuck at a cap of ∼94%. So far, this has been our best performance.

v2 training results (what I saw in the logs)

These are the final metrics from my v2 run (`v2_longer_train_ci`) in Colab:

Difficulty	Epochs	Train loss	Val loss	Train F1	Val F1
Easy	25	0.0091	0.0176	0.997	0.995
Medium	35	0.0508	0.0468	0.981	0.982
Hard	40	0.1315	0.1233	0.948	0.949

Table 1: Task 3 v2 training summary (final epoch metrics from my Colab logs).

What I take away:

- Easy climbs fast and then saturates; after ∼15 epochs I'm mostly squeezing out tiny gains.
- Medium improves steadily through epoch 35, and train/val stay close (this does not look like overfitting).

- Hard is noisier/harder, but it still improves steadily through epoch 40.

One annoying caveat: Colab disconnected while I was asleep, so I could not pull the `.pt` checkpoints down to my machine to re-run gameplay evaluation later. The notebook already contains the evaluation code (loss vs steps, gameplay vs steps, heatmap evolution) — I just need to rerun it and immediately download the exported artifacts.

Versioning (v1 baseline → v2 to match the PDF better)

I treated Task 3 as a two-iteration project.

v1 baseline (what I started with). My first pass trained one model per difficulty and saved the untagged checkpoints: `task3_easy.pt`, `task3_medium.pt`, `task3_hard.pt`. That version focused on the *prediction-side* evidence: loss vs thinking steps and heatmap evolution. If I retroactively tag that baseline, I call it `v1_baseline_15ep_loss_heatmap`.

Why v1 was not enough. The PDF explicitly asks for both:

- loss goes down as thinking time increases, and
- bot performance goes up as thinking time increases.

My v1 writeup was missing the second claim in a convincing, reproducible way (no gameplay-vs-steps curve).

v2 (what I changed). In v2 (`v2_longer_train_ci`) I added gameplay-vs-steps evaluation (same seeds across steps) and I report win rate vs steps with bootstrap confidence intervals. I also export plots to `docs/figures/` so my analysis notebook can reproduce figures without copy/paste.

Showing that thinking longer helps

The notebook evaluates:

- loss vs steps (e.g., 1, 2, 4, 6, 8)
- a heatmap sequence of p(mine) after each step on the same input board
- gameplay metrics vs steps using the policy "click lowest predicted mine probability"

Variance / confidence intervals

Gameplay is stochastic (different boards / different forced-guess situations), so when I evaluate bots I want more than a single mean. The plan in my notebook is to report win rate vs steps with 95% bootstrap confidence intervals, keeping evaluation deterministic across steps by using the same seeds so "more steps" is the only variable I change.

Bot vs LogicBot

- LogicBot is best on deterministic reasoning.
- Task 3 helps most on forced guesses: more steps can refine probabilities before committing.
- It can still be worse if the distribution shifts or if the model is overconfident on rare patterns.

Issues and how I overcame them

- Without deep supervision, extra steps can become wasted compute → deep supervision fixed it.
- To make the result convincing, I evaluate loss vs steps and show per-step heatmaps (same input).

Training snippet (deep supervision)

```
final_logits, per_step = model(x_int8, steps=steps_train, return_all=True)
loss = 0.0
for i, li in enumerate(per_step, start=1):
    w = i / sum(range(1, len(per_step)+1))   # normalized weights
    loss += w * masked_bce_with_logits(li, y_mine, loss_mask, pos_weight=pos_weight)
```

Plots (generated by Notebook 05)

I generate all of the plots below using `notebooks/05_results_analysis.ipynb` and save them into `docs/figures/`.

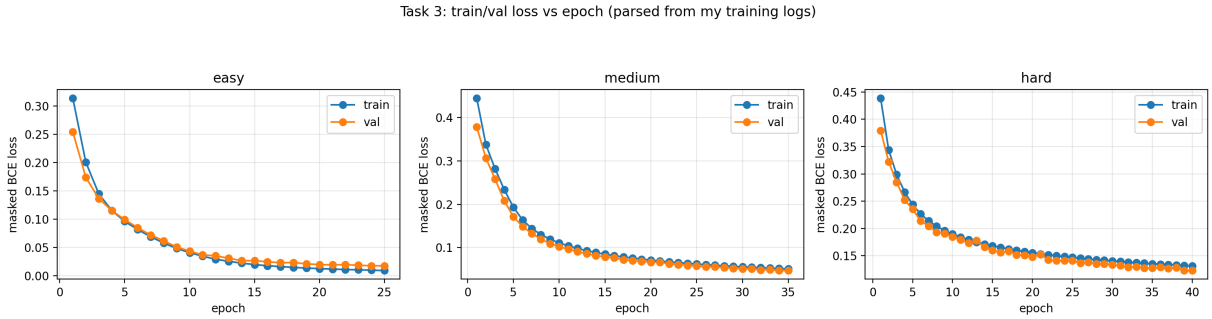

Figure 1: Task 3: train/val loss vs epoch (parsed from my Task 3 training logs).

Takeaway: all three difficulties show the same learning shape: a steep early drop in loss, then a long tail of smaller improvements. Train and val track each other closely (no obvious overfitting), with the expected ordering: easy converges fastest/lowest loss, hard converges slowest/highest loss.

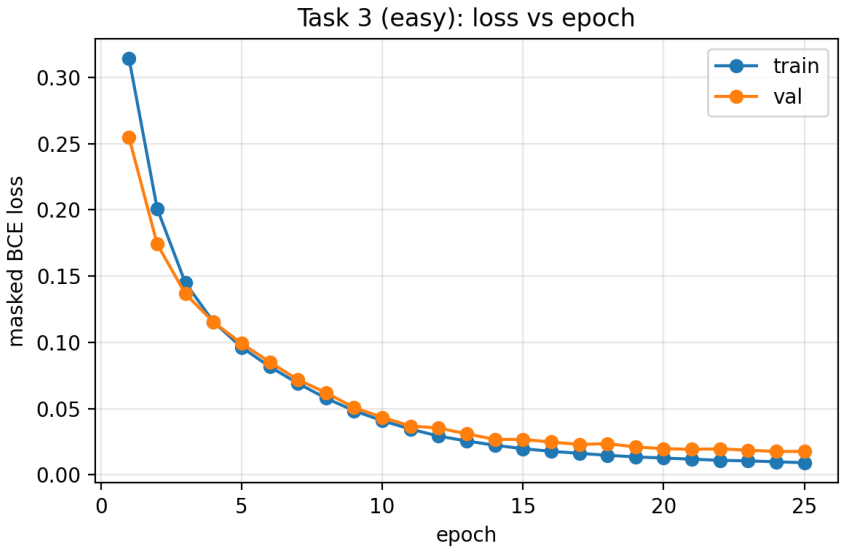

Figure 2: Task 3 (easy): loss vs epoch.

Takeaway: easy reaches diminishing returns by around epoch ∼15; after that, improvements are incremental.

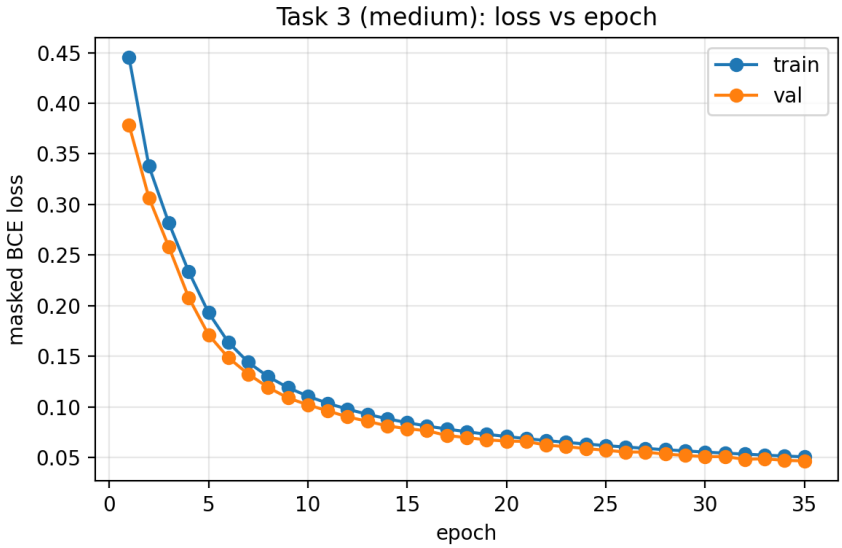

Figure 3: Task 3 (medium): loss vs epoch.

Takeaway: medium continues improving steadily through the full epoch budget, and train/val stay close (this does not look like classic overfitting).

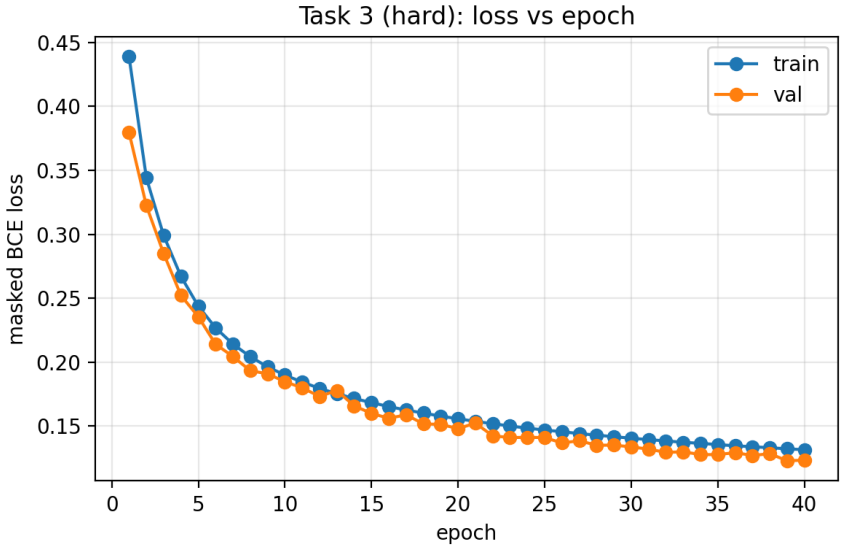

Figure 4: Task 3 (hard): loss vs epoch.

Takeaway: hard converges more slowly and ends at a higher loss (the hardest distribution), but it still improves smoothly for the whole run.

One practical note (and a real limitation of what I can show here): I trained v2 in Colab, but

Colab disconnected while I was asleep and I could not recover the v2 checkpoints. Because of that, I could not re-run the full "thinking longer" heatmap experiment for v2 (or for additional step counts beyond what I already exported). The heatmap grids below are therefore generated from the v1 checkpoints that exist in this repo.

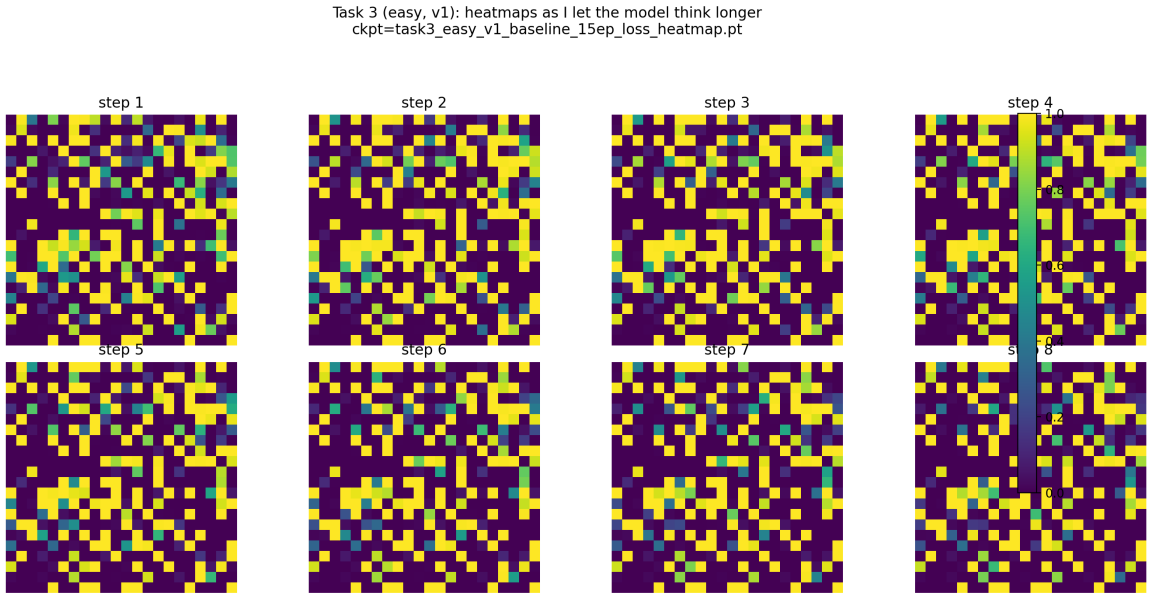

Figure 5: Task 3 (easy, v1): mine-probability heatmaps as I let the model think longer (steps 1–8).

Takeaway: in this particular v1 export, the eight panels are visually almost identical. So I cannot *claim that "more thinking steps refines the belief map" from this figure alone; the best I can say is that v1 appears to converge immediately (or fails to use extra compute), and I would need the v2 checkpoints to rerun this properly.*

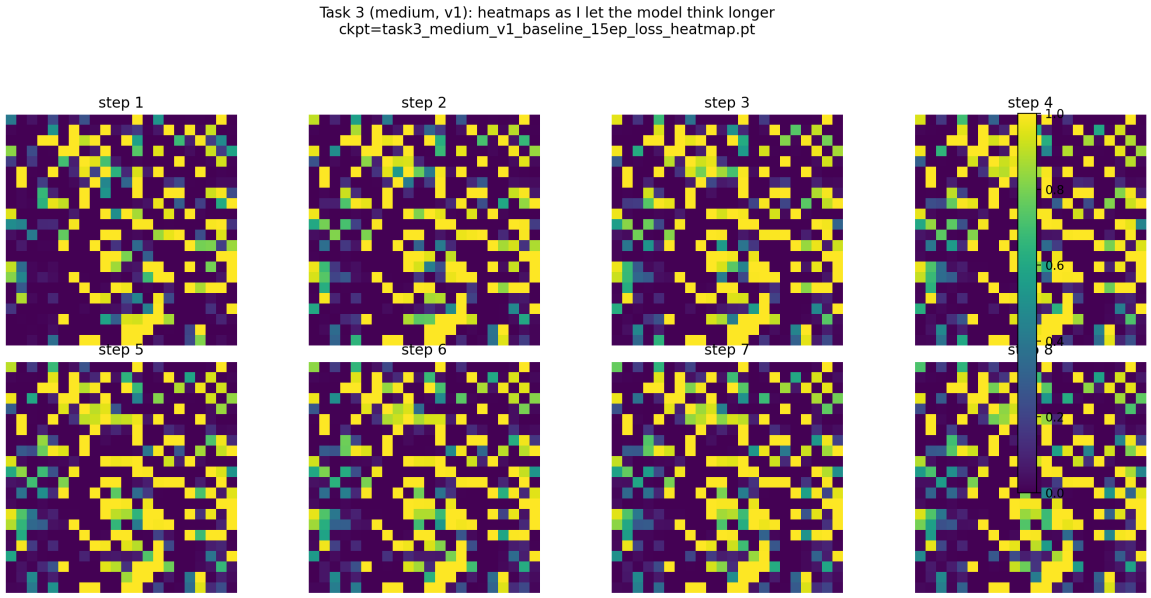

Figure 6: Task 3 (medium, v1): mine-probability heatmaps as I let the model think longer (steps 1–8).

Takeaway: same issue here — steps 1–8 look effectively the same in my saved v1 grid. This means I can't use this export to demonstrate a per-step refinement effect; I would need to rerun the evaluation with the v2 models (which I couldn't recover after the Colab disconnect).

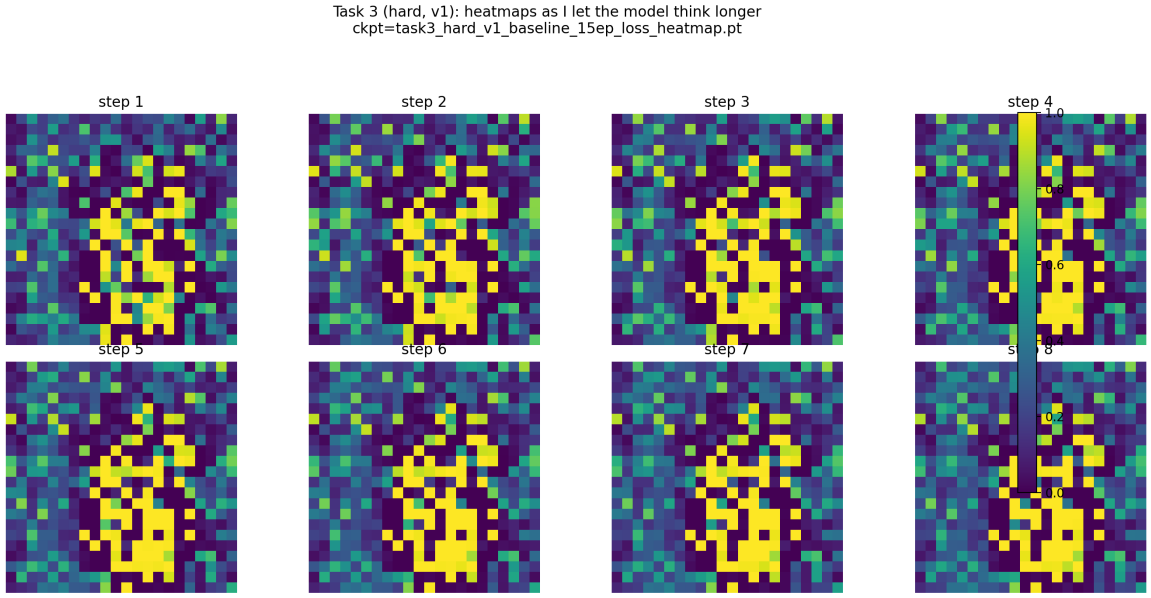

Figure 7: Task 3 (hard, v1): mine-probability heatmaps as I let the model think longer (steps 1–8).

Takeaway: hard also shows little to no visible change across steps 1–8 in this v1 export. So, again, I can't honestly claim a "thinking longer sharpens the heatmap" result from this figure; it's a placeholder showing what I intended to measure, but the missing v2 checkpoints prevented me from generating the real heatmap-evolution evidence.