Task 2 — Playing by Move Prediction (Actor / Critic)

This is my writeup for **Task 2**. I describe the model I built, how I generated data for it, how training/bootstrapping behaved, how the bot compares to LogicBot, and the main problems I hit while iterating.

What Task 2 is asking me to do (in my own words)

Task 2 is: learn a function that tells me "if I click this unrevealed cell next, how good is that decision?", and then use that learned function as an **actor** to play better than a logic bot when guesses are required. The critic learns from rollouts, and then the actor uses the critic to choose higher-value moves (bootstrapping).

What I got wrong at first (and what I changed)

This task took the most iteration for me. The biggest "gotchas" were:

- **Bad target (initially):** raw "steps survived" does not align cleanly with clearing the board, and it becomes misleading if I stop the rollout at the first mine.
- **Too-slow scoring:** scoring each candidate move by re-encoding the board killed samples/sec.
- **Bootstrapping regressions:** tiny actor datasets (especially on easy) produced noisy critics and worse later rounds.

The fixes were: (1) define survival as F/T and allow continuing after mines for measurement, (2) predict a full per-cell value map in one forward pass, and (3) add minimum-sample gates + forced-guess sample targeting + pick the best round by a lexicographic score.

What I'm predicting (survival target)

I use a normalized survival score in $[0, 1]$:

- Let T be the total number of clicks until the run reaches 100% progress (all safe cells opened).

- Let F be the (1-indexed) step of the **first** mine trigger. If no mine triggers, set $F = T$.
- Then survival $= F/T$.

Model structure

I use one model that outputs two per-cell maps in one forward pass:

- **value_map:** predicted survival score (trained against a target in $[0, 1]$)
- **mine_logit_map:** mine probability head (auxiliary)
- **Backbone:** conv/residual encoder + Transformer (global reasoning)
- **Code:** `models/task2/value_map_model.py` (`BoardValuePredictor`)

Model structure (more detail + why I chose it)

The biggest practical constraint in Task 2 was **samples/sec**. If I score every candidate click by re-encoding the board, rollouts become painfully slow. So the main architectural choice I made was:

> **Encode the board once, then score all cells in parallel.**

That is exactly what the `BoardValuePredictor` does: one forward pass returns two dense maps over the board.

- **Shared board encoder:** just like Task 1, I start from the visible board encoding and run a conv+residual stack, then a Transformer over HW tokens. The conv part learns local clue geometry; the Transformer handles longer-range interactions (multiple frontiers affecting each other).
- **Two heads:**
  - **Value head:** outputs a per-cell predicted survivability $\hat{s}(r, c) \in \mathbb{R}$, which I squash with a sigmoid when I interpret it as a $[0, 1]$ score.
  - **Mine head (auxiliary):** outputs a per-cell mine logit map. I included this because Task 2's main failure mode is "I clicked a mine", so I wanted a direct training signal that explicitly punishes mine-like cells.

- **Why the global feature concat exists:** in my implementation I concatenate a simple global summary token (mean over tokens) into the value head's input. The intuition is that "how safe is this cell to click" depends not only on local evidence, but on the overall stage of the game (how constrained the frontier is, how much information I've already uncovered).

This setup lets the actor evaluate *every* cell with one forward pass, then apply LogicBot's mask (safe vs guess set) and pick the best-scoring allowed coordinate.

Model snippet (value-map + mine head)

```
# models/task2/value_map_model.py (BoardValuePredictor.forward)
tokens = self.encode_tokens(x_int8)                 # (B,HW,d)
global_feat = tokens.mean(dim=1, keepdim=True)      # (B,1,d)
feat = torch.cat([tokens, global_feat.expand_as(tokens)], dim=-1)

value = self.value_head(feat).squeeze(-1)           # (B,HW)
mine_logit = self.mine_head(tokens).squeeze(-1)     # (B,HW)
return value.view(b,h,w), mine_logit.view(b,h,w)
```

Actor policy (how the critic becomes a bot)

For a candidate click a I score:

$$\text{score}(s, a) = \hat{s}(s, a) - \lambda \cdot \hat{p}(\text{mine} \mid s, a)$$

and pick a high-score move (with optional top-k / ϵ exploration during data collection).

I also use LogicBot inference as a mask: if there are inferred-safe cells, I restrict to them; otherwise I exclude inferred mines.

Policy snippet (scoring moves)

```
value_map = sigmoid(value_map)
mine_prob = sigmoid(mine_logit_map)
score = value_map - mine_penalty * mine_prob
```

`choose argmax over allowed coords`

Data generation (how I learned from data)

I generate supervised samples by simulation and caching:

- Round 0: play with LogicBot to get a strong baseline distribution of states.
- Rounds 1+: play with the current actor and retrain the critic on that actor's decisions (bootstrapping).
- Each labeled sample stores: `x_visible`, `action_rc`, `y_survival`, `y_mine`, and `episode_id`.

Why I don't always take the single "best" move

During data collection, always taking the greedy argmax can collapse exploration and produce a narrow dataset (the critic then overfits to the actor's early habits). So during collection I allow light exploration (top-k / ϵ-greedy), but for evaluation I keep the policy deterministic so results are comparable.

Training and overfitting prevention

- survival regression: SmoothL1 on σ(value_head)
- mine head: BCEWithLogits (mine clicks weighted more)
- split: by episode id (avoid leakage)
- optimizer: AdamW + weight decay
- early stopping on validation metrics
- practical guardrails: skip bootstrap rounds if the actor dataset is too small; target a minimum number of forced-guess samples on medium/hard

Training snippet (loss)

```
value_map, mine_logit_map = model(x_int8)
pred_surv = sigmoid(gather(value_map, a_rc))
pred_mine_logit = gather(mine_logit_map, a_rc)

loss_surv = smooth_l1_loss(pred_surv, y_survival)
loss_mine = bce_with_logits(pred_mine_logit, y_mine, weight=(1 + 4*y_mine))
loss = loss_surv + mine_loss_weight * loss_mine
```

Bot vs LogicBot

- **LogicBot is better** on deterministic reasoning (provably safe moves).
- **My actor can be better** on forced guesses by ranking risk/utility instead of uniform random guessing.
- **Easy is near-ceiling**: on easy, LogicBot already has few forced guesses, so "improvements" are hard to measure without large evaluation runs.

Because of this, on **easy** I run the actor in a practical **"guess-only"** mode: if LogicBot has a provably-safe move, I take it and never override it. This keeps easy near-ceiling and focuses the learning signal on the part of the game that actually causes failures (forced guesses on medium/hard).

Issues I ran into and how I overcame them

- Low samples/sec when scoring many actions $\rightarrow$ switched to a full value map in one forward pass.
- Misaligned labels when stopping at first mine $\rightarrow$ used survival F/T and allowed completion after mines for measurement.
- Bootstrapping regression from tiny datasets $\rightarrow$ forced-guess sample targeting, minimum-sample gates, and selecting the best round by (perfect_win_rate, avg_survival).

Results (v10_target_samples_easy_guess_only) and analysis

I summarize the key evaluation numbers here (80 games each).

Easy: My round-0 actor gets perfect win rate 0.9500 with avg survival 0.9952. On easy, LogicBot is already near ceiling (few forced guesses), so small differences are mostly noise. The practical win is that I run the easy actor in "guess-only" mode so it does not regress perfect wins by choosing different safe moves.

Medium: Round 0 actor is 0.3250 perfect win rate with avg survival 0.7453; round 1 improves to 0.4125 with avg survival 0.8050; round 2 is 0.3875 with avg survival 0.8279. This is the cleanest "bootstrapping worked" story in my runs. The key change is that actor-round collection hits the forced-guess sample target, so the critic has enough signal. My `task2_medium.pt` checkpoint corresponds to **round 1** (best by lexicographic selection on (perfect win rate, avg survival)).

Hard: Round 0 actor is 0.0625 perfect win rate with avg survival 0.4453. Later rounds are noisy: perfect wins drop (0.0375) then partially recover (0.0500) while mines triggered increases. With lexicographic selection on (perfect win rate, avg survival), my `task2_hard.pt` checkpoint corresponds to **round 0** (and `task2_easy.pt` also corresponds to round 0).

Plots (generated by Notebook 05; placeholders here)

I generate plots with `notebooks/05_results_analysis.ipynb` (Task 2 section) and save them under `docs/figures/`.

Variance / confidence intervals

For Task 2, forced guesses make the evaluation stochastic, so a single average can be misleading. In my analysis notebook I compute 95% bootstrap confidence intervals over repeated games (same difficulty, many seeds) for metrics like:

- perfect win rate
- avg_survival (my normalized F/T metric)
- avg mines triggered (when I allow continuing after mines)

This is the main reason I keep my evaluation deterministic where possible (fixed seeds for evaluation runs), and I only allow exploration during data collection.

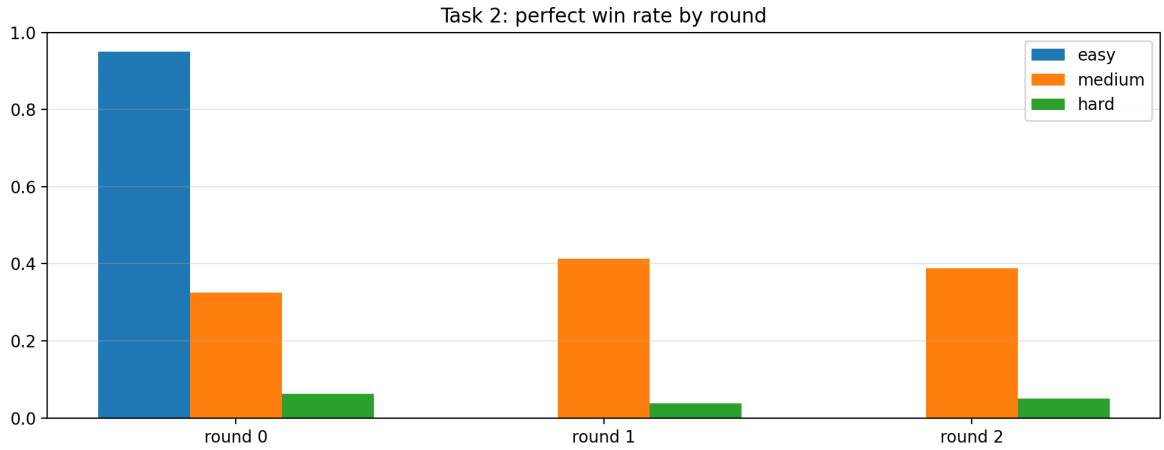

Figure 1: Task 2: perfect win rate by difficulty and bootstrap round.

Takeaway: medium shows the clearest bootstrap win (round 1 > round 0), which is the "actor learns to guess better" behavior I was aiming for. Hard stays low and noisy across rounds (guesses are much harder), and easy is near-ceiling / not meaningfully bootstrapped in my run.

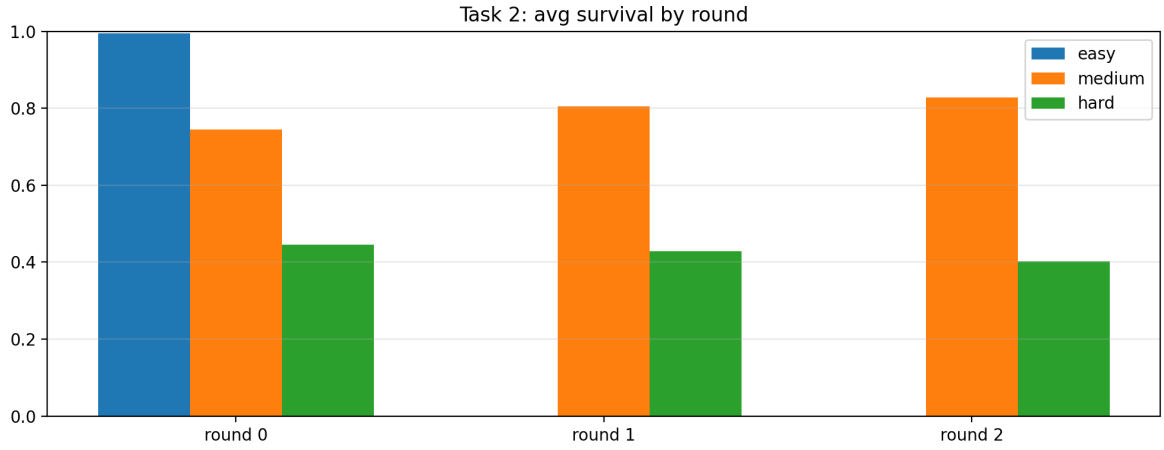

Figure 2: Task 2: avg survival by difficulty and bootstrap round.

Takeaway: medium survival improves across rounds (even when perfect-win rate wobbles), which suggests the actor is delaying its first mine trigger and making "less catastrophic" guesses. Hard survival trends downward as rounds progress, matching the story that later-round hard actors are noisy and are triggering mines earlier.

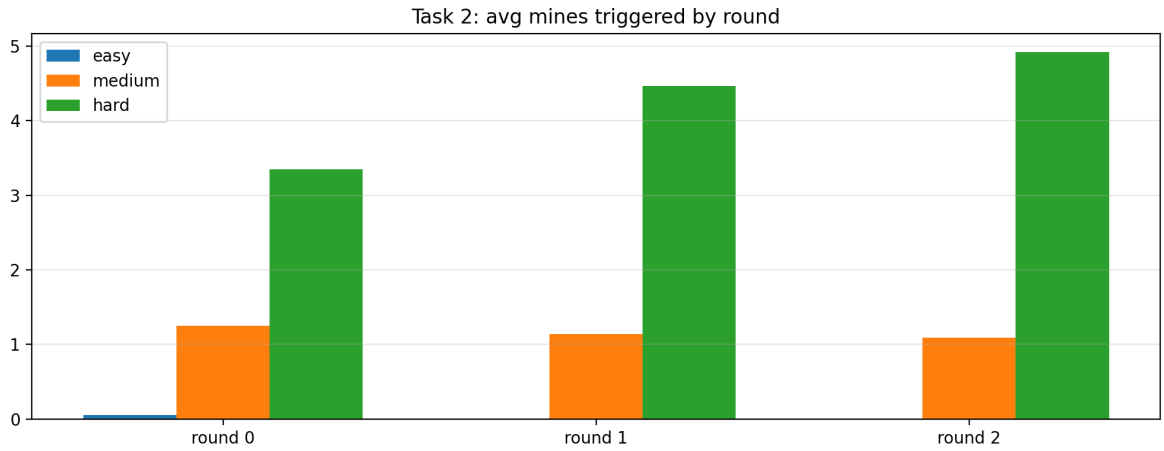

Figure 3: Task 2: avg mines triggered by difficulty and bootstrap round.

Takeaway: medium decreases slightly in mines triggered as it bootstraps (consistent with a better guessing policy), while hard increases a lot (round 1/2 trigger mines more often). This matches the perfect-win picture: medium learns something real; hard is still dominated by difficult forced guesses and variance.

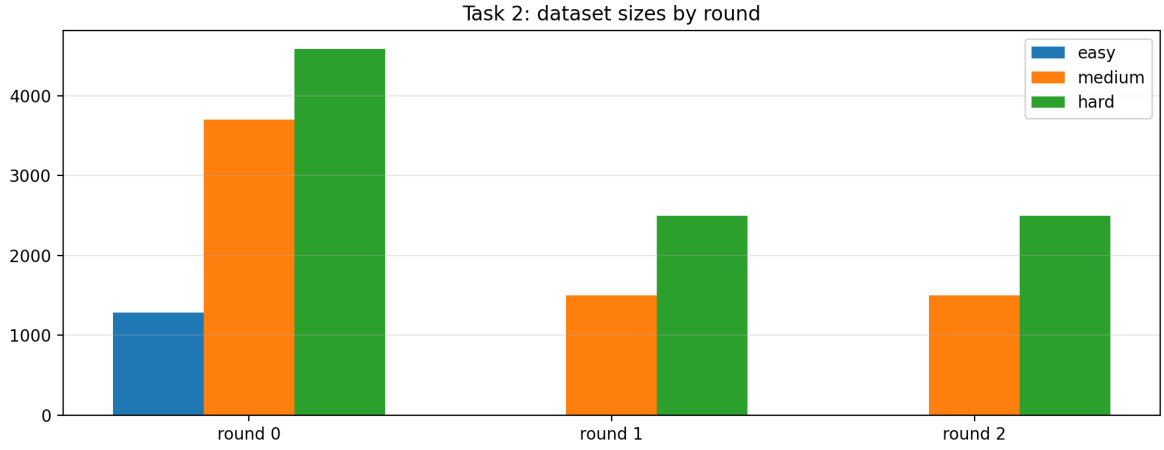

Figure 4: Task 2: dataset sizes per round (shows when bootstrapping is starved of forced-guess samples).

Takeaway: round 0 datasets are much larger because they include lots of logic-phase states, while later rounds are capped by my "target forced-guess samples" settings (and are smaller). Small actor-round datasets can make bootstrapping unstable; the critic is then fitting a narrower, noisier distribution, which is exactly what I see on hard.