Task 1 — Mine Prediction (Easy / Medium / Hard)

This is my writeup for **Task 1**. I describe the model I built, how I generated the data, how training went (and whether I overfit), how the NN bot compares to LogicBot, and what went wrong along the way.

How I trained the models (what I actually ran)

I trained three separate checkpoints (easy/medium/hard) in Colab (`notebooks/02_train_task1_colab.ipy`

- train/val split: random split over saved decision points (samples), `val_frac=0.1`
- epochs: up to 15 with early stopping on validation mine-class F1 (patience = 4)
- loss: masked BCE-with-logits on unrevealed cells only, with `pos_weight` for class imbalance
- optimizer: AdamW (lr $3 \cdot 10^{-4}$, weight decay 10^{-2})

Inputs/outputs and masking (what I supervise)

The network input is the **visible board** (what a real player sees). The environment also knows the hidden mine layout, so I can create labels:

- y**_mine:** a binary $H \times W$ mine mask from the hidden board
- **loss mask:** only unrevealed cells contribute to the loss

This masking matters: I'm not trying to predict already-revealed cells (those are trivial). I'm training only on the cells where the player still has uncertainty.

Why this is not "just accuracy" (class imbalance)

Mines are a minority class (e.g., 50/80/100 mines out of 484 cells). A model can get a deceptively high accuracy by predicting "not a mine" most of the time. Because of that, I focus on mine-class precision/recall/F1 (computed on the masked unrevealed cells) instead of treating accuracy as the main metric.

Turning the network into a bot (gameplay policy)

Once I have per-cell mine logits, I play by clicking the unrevealed cell with the **lowest predicted mine probability**. This is implemented in `models/task1/policy.py` (`select_safest_unrevealed`).

Model structure

Task 1 is a per-cell prediction problem: given the **visible** Minesweeper board, I predict which **unrevealed** cells contain mines.

- **Input:** visible board (unrevealed + revealed clues).
- **Output:** $H \times W$ mine logits/probabilities.
- **Architecture:** conv/residual encoder (local motifs) + Transformer over HW tokens (global reasoning) + per-cell head.
- **Code:** `models/task1/model.py` (`MinePredictor`).

Model structure (more detail + why I chose it)

There are two kinds of structure in Minesweeper that I wanted the network to capture:

- **Local patterns:** the 3×3 neighborhood around a clue contains most of the "micro-rules" (e.g., a 1 next to one unrevealed cell is basically a forced mine). Convolutions are a natural fit for this.
- **Global constraints:** the same clue patterns can mean different things depending on the larger context (multiple frontiers interacting). I used a Transformer so information can move across the whole board, not just locally.

Concretely, my forward pass looks like this:

- **Input encoding:** I convert the visible board into an integer grid $x \in [-1, 9]^{H \times W}$ (unrevealed is a special value). I then one-hot it into a small vocabulary channel stack. This is a simple "tokenization" step that lets the model treat each cell state as categorical rather than numeric.

- **Conv stem + residual conv blocks:** I run a conv stem and a few residual blocks to build a feature map. This is where the network learns the short-range Minesweeper motifs around clue numbers.
- **Flatten to tokens + positional encoding:** I reshape the feature map into HW tokens and add learned row/col positional embeddings, so the Transformer knows where each token came from.
- **Transformer encoder:** the Transformer mixes information across the entire board (all-to-all attention). This is my "global reasoning" component.
- **Per-cell head:** I reshape back to $H \times W$ and use a 1×1 conv head to output mine logits per cell.

I'm not claiming this is the only architecture that could work, but it matched the way I think about the task: convs handle local clue geometry; attention handles long-range coupling between frontiers.

Model snippet (core forward pass)

```
# models/task1/model.py (MinePredictor)
idx = self._to_vocab_idx(x_int8)
x_oh = F.one_hot(idx, num_classes=self.cfg.vocab_size).float()
x_oh = x_oh.permute(0, 3, 1, 2).contiguous()     # (B,C,H,W)

feats = self.conv(self.stem(x_oh))               # (B,d,H,W)
tokens = feats.permute(0, 2, 3, 1).reshape(b, h*w, d)  # (B,HW,d)
tokens = tokens + pos
tokens = self.transformer(tokens)

out = tokens.reshape(b, h, w, d).permute(0, 3, 1, 2)
mine_logits = self.head(out).squeeze(1)          # (B,H,W)
```

Data generation (how I generated the data)

I generate supervised examples by simulating many games and recording intermediate states under a LogicBot teacher:

- record the visible board
- record the hidden mine mask (from the environment)
- record a loss mask (only unrevealed cells contribute to the loss)

Datasets are cached as `.npz` so I do not regenerate every run (`notebooks/02_train_task1_colab.ipynb`).

Training objective (how I learned from it)

- masked BCE-with-logits on unrevealed cells only
- `pos_weight` to handle class imbalance (mines are the minority class)
- early stopping on validation mine-class F1 (patience = 4)
- optimizer: AdamW with weight decay

Training snippet (what I optimize)

```
logits = model(x_int8)  # (B,H,W)
loss = masked_bce_with_logits(logits, y_mine, loss_mask, pos_weight=pos_weight)
loss.backward()
opt.step()
```

Improving training / reducing overfitting

Even though train and val stayed close in my runs, I still used standard guardrails (masking, `pos_weight`, early stopping, 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%.

Notice that we have ∼96.89% final train accuracy and ∼93.97% final test accuracy, and these have both seemed to cap. Since we haven't hit higher than ∼99% train accuracy and there remains a relatively small gap between train and test loss, we can safely say that we haven't yet overfit. However, further training doesn't seem to be netting improvements.

Bot vs LogicBot

- **LogicBot is better** on deterministic reasoning: it can find provably safe moves.
- **My NN helps** on guess steps: it can rank unrevealed cells by risk (lowest predicted mine probability).
- **My NN is worse** if it becomes confidently wrong on rare patterns and it does not enforce hard logical constraints.

Issues and how I overcame them

- Accuracy is misleading under imbalance → I focused on mine-class precision/recall/F1 and used `pos_weight`.
- Random train/val split can be optimistic (correlated states from the same game) → for a stricter estimate I would split by game seed / episode (whole games).
- Plateauing metrics → I would rather add data diversity than just crank epochs.

Results analysis (what I think is happening)

The overall pattern is exactly what I want:

- big improvements early (loss down, F1 up)
- diminishing returns later
- train and val stay close (not a classic overfit signature)

Easy basically hits a ceiling by epoch 15 (train F1 $\approx$ val F1 $\approx$ 0.994). Medium ends around train F1 0.971 / val F1 0.974, and hard ends around train F1 0.929 / val F1 0.937. The ordering makes sense: hard has the highest mine density and uncertainty, so it's the hardest distribution.

Results table (final epoch snapshot)

For reference, here is the final-epoch snapshot from the run I pasted into my analysis notebook (mine metrics are on unrevealed cells):

Difficulty	Train loss	Train F1	Val loss	Val F1	Val prec	Val rec
Easy (50)	0.0147	0.994	0.0188	0.994	0.995	0.993
Medium (80)	0.0707	0.971	0.0632	0.974	0.974	0.974
Hard (100)	0.1661	0.929	0.1507	0.937	0.957	0.919

Table 1: Task 1: final-epoch mine metrics (masked to unrevealed cells).

Train/val split caveat (what could be optimistic)

Right now my train/val split is over samples, not whole games. That can be optimistic because many states from the same trajectory are correlated. If I wanted a stricter estimate of generalization, I would split by game seed / episode id (whole games held out).

Variance / confidence intervals

For the bot comparison, I did not want to report a single average and call it a day. I evaluate many games and compute 95% bootstrap confidence intervals for metrics like:

- clear rate (board cleared, even if mines were triggered)
- perfect win rate (board cleared with 0 mines triggered)
- average mines triggered (when continuing after mines is enabled)

I generate these comparisons (and the bootstrap CIs) in `notebooks/05_results_analysis.ipynb`, and I save the resulting plots into `docs/figures/`.

Plots (generated by Notebook 05)

I generate these plots with `notebooks/05_results_analysis.ipynb` and save them into `docs/figures/`.

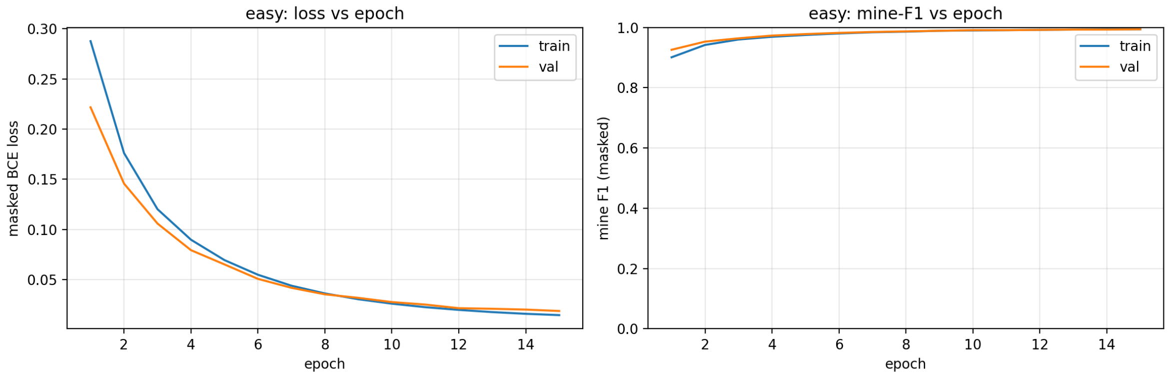

Figure 1: Task 1 (easy): train/val loss and mine-F1 vs epoch.

Takeaway: loss drops sharply in the first few epochs and then flattens out, while mine-F1 quickly saturates near 1.0. Train and val stay close, so this looks like diminishing returns rather than classic overfitting.

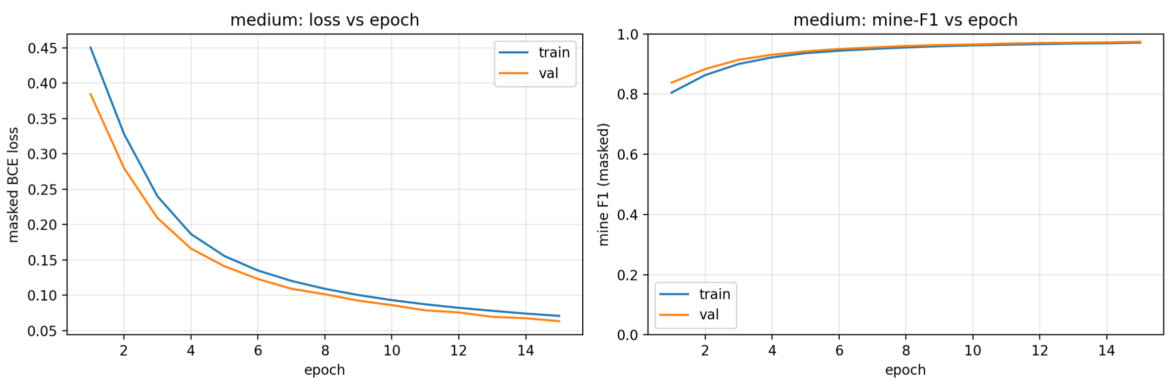

Figure 2: Task 1 (medium): train/val loss and mine-F1 vs epoch.

Takeaway: medium improves steadily over all 15 epochs, but it saturates below easy (harder distribution). The small train/val gap suggests I'm more limited by the problem/model than by overfitting.

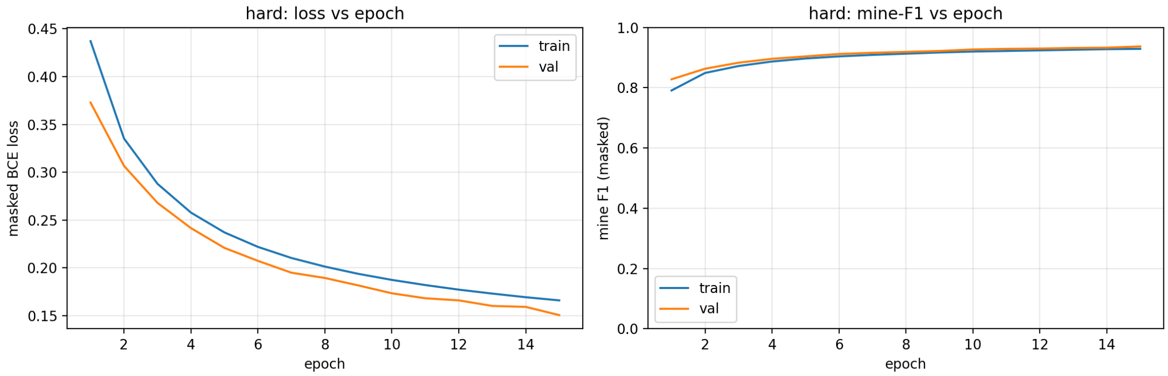

Figure 3: Task 1 (hard): train/val loss and mine-F1 vs epoch.

Takeaway: hard improves the slowest and saturates at the lowest F1 (highest mine density). Train and val move together, so this reads like a tougher inference problem rather than memorization.

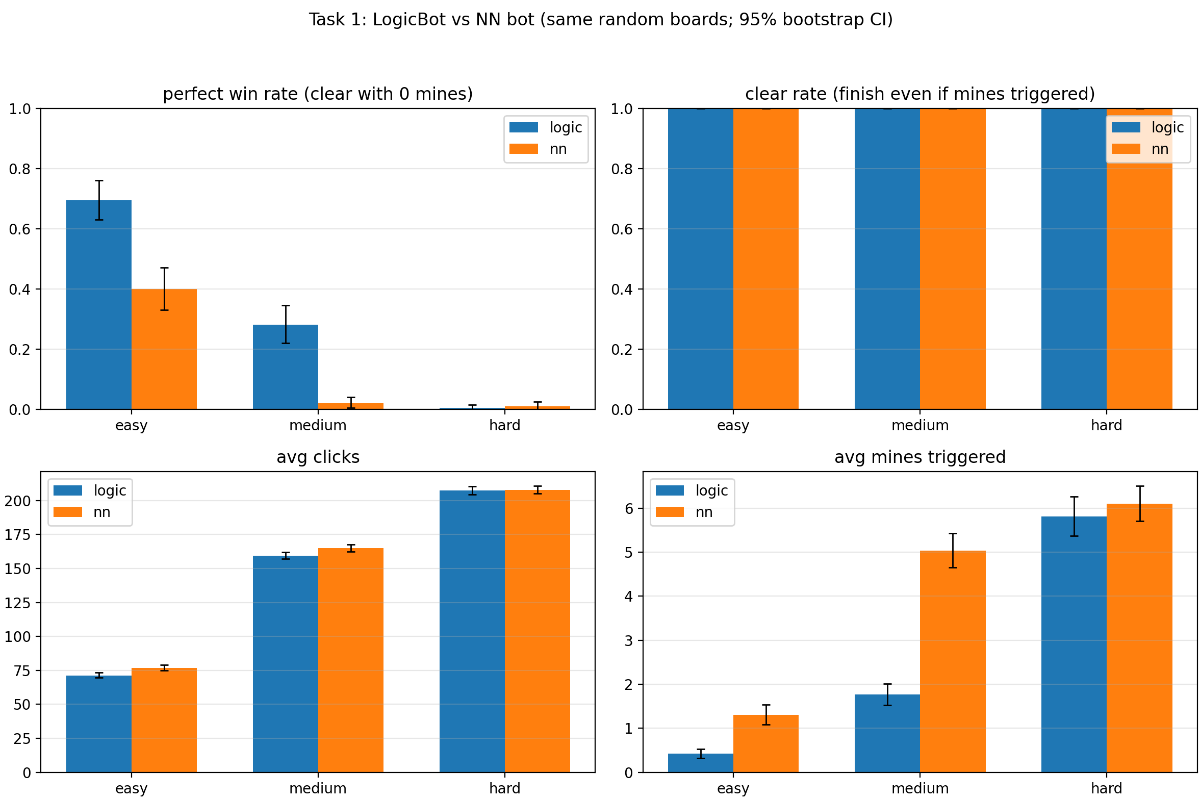

Figure 4: Task 1 gameplay: LogicBot vs my NN bot on the same random boards (95% bootstrap CI).

Takeaway: both bots usually "clear" the board when I allow continuing after mine triggers, but my NN triggers more mines (especially on medium), which collapses perfect wins. This is the *limitation of my Task 1 policy: "click the lowest predicted mine probability" does not enforce hard logical constraints the way LogicBot does.*