Final Project: Minesweeper

01:198:462

If you are unfamiliar with Minesweeper, I would recommend looking it up (Google has a built in implementation if you search for it). The rules are simple. A player is confronted with a square grid, H cells by W cells. Each cell is initially obscured, but the player can choose to reveal any unrevealed cell, one at a time. When a cell is revealed, one of two things happen:

- Either the cell is revealed to be a mine, which is then triggered, (traditionally) ending the game.
- Or the cell is revealed to not have a mine, and instead reveals a 'clue' number, from 0 to 8, indicating the number of adjacent cells that have mines.

The player is then tasked with clearing or sweeping the board - revealing cells, and using the clues to try to infer locations of mines and safely identify and clear non-mined cells. This is on some level a logic puzzle, requiring combining knowledge and information to produce inferences that can inform action.

A simple logic bot for playing minesweeper might look like this:

- Initialize the *game_environment*. The bot gets information by querying the environment.
- Initialize the following sets: *cells_remaining* (initially all cells), *inferred_safe* (initially empty), *inferred_mine* (initially empty)
- Initialize a hashmap, *clue_number*, to store for each opened cell what its clue number was.
- Loop until end of game:
  - If *inferred_safe* is not empty, pick a cell from it to open, otherwise, select a cell from *cells_remaining* at random.
  - Open the selected cell. If it is a mine, the game ends. If it is not, update the cell sets appropriately, and save the clue number.
  - For each cell with a revealed clue:
    - If (cell clue) - (# neighbors inferred to be mines) = (# unrevealed neighbors), mark each unrevealed neighbor as a mine, and remove them from *cells_remaining*.
    - If ((# neighbors) - (cell clue)) - (# neighbors revealed or inferred to be safe) = (# unrevealed neighbors), mark each unrevealed neighbor as safe, and remove them from *cells_remaining*.
  - If any cells were inferred as safe or mine during the above for loop, repeat the loop, until no new inferences are found.
- Return the number of cells successfully opened or revealed as safe, and the number of mines triggered.

The goal of this assignment is to build a minesweeper bot, based on a neural network, and train it to successfully play the game, as good and *ideally better* than the above bot.

> Note, traditionally the first cell, which must be chosen as random, is taken to not be a mine - the game environment might return a non-mine cell with a 0 clue value, selected randomly, at the start of the game, to get the process going.

1 Task 1: Playing by Mine Prediction

Consider playing the game on a 22x22 board, with either 50 mines (easy), 80 mines (medium), or 100 mines (hard). Your goal in this section is to build a neural network that takes as input a current state of the game, and gives as output a prediction of which cells are safe to open. The agent should iterate on this - evaluating the network on the current board, making a determination of which cell to uncover, uncovering that cell, and repeating, until the game is complete. Note, your bot will not implement any additional *reasoning* outside the learned calculation of the underlying networks.

Be clear about the following:

- How are you representing your input? What will your 'actual' output be that you are trying to predict?
- What predicted output are you going to be calculating?
- What model structure are you using?
- How can you assess the quality of your model?

Training will undoubtedly be done in the usual way, but be clear in your writeup about anything you do to make training easier or more tractable, and what you do to avoid or handle the problem of overfitting.

For this task, aside from training the network, the hardest part is *understanding what your data should be.* In many tasks we've seen, a data set has been provided, demonstrating desired inputs and outputs. But you don't have that here. How can you generate data for this task, what data should you generate, and how can you use it? For instance, while the input to the network should be a representation of the information the player has available (i.e., not the locations of hidden mines), you can potentially predict any output that you have the data to learn from. So what would be useful?

Goals for this section:

- Train a network to at least meet or mimic the performance of the logic bot.
- Ultimately, train it to *exceed* the performance of the logic bot. How could this be done?
- Draw on as many concepts and ideas from the course as possible in the development and training of your network.

Note, a simple CNN will not suffice here! Note that one cell being a mine or safe can actually provide information about other cells that are much further away.

Aside from the above considerations, your writeup should also include a comparison between your neural network bots and the logic bot for each of the three minesweeper difficulties. Note, you are welcome to train a different network for each difficulty setting. In this comparison, I would like to see:

- How often the logic bot clear the board vs how often your neural network bot does.
- The number of steps each bot survives, on average.
- If the bots are allowed to trigger mines, and keep going with that information, the average number of mines set off by the time the last safe cell is opened.

- Are there any situations / board configurations where the logic bot and the network bot make different decisions, and if so, why? Is the network bot making a *better* decision?

In order to make the above comparisons, you'll need to assess not only averages over play data, but also variance or confidence intervals on the results.

You should aim for your bots to be superior to the logic bot, but as indicated above, there are several potential metrics for comparison.

2 Task 2: Playing by Move Prediction

As an alternate approach: instead of predicting what cells are safe or mined, consider the problem of predicting *how well the logic bot is going to do.* That is, given the current state of the board, and a selected, unopen cell - can you predict (on average) how many moves the logic bot is going to survive? Build, train, and evaluate a network to do so. Be thorough in your discussion of any design choices, and in your analysis of its performance and results.

But then notice the following: having trained this network, you may find that there are some choices for a given board that actually have a *higher predicted survival duration* than the choice the logic bot ends up making. This suggests the following - designing a new bot, based on taking moves with higher predicted survivability than the moves the logic bot would have taken.

Iterate on this process: consider two networks based bots.

- The Actor: predicts values for every possible move based on the current state of the board, and then takes a move of high predicted value (*is always taking the 'best' move a good idea? Why or why not?*) until the game ends.
- The Critic: based on the decisions of the actor, the critic attempts to learn to predict how long the actor will survive based on the current state of the board, and the selected cell to uncover.

By collecting data on the Actor's performance, the Critic can be trained to predict utility of the Actor's decisions. But once the Critic is trained, the Critic's network can be used by the Actor to make better decisions. And the process repeats - bootstrapping the Actor to greater success (ideally).

Build and implement an actor/critic model to play the game (medium difficulty). Be clear on your data collection strategies, your network designs, training choices, and any issues you run into along the way. Be sure to demonstrate that a) your critic is successfully learning how to predict the performance of your actor, and b) the actor's performance is improving over time.

Note, I imply above the idea of using the logic bot as the 'initial actor'. Is this necessary? Is this useful?

3 Task 3: Thinking Deeper

This last task is somewhat speculative. Consider the problem of minesweeper, and in general the problem of logical reasoning. Intuitively, the longer you think about something, the more you can conclude, and the more certain you can be in your conclusions. (This is one reason that one way to level the playing field against an opponent in chess is to limit how much time they have to think about their move.)

Consider the following: implement a network capable of 'thinking longer' about the minesweeper game, by modeling it as a sequential process or calculation - in this case, 'thinking longer' translates directly to running the sequential processing of the network further.

Note, at the time of posting, we have not covered sequential models yet, but we will soon.

- How can you model this as a sequential computation? What topics or ideas from class are going to be relevant here?
- What has to change about your data collection process or training process, in order to make this work?
- Build and train a model to do this (again, drawing on as much from class as you can to do so - sequential models and resnets will both be relevant here, I think). Show that
  - The loss in predictions goes down the longer you give the model to think about its next move.
  - The performance of the network bot goes up the longer you give the model to think about its next move.
- Illustrate this with a heat map of predicted mine locations for a partially revealed board, and show how that heat map changes with the amount of 'thinking' time.

4 Writeup and Requirements

In addition to answering the questions laid out in the above sections, your writeup should be clear about:

- The structure of your models.
- How you generated the data you used, and how you learned from it.
- Anything you did to improve training and reduce overfitting.
- Where, if anywhere, your bot is superior or inferior to the logic bot, and if so, why.
- Any issues you ran into, and how you overcame them.

Lastly, to reiterate - I would like you to try to draw on as much material as possible from what we've covered. I don't think it's particularly surprising that convolutional networks may be useful here. But what else, and from what other topics or areas?