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. There are three tiers of task to complete, and a fourth bonus task.

> 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: Traditional Minesweeper Boards

There are three tiers of traditional minesweeper:

- Easy: a 9x9 board with 10 mines.
- Intermediate: a 16x16 board with 40 mines.
- Expert: a 30x16 board with 99 mines.

For each of these cases, you need to design an agent for playing the game, based on a neural network. The neural network should take as input some representation of the current state of the game, and give as output something that can be used to determine what cell to open next. The agent should iterate this (updating the state of the game, computing via the network the next thing to open), until the game is completed.

> Note: Your agent should implement no reasoning outside the neural network calculation. The question is not, can you the student write a program to play minesweeper - the question is can a neural network learn how to play minesweeper.

Be clear about the following:

- How are you representing your input?
- What output are you going to be calculating, and how is it used to pick a cell to open?
- 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?

Theoretically you could learn to mimic the logic bot (*how?*), but how could your network bot learn to be *better*?

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: Variable Numbers of Mines

In the previous section, the bots you trained were always faced with the exact same total number of mines (for each difficulty setting). While you may not have explicitly expressed the total number of mines to the network during training, it would've implicitly learned to 'expect' a given number of mines.

In this task, I'd like you to build and train a network capable of performing well regardless (within limits) of how many mines there are on the board.

For a 30x30 board, build and train a network to play minesweeper where the number of mines may be anywhere from 0% to 30% of the board. (Note, in expert difficulty, the mines cover about 20% of the board). What has to change in your data, model, and training?

For a comparison between your bot here and the original logic bot, consider plotting as a function of how many mines there are

- Probability of clearing the board (logic bot vs network bot).
- Average number of steps before hitting the first mine (logic bot vs network bot).
- If the game is allowed to continue until the last safe square is revealed, the average number of mines triggered to finish the game (logic bot vs network bot).

You'll need to generate enough data to make the comparisons between the two bots clear.

Again, you should aim for your bot to be superior to the logic bot, but as before there are several potential metrics for comparison.

3 Task 3: Variable Size Boards

In the previous two tasks, the boards the network was parsing were always of a fixed size. For this task, you want to construct a network architecture capable of playing the game on variable sized boards. (You may assume the boards are square.) I would like your network to be able to handle boards of any size $K \times K$ for $K \geq 5$. How does your model have to change in order to accommodate this? How does your data or training have to change?

For a comparison between your bot and the logic bot, create a plot of performances as a function of K, for K from 5 to 50, with $\approx 20\%$ mines or more. Include at least the following:

- Probability of clearing the board (logic vs network).
- Average steps before hitting the first mine (logic vs network).

- If the game is allowed to continue until the last safe square is revealed, the average number of mines triggered to finish the game (logic vs network).

As before, you should aim for your bot to be superior to the logic bot, but as before there are several potential metrics for comparison.

To emphasize: I want *a single network* that can be used to play on boards of any size. I do not want a whole suite of models for all possible sizes. And I do not want you to hardcode an upper limit of 50 into your board implementation - I'm only drawing a limit there for plot reasons.

In the previous two tasks, the main issue (aside from training) was constructing and utilizing a good data set. That should largely be resolved by this point - the more interesting question for this task is in terms of network architecture, to handle this flexible input size.

4 Bonus Task: Board Generation

As a stretch goal (I am hoping we get deep enough into the relevant material) - how could you approach the problem of generating good boards to play on? In particular, how could a neural network learn to generate minesweeper boards that your previous bots perform well on?

I'll expand on this more when we talk about generative models, but for now - how could you construct a model that outputs minesweeper boards? How could you evaluate how good that model is?

5 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.

Lastly, 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?

Two questions that I would explicitly like you to experiment with and try to answer:

- Should the game be viewed sequentially, or does the current state of the game suffice?
- Can **attention** be applied here in a useful way?

Note, at the time of posting, we will not have talked about sequential models or attention yet, but we are getting there shortly.