CSCI 0410/1411 Spring 2025
Final Project Part 1: Go Engine

Milestones	Release Date	Due Date	Due Time
Part 1 (Heuristic Search and Learning)	4/9	4/16	11:59 pm ET
Part 2 (Anytime Search)	4/16	4/25	11:59 pm ET
Final Bot & Writeup	4/25	5/6	11:59 pm ET
Tournament Ends		5/12	11:59 pm ET

1 Introduction

Go (*a.k.a.* Wieqi, Igo, or Baduk) is a two-player zero-sum game of strategy popular throughout the world. Game-playing agents have long been a focus of the AI community. Indeed, the term "machine learning" was dubbed by Arthur Samuel in 1959, while building a Checkers-playing agent that learned by playing against itself. Since then, AI researchers have capitalized on Samuel's idea of self-play to build successful game-playing agents for many parlor games, including Backgammon (TDGammon, 1992) and Chess (DeepBlue, 1997). Nonetheless, until very recently, it was thought that Go-playing agents that could perform as well as humans were out of reach. When Alpha-Go defeated the world champion Lee Sedol in 2016, it shocked the AI and Go communities alike. In this project, you will implement game-playing agents for the game of Go. You, too, may be shocked by how well your agent learns to play!

2 Learning Objectives

The goal of the final project is to help you synthesize the material you've learned throughout this course. We will provide structure for Parts 1 and 2, but the techniques you choose to use for Part 3, your final agent, will be up to you.

What you will be able to do:

- Combine techniques from different subfields of AI to produce a strong Go-playing agent
- Design experiments to evaluate the performance of different game-playing agents, and summarize the results in easy-to-interpret tables or plots

What you will understand:

- How novel methods in deep learning can be combined with classical methods in AI to improve upon each.

3 Setup

There are two possible *backends* you can use in this project. The *backend* refers to the game engine that runs the game of Go (i.e., checks valid moves, scores the game, and keeps track of related info). The first option for a backend is Open-Spiel, a library developed and maintained by Google Deepmind for research on learning and games. It contains implementations of many popular games, including Go. You can install it with the following command:

```
pip install open_spiel
```

Unfortunately, Open-Spiel does not cooperate well with the windows operating system. It is possible to install, but with some additional work.

The alternative is to use the implementation of Go provided in the `pygo` folder of your repository. If you do not have `open_spiel` installed, it will be used automatically (i.e., you don't have to do anything special). So what's the difference and why have two options (and why is open-spiel hard to install on a Windows computer)? Open-Spiel uses an implementation of Go written in C++ and compiled to execute very quickly. When running a large number of games, every optimization can help. When we evaluate your code (i.e., on Gradescope and during the final tournament), the open-spiel backend will be used to give results faster.

4 Go

Like Checkers and Chess, Go is a two-player, zero-sum, deterministic, alternating-move game. Also like Checkers and Chess, Go is played on grid. Two players alternate placing colored stones on the grid's intersection points, called **territory**. The first player to move places a black stone, while the second player places a white stone. The two then alternate until the game ends. If one player completely encircles any of the other's stones, the encircled stones are captured, and removed from the board. At any point, a player may pass. The game ends when no legal moves are available, or if both players pass consecutively. When the game ends, the players tally their total territory.[1] The player who has surrounded more territory wins.

Go can be played on a variety of board sizes, ranging from as small as 5x5 (beginner play) up to 19x19 (professional play). In this project, your agents will play on 5x5 and 9x9 boards.[2]

The player who moves first (Black, in Go) typically has a significant advantage, as in Tic-Tac-Toe, Connect 4, Chess, etc. To account for this advantage, additional points, called komi, are added to White's score at the end of the game. We will be using a komi of 5.5 in this project for 9x9 games. This means that Black has to score 6 or more points better than White to win. For games on a 5x5 board, a komi of 0.5 is used. If a fractional value is used for komi, the game cannot end in a draw.

We highly encourage you to try out an interactive Go tutorial, such as this one, to better understand the rules of Go. There are also many video introductions to Go available online.

4.1 Time Controls

Many competitive board games are played with per-move or per-game time limits; without such limits, games could take forever! These time limits are often referred to as *time controls*. At the start of each player's turn, a clock starts counting down. After the player makes a move, the opponent's clock begins. If a player's remaining time runs out before a move is made, that player forfeits the game.

Our games will use **incremental** time control, in which each player is allocated an initial block of time (in this project, 15 seconds), plus an additional increment of time after every move (in this project, 1 second). For example, if a game lasts 200 moves, then the total number of seconds of search time would be 215 seconds. Time management, meaning how a player uses the available time, is key to a successful Go agent.[3]

5 Part 1 Phase 1: Heuristic Tree Search

Your first task in this project is to create a set of simple benchmarking agents to have on hand for evaluating the more sophisticated agents you will build later.

Our implementation of Go implements the same abstract `AdversarialSearchProblem` interface we used in Homework 2, when you built minimax and $\alpha\beta$-pruning agents to play Tic-Tac-Toe and Connect 4. As a result, you can import your implementations of minimax and $\alpha\beta$-pruning into your final project code base.

[1] There are many other variations of scoring for Go, such as counting captured pieces as part of the score.
[2] If you want to play against your agent, the GUI is much faster on 5x5 boards. RL is also much faster on 5x5 boards.
[3] and to a successful Brown student!

As the state space in Go is vast (e.g., there are 81 intersections on a 9×9 board, making for 2^{81} states), it will not be feasible to run minimax and $\alpha\beta$-pruning without limiting the depth of the search. Recall that the depth-limited versions of these algorithms require a heuristic, i.e., a means of evaluating the quality of an arbitrary state. We have provided a very simple heuristic, `GoProblemSimpleHeuristic`, to jump start your use of your existing adversarial search implementations. This heuristic evaluates a state by computing the difference between the number of black and white stones on the board. We also include an implementation of `GreedyAgent`, which uses `GoProblemSimpleHeuristic` to greedily select an action.

In Homework 2, your agents could take as long as you were willing to wait for them to make decisions. In contrast, in this project, time is not unlimited. Since you are allocated an additional one second (beyond the initial 15) for every move, the easiest way to make sure your agent does not run out of time is to make sure it uses less than one second per move. For minimax and $\alpha\beta$-pruning, one way to do this is to find a search depth that always terminates within one second.

However, fixing a search depth throughout a game is very limiting. At the start of the game, Black, the first player, has 81 legal moves.[4] The second player then has 80 legal moves, which yields a total of $(81)(80) = 6840$ possible states after the first two moves. Later in the game, when there are, for example, 20 available moves, searching to a depth of 3 plies would search the same number of states as just 2 plies at the start of the game ($((20)(19)(18) = 6840$).[5]

Rather than fixing a search depth in advance, an alternative is to build an agent that searches for a fixed amount of time. Assuming there is just enough time to search 6480 states, such an agent would lookahead through depth 2 at the start of Go, and through depth 3 after 20 moves. This approach can be realized using an **anytime** algorithm, which, as the name suggests, can return a solution at any time! When the time comes, an anytime algorithm simply returns the best solution found so far. In part 2 of the final project, you will write two anytime algorithms. For now, you will be porting your implementations of minimax and $\alpha\beta$-pruning to your Go code base, so that you can include them in your initial experimentation.

Phase 1 Tasks

1. Run 100 games of `GreedyAgent` vs `RandomAgent` using the command:

   ```
   python game_runner.py --agent1-type greedy --agent2-type random
       --mode tournament --num-games 100
   ```

   A tournament outputs the following statistics:

   - Score for each agent, i.e., total wins − total losses
   - Score as Black (first mover) for each agent
   - Statistics about the amount of time used by each agent to find its moves

   What are the results? Do the scores surprise you at all? Report your findings in your README.

2. Play a few games against the `GreedyAgent` using the command:

   ```
   python gamerunner.py --agent1-type greedy --mode gui
   ```

   Note: Use the arrow keys to select a cell to play and use the enter key to take an action.

   Or, if you are using Github Codespaces, windows WSL, or some other system without graphics capabilities, you can also run:

[4] Ignoring *pass* for simplicity.
[5] Assuming no captures took place and no player passed.

```
python gamerunner.py --agent1-type greedy --mode text
```

What is `GreedyAgent`'s strategy? Why, across multiple games, is it always more or less the same? What is the problem? Describe your findings in your README.

3. Modify `GreedyAgent` to randomly select among all actions with the best heuristic value. Does this alter its performance against `RandomAgent`? Why are the results so different? Explain your results in your README.

4. Implement `MinimaxAgent` and `AlphaBetaAgent` in `agents.py`. As these agents implement the `GameAgent` interface, all that is required is that you implement the `get_move` method for each agent. This method should simply call your implementations of `MiniMax` and `AlphaBetaSearch` from Homework 2.

   - We have included command line arguments for `AlphaBetaAgent` in `game_runner.py`. If you'd like to use `game_runner.py` for new agents, you can add them to the `create_agent` method of `game_runner.py`. Add `MinimaxAgent` to `create_agent` if you'd like to use it with `game_runner.py`.
   - Tip: In this project, the terminal values of the game are either $+1$ or -1. In contrast, in Connect 4 in Homework 2, the values were $+\infty$ or $-\infty$. You may have to adapt your Homework 2 implementations slightly to handle this change.
   - Tip: `get_move` takes in `time_remaining` as an argument. Minimax and $\alpha\beta-$search do not need to use this variable. It will become useful for other search algorithms in part 2 and 3.

5. Experiment with both of your agents to discover a search depth that obeys Go's time constraints. What is this search depth for each agent? Include your answers in your README.

6 Part 1 Phase 2: Learning a Heuristic

In this phase of part 1, you will be working on the `supervised_learning.ipynb` notebook. Completing the notebook will complete most of the tasks described in the following section.

The Data

Supervised learning is the task of building a function approximator (i.e., a model) from labeled data. We provide the dataset for you to learn from, in the files `5x5_dataset.pkl`. The dataset consist of a list of tuples of the form (state, action, outcome).

The 5×5 version of Go is used to teach people to play the game, but most people who fancy the game are quick to move on to larger boards. As a result, there are very few high-level games available to learn from. We therefore provide a dataset of games played by our `MCTSAgent` (which you will implement in Part 2), which will outperform the other agents you built for Part 1 of the project so far.

Task 2.1: Feature Encoding

Recall that supervised learning, or function approximation, is a means of building a model from inputs to outputs. The inputs are generally described in terms of **features**, such as the position and velocity of a cartpole. The Go dataset, however, does not contain features. It contains `GameState` objects. As a result, your first task is to write an **encoder**, meaning a function that converts `GameState`s into feature vectors.

What makes for a good encoding, or a good choice of features? We contend that an encoding should be both *expressive* and *informative*. An expressive encoding is one that is capable of expressing many, if not all, of the possible inputs to the model, in our case the `GameState`s. An informative encoding is encodes information about the input that is relevant to the task at hand, in our case, playing the game of Go.

An expressive feature encoding for the game of Go should encode the positions of all the stones on the board. As a first attempt, you might try representing this information using two lists of coordinates, one for Black and another for White. A difficulty with this approach, however, is these list sizes are variable.[6]

How can we instead represent all possible `Gamestate`s using features that are constant in size? The board size is constant, so perhaps we could use one feature per cell—25 features for a 5×5 board—each of which can take on one of three possible values. For example, we could represent an empty cell by 0, a cell with a white stone on it by 1, and a cell with a black stone on it by 2. Encoding categorical variables as continuous values is *not* generally a good idea, however, because a model can ascribe meaning to the continuous values where there is none: e.g., it might surmise that black stones are worth twice as much as white stones.

The preferred way to represent categorical features is to use a **one-hot encoding**. To build a one-hot encoding of the aforementioned representation of the positions of the stones on a 5×5 Go board, i.e., 25 features, each of which can take on three possible values, we create $25 \times 3 = 75$ binary features. The first 25 are on or off depending on whether the cell is empty or not; the next 25 are on or off depending on whether the cell is occupied by a white stone or not; and the final 25 are on or off depending on whether the cell is occupied by a black stone or not. Note that there is redundancy in this feature representation: if a cell is occupied, it is not empty, and vice versa. Indeed, the 50 features indicative of the black or white stones' positions are alone sufficient to represent all the possible positions of the stones on a 5×5 Go board.

One additional feature is necessary to fully capture `GameState`, namely the player-to-move. These 51 binary features are sufficient to represent all the `GameState`s in 5×5 Go. In other words, this encoding is fully *expressive*. But expressivity alone is not enough; your encoding must also be informative! If you do not also encode enough relevant information about your inputs in your features, then no matter big your dataset is, and how fancy your neural network is, it will never be able to learn effectively. In the case of games (and single-agent sequential decision making), an informative feature set encodes information about the state that facilitates choosing good actions (or making good decisions).

	Not Expressive	**Expressive**
Not Informative	-	A single integer feature between 0 and 2^{51}, where each game state is mapped to a unique value
Informative	Number of pieces per player	**Your Goal**

Feature engineering/encoding is an often overlooked aspect of machine learning, but it can be key to the practical success of models. The neural network built for AlphaGo used 17 features per grid cell on a 19×19 board for a grand total of 6137 features! You do not need to use anywhere near that many features in this project, but you should dream up a few informative features, and then use them together with the expressive encoding we described above to improve your models.

Task Write `get_features`, which encodes `GameState`s as feature vectors. Test its correctness on a few sample inputs and outputs.

Note You cannot evaluate the efficacy of your encoding until you train a model to fit your dataset (Tasks 2.2) and build agents based on these models (Task 2.3).

Task 2.2: Learning a Value Function

Adversarial games like Go are sequential decision problems, albeit for two players. By fixing the strategy of the opponent (e.g., the dealer in Blackjack), these games can be conceptualized as (single-agent) MDPs.

[6] There are techniques to handle variable input sizes. Large Language Models, for instance, handle different context sizes (i.e., number of words in the prompts).

When a zero-sum game with rewards $+1$ and -1 is viewed as an MDP, the value of a state—by definition, the expected sum of future rewards—is indicative of how likely the agent is to win or lose the game from that state. Learning a value function can therefore be framed as a binary classification problem, where the goal is to classify each state as a future win for one player or the other, or more generally, to generate a prediction in the range $[-1, +1]$ that is indicative of which player will win the game.

Task Your task is to implement a neural network that represents a value function and to train it using the data provided to predict the outcome of a game given a(n encoded) state. Be sure that your training and test error decrease as your model learns.

Tip #1 Pytorch provides a number of built in loss functions. We covered some of these in class (e.g., mean squared error, log-loss/cross entropy). Which one is applicable to predicting the outcome of a game?

Tip #2 In previous assignments, you implemented gradient descent from scratch (i.e., computing $\theta - \alpha \nabla_\theta f$ explicitly). Pytorch provides a powerful set of optimizers that implement variations of **stochastic gradient descent (SGD)**. In the stencil code, we provide an example of how to use one these optimizers, namely Adam. Adam (ADAptive Momentum) is a variation of SGD that uses momentum to try to escape local minima quickly. We provide you with an outline of the training loop, but leave it to you to fill in the details.

Task 2.3: Let's Play!

Your next task is to incorporate your supervised learning models into a Go-playing agent. There are myriad ways to accomplish this task.[7]

Task Implement an agent that utilizes your learned value function as a heuristic. You can first implement this agent within `supervised_learning.ipynb`, but you need to copy the relevant agent code to `agents.py` and `heuristic_go_search_problems.py` before moving on so that you can use this agent later. You will need to include the agent definition in `agents.py`, the learned heuristic search problem definition `learned_go_search_problems.py`, and the feature encoding function used in `learned_go_search_problems.py`.

7 Downloads

You can access the support and stencil code for this assignment through Github Classroom.

7.1 Support Code

- This project uses `GameAgent`, `GameState`, `AdversarialSearchProblem`, and `GameUI` from Homework 2. These classes can be found in `adversarial_search_problem.py`
- `GoSearchProblem.py`: Provides implementations of `GoState` and `GoSearchProblem`, which implement `GameState` and `AdversarialSearchProblem`, respectively.
  - One thing to note is a (perhaps unintuitive at first) design choice made by the developers of OpenSpiel, namely that `Actions` in `GoSearchProblem.py` are integers, not pairs of (x, y) coordinates. In particular, the actions available on an empty 9×9 Go board are $\{0, 1, \ldots, 81\}$. The first 81 of these actions (0 through 80) actions correspond to locations on the board and the final action (81) corresponds to pass. To convert from an integer action a to coordinates on an $n \times n$ board, you can compute $(x, y) = (a \mod n, \lfloor a/n \rfloor)$.

[7] The AlphaGo algorithm itself is perhaps the most well-known among them. In Part 3 of the final project, which is open ended, you may choose to implement AlphaGo.

- `heuristic_go_problems.py`: Contains an implementation of `GoProblem` and a *very* simple heuristic `GoProblemSimpleHeuristic`, which you can pair with your `MinimaxAgent` and `AlphaBetaAgent`.
- `game_runner.py`: Contains methods for running games between two agents, and tournaments among multiple agents.
- `go_gui.py`: Provides an implementation of `GameUI`, for visualization and user interaction.
- `go_utils.py`: Contains utility functions for setting up adversarial search problems.

7.2 Stencil Code

- `agents.py`: Where you will implement your agents. `GameAgent` defines an interface that your agents must implement. The core of the agent behavior is the `get_move(state, time_limit)` method, which returns an `Action` given the current game state and the remaining time on the clock.
- `supervised_learning.py`: where you will develop code to learn a value function.

8 Submission

Submit your assignment via Gradescope.
To submit through GitHub, follow this sequence of commands:

```
git add -A
git commit -m "commit message"
git push
```

Now, you are ready to upload your repo to Gradescope.
Tip: If you are having difficulties submitting through GitHub, you may submit by zipping up your hw folder.

8.1 Rubric and Grading

Task	Points	Details
Heuristic Search Agents	20	Points awarded for implementing the `get_move` function in `MinimaxAgent` and `AlphaBeta` agent
Feature Encoding	10	Points awarded for
Learning a Value Function	25	Points awarded for learning a value function that achieves high accuracy on the train and test set
Building an Agent with a learned heuristic	25	Points awarded based on whether the agent works (i.e., returns a move when a state is provided) and its performance.
README Questions	20	Points awarded for answering each README question