CSCI 0410/1411 Fall 2024
Final Project Part 2: Supervised Learning for Go

Milestones	Release Date	Due Date	Due Time
Part 1 (Search)	11/15	12/4	5:59 pm ET
Part 2 (Learning)	12/5	12/12	5:59 pm ET
Final Bot & Writeup		12/17	11:59 pm ET
Tournament Ends		12/19	11:59 pm ET

1 Introduction

In Part 1 of the final project, you implemented two tree search algorithms for the game of Go. Several of these methods (e.g., greedy, $\alpha\beta$-search, and iterative deepening) depended on a heuristic, and thus struggled to perform well with the naive Go heuristic we provided. In Part 2, you will use supervised learning to learn a better heuristic, and then hopefully demonstrate improved performance of your heuristic-based agents.

Part 2 of the project comprises three main tasks, followed by experimentation:

1. Build a state representation: i.e., develop a set of features that characterize the states of the game
2. Learn a value function: i.e., learn a function from (encoded) states to values
3. Learn a policy: i.e., learn a function from (encoded) states to actions
4. Experiments with your learned value functions and policies.

2 The Data

Supervised learning is the task of building a function approximator (i.e., a model) from labeled data. We provide two datasets for you to learn from, in the files `9x9_dataset.pkl` and `5x5_dataset.pkl`, which, as their names suggest, correspond to 9×9 and 5×5 games, respectively. Both datasets comprise a list of tuples of the form (state, action, outcome).

The 9×9 dataset is a collection of games played by humans (not AIs) on the Online Go Server (OGS).[1] The original collection of games can be found here. The processing code used to translate from the zip file to OpenSpiel data can be found in `data_processing.ipynb`, which may be useful to you if you want to learn from more data or filter out some of the data provided.

The 5×5 dataset comes from a different source. 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 outperformed the other agents we built for Part 1 of the project.

[1] We cannot attest to the quality of these training data, but we suspect that many of the players are stronger than our `MCTSAgent`. As a possible extension, you might try to filter this dataset for quality play.

3 Task 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.[2]

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.

[2]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).

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 and 3) and build agents based on these models (Task 4).

4 Task 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. 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.

Note You cannot evaluate the effectiveness of your learned value function until you build agents based on it (Task 4).

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.

5 Task 3: Learning a Policy

Your third task is to learn a policy, rather than a value function. In a sense, learning a policy is very similar to learning a value function. The primary difference is that it is a multiclass classification problem (i.e., to learn an action to take), instead of a binary one.

There are two basic approaches to multiclass classification. In the first, the goal is simply to predict the class directly (e.g., an integer between 1 and 26 in a 5×5 game of Go). In the second, you predict a probability distribution over classes, which you compare to a one-hot encoding of the true class.

The loss functions that correspond to these two approaches are called **sparse categorical cross-entropy loss** and **categorical cross-entropy loss**, respectively.

Task Your task is to implement a neural network that represents a policy and to train it using the data provided to choose an action given a(n encoded) state. Be sure that your training and test error decrease as your model learns.

Note You cannot evaluate the efficacy of your learned policy until you build agents based on it (Task 4).

6 Task 4: 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.[3] For now, we ask that you build two agents:

- The first should use your learned value function from Part 2 of the project in your heuristic-based search agents from Part 1, in place of the naive heuristic we provided in Part 1.
- The second should simply play according to your learned policy.

Support Code We have provided the `GoProblemLearnedHeuristic` class with a `heuristic` method as an alternative to `GoProblemSimpleHeuristic` from Part 1, which can be used by the heuristic search agents you built in Part 1.

Experiments Your final task is to design and run experiments with these agents. Among other things, you should hopefully be able to show that (at least some of) your Part 2 agents outperform your Part 1 agents. Summarize your experimental design in a table in your README, an explain your results, using figures or tables as necessary. For example, you might create a matrix with agent names as rows and columns, and agent scores in the cells.

7 Extensions

There are many possible ways to improve your supervised learning models, and to better use these models in your agents. Here are two simple examples.

- Learned models should lead to relatively strong opening moves, compared to, say, minimax, but can perform worse than minimax in end games, when there are only a few possible moves remaining and the search tree is relatively small. Your agent need not employ only one strategy; rather, it can use different search methods in different parts of the game.
- We provided you with a training dataset of 2,000 games of 5×5 Go. Rotating any of these games by 90, 180, or 270 degrees would still be a valid game of Go, but most likely one that is not in the dataset. This process is a means of augmenting the given dataset (**data augmentation**), by modifying data to produce additional data, without incurring the cost of running new games.

Consider enhancing your agents with one of these two techniques, or any others of your own conception, and updating your experiments accordingly.

8 Downloads

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

9 A Hint about Part 3

For Part 3 of the final project, we will encourage you to use all the tools in your AI toolkit to try to build even better Go agents. We hope you have fun!

[3] 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.