FAI | Spring 2025
Final Project - Report

B12705012 資管二 鄭博宇

Overview

In this project, I have tried two methods: **Deep Learning Model for Knowledge Distillation from Baseline**, and **Heuristic Algorithm assisted by Monte-Carlo Sampling**.

Firstly, I will explain the mechanism and results from **Deep Learning Model for Knowledge Distillation from Baseline**, and discuss why I believe this is not a suitable solution for Texas Hold'em.

Then, I will introduce my second method, which I proposed finally as my final agent, the **Heuristic Algorithm** with a lot of calculation from Monte-Carlo sampling.

Deep Learning Model

Progress

1. **A logging agent to record all instructions from baseline**

   In the uploaded source code, I include a file called `logging_agent.py`, which will directly import the baseline model and declare action decided from the baseline model. Let's call the baseline model we want to mimic as **Teacher model**.

   All the logged result is saved in a folder, containing json-format files for each round.

2. **Vectorization and Model Training**

   In the uploaded source code, there are three different files naming `train_model_XXX.py`, which are some different networks I have tried. All of them are using `pytorch` and a self-defined network by myself.

   I have tried the basic MLP network, RNN network, and an AWAC version (to include Monte-Carlo sampling in deep learning method). The traditional MLP network perform the best. (I will state the network, hyperparameter sets I have tried, and the result in next section.)

   For the RNN model, I include historical data of the same round as input, but the result is very bad. Although intuitively I believe there should be some **time-series connection** between some consecutive actions of the same round, it seems that **the teacher models (baseline we imported) are not considering this time-series effect**. All of the baseline models are just simply making decision based on the current situation.

   As for Monte-Carlo, the sampling procedure in training process takes too much time for me, so unfortunately I need to lower down the number of samples. As a result, this gives me a worse result compared to the standard MLP version. I think this is due to both **the lack of teacher model's data** and **the number of sampling in my training process**. If we want to have a good result in Monte-Carlo method, we might need significantly large data, as the scenarios in Texas Hold'em is too large...

3. **Agent using the trained model for decision**

   Finally, I use `decision_agent.py` as the gaming agent, to query the model we have trained previously.
   We input the current state into the model, and the result from our model is returned to the game engine (both Action and Value).

Training Attempts for MLP

The best model after days of trials is:

```
class SimplePolicyNet(nn.Module):
    def __init__(self, n_action: int = 3,
                 emb_dim: int = 16, hidden: int = 256):
        super().__init__()
        self.emb = nn.Embedding(53, emb_dim, padding_idx=PAD_IDX)
        in_dim = emb_dim * 7 + 4    # 2 hole + 5 board + num_feat
        self.mlp = nn.Sequential(
            nn.Linear(in_dim, hidden *4),
            nn.Dropout(0.3),
            nn.GELU(),
            nn.Linear(hidden *4, hidden*2),
            nn.Dropout(0.2),
            nn.GELU(),
            nn.Linear(hidden *2, hidden*2),
            nn.Dropout(0.0),
            nn.Sigmoid(),
            nn.Linear(hidden *2, hidden),
            nn.Dropout(0.0),
            nn.GELU(),
            nn.Linear(hidden, n_action)
        )
```
This training process is finished with **batch_size = 256**, **epochs = 50**, **AdamW Optimizer's lr = 3e-4**.

Some observations

1. For the activation function selection, using `GELU()` is significantly better than `Sigmoid()`, and is a little bit better than `ReLU()`.
2. At the beggining, I set dropout rate to be very high (about 0.8), and set a high epoch times simultaneously. In my previous experiences (from other courses), this should significantly help reducing overfitting problem and improve the overall performance. However, this doesn't happen here: a high dropout rate will result in a high loss value, and this value will not lower down during epochs (it reduce very fast at the beginning, then stop at there LOL). I found out that about ratio 0.3 is the best.
3. Adding hidden layers and hidden nodes in each layer helps increase performance in the game. But I stop at 5 layers in total considering the computer's performance.
4. I've also tried weightdecay technique, but it didn't help at all.

Some Inference and Result

From the attempts that I have tried above, I believe the overall problem need not avoid overfitting problem. In addition, I believe the dataset that I have logged for training (from the teacher model) is way not enough.

The problem is: I have tried my best to log the competition data, but the baseline is reacting too slowly! (or maybe this is deliberately designed, to ensure that we will not use this kind of knowledge distillation (like me) to mimic the final agent?)

At the end, my agent (declaring decisions from my deep learning model) can easily beat `baseline0` and `call_ai`. But surprisingly, it almost lose everything when fighting with the `random_ai` !!! (which shocked me a lot～～～)

I believe, this is because all of my learning is from the given baseline model, and they don't actually consider the case that their opponents are irrationable. But at least, I think **this proves that all of my training process make sense, and I am implementing the mimicking process correctly**.

After seeing this result, I decided to give up the deep learning method, and start to find out some way to beat other powerful baseline models.

Heuristic with Monte-Carlo

At first, I recall that there is a technique called **Monte-Carlo sampling**, which helps us to find out some not-too-bad solution without using a lot of computation resources. This actually fit our purpose: I need to declare my action within 5 seconds (it is changed to 10 seconds eventually).

Adding If-else Section

However, simply using Monte-Carlo as my heuristic algorithm to decide whether call/raise or give up (fold) is not possible. After my testing on the CSIE workstation, only about 2000 samples can be finished in 5 seconds. This is way lower than the actual scenarios (like about only 0.000001 of the whole cases in total). I am afraid of having some very-good or very-bad scenario being incorrectly declared, so I added some if-else sections to manually select these featured cases.

Originally, it can only beat the `random_ai()` (which I believed is the stupidest), and can't even beat `baseline0` and `baseline1`. After adding some if-else section as follows (to manually decide some extreme cases without actually running Monte-Carlo method), I successfully beat `baseline1` to `baseline4`!

The first version **using if-else for decision** is like the follows:

1. **Some Tricks**: I add a testing part before actually start computing my action.

   If the leading money has already existed the small_blind * round_left * 1.5, then the result is already determined. Hence, I can still win the whole game even if I fold every round. To avoid any further turbulence, I will declare fold on every round until the end of the game.

2. **Calculate the Win-rate** using `mc_equity` from `utils.monte_carlo_utils`.

   I wrote a Monte-Carlo utility function in another python script, for simplicity.

   In this function, I set the number of sampling `n = 20000` to finish the computation within the time limit.
   After computation, add 1 point for winning scenarios, and 0.2 point for ties.

   Some Improvement here!

   Considering limited computation, I **penalize on high-variance sampling results**. That is, I will also log the variance when doing sampling. Then, if I am not sure about the result, lower down the win-rate before return it to the agent.

3. **Some Heuristic Decision**

   If the winning rate is high, raise the value. If the winning rate is moderate, then we only allow call. If it is extremely low and the opponent raise, our agent will fold for this round.

   I splited the above concept into many subcases, and tuned the borders and parameters manually to find out the best set of condition statement. The result code is as follows:

   ```
   # Very simple heuristic
       win_rate = self._estimate_win_rate(hole_card, community_card)
       print("Win_rate", win_rate)

       if win_rate >= 0.85 and raise_act["amount"]["max"] > 0:
           target_amount = max(300, raise_act["amount"]["min"])
           print(f"AAA raise {target_amount}")
           return raise_act["action"], target_amount

       if win_rate >= 0.70 and raise_act["amount"]["max"] > 0:
           target_amount = max(100, raise_act["amount"]["min"])
           print(f"AAA raise {target_amount}")
           return raise_act["action"], target_amount

       call_amount = call_act["amount"]
       if win_rate > 0.6 and call_amount <= 100:
           print("Moderate win_rate > 0.65, calling up to 100: AAA", call_amount)
           return call_act["action"], call_amount
       elif win_rate > 0.4 and call_amount <= 50:
           print("Mild win_rate > 0.4, calling up to 50: AAA", call_amount)
           return call_act["action"], call_amount
       elif win_rate >= 0.25 and call_amount <= 10:
           print("Low win_rate > 0.25, calling up tp 10: AAA", call_amount)
           return call_act["action"], call_amount
       elif win_rate >= 0.1 and call_amount <= 5:
           print("Mini win_rate > 0.1, calling up tp 5: AAA", call_amount)
           return call_act["action"], call_amount
       elif call_amount == 0:
           return call_act["action"], call_amount
   ```

Further improvement against advanced baseline

I wrote another Python script to do some statistics on why I cannot beat baseline 5 to 7, and it shows that **the raise value acted by those advanced baselines are kind of insane**! Me being like a god (seeing both agents cards from `start_game.py`), I found out that baseline 6 and 7 tend to make a high-value raise or even all-in while they actually don't have such good cards. However, This smart strategies successfully beat my previous agent completely. My previous model consider the reaction based on the Monte-Carlo strategies (to evaluate my win rate), so it of course will fold due to fearness.

Hence, I add some **conservative feature** and **randomness**, as follows:

```
rank_threshold_1 = 10
    rand_threshold_2 = 12
    a, b = [int(card[1]) if card[1].isdigit() else MonteCarloPlayer._RANK_MAP[card[1]] for card in
hole_card]
```

```
    print(a, b)
    if a == b :
        print("SAME! raise 50 or min")
        target_amount = max(50, raise_act["amount"]["min"])
        if target_amount >= 150:
            print("BUT: too high, decide later")
        else:
            return raise_act["action"], target_amount
    if a >= rand_threshold_2 and b >= rand_threshold_2:
        print("BOTH Pass 2, raise 50 or min")
        target_amount = max(30, raise_act["amount"]["min"])
        if target_amount >= 150:
            print("BUT: too high, decide later")
        else:
            return raise_act["action"], target_amount
    if a >= rand_threshold_2 or b >= rand_threshold_2:
        print("One Pass 2, raise 20 or min")
        target_amount = max(20, raise_act["amount"]["min"])
        if target_amount >= 70:
            print("BUT: too high, decide later")
        else:
            return raise_act["action"], target_amount
    elif not (a + b >= rank_threshold_1):
        print("Hole cards too weak, folding:", a, b)
        return fold_act["action"], fold_act["amount"]
```

1. It is **Conservative**, since it folds at the beginning if the holding cards are too small. (But it is irrationable! We will lose small/big blind for the round.)
2. It is adding **Randomness**, since it raise some money if the holding cards look great! (Without considering hole cards and do Monte-Carlo sampling, this is actually not that reasonable. But on the other hand, since our INSANE opponent will also see the exactly same hole cards, maybe doing decision without looking at the common part is kind of make sense?)

As a result, though I can't provide a clear explanation, this **new version of AI agent perform better** and has the ability to beat those advanced baselines.

Some Reflect

Originally, when I start this project, I actually use some very naive approaches (just some stupid python script) to learn more about the given baselines (which I didn't mention the result in this report), and I believe all the baselines are very, very strong! That's the reason why I start doing deep-learning method in the first part, I believe I can mimic their strategies and train another powerful model. However, this approach failed.

After doing Monte-Carlo method, I found out that these baselines actually are not that powerful like I have thought. When testing the first four baselines, they can be easily beated by my heuristic algorithm, which simply do a large amount of computation and simulation. "Maybe this is the truth of gambling, just a mathematician's game", I thought at that time.

However, after continue testing baseline 6 and 7, my mind changes again. These advanced baselines' strategies can actually beat all of the rational agent that I have made, forcing me to build my agent also being not that reasonable. At the end, using the term from **Game Theory**, both agents are **signaling** and **lying** the other agent in every round.

Hence, gambling seems not to be that "calculatable" again?

The `agent.py` is the final proposed agent, **using heuristic algorithm assisted by Monte-Carlo sampling** (as in the last paragragh).

References

- All of the written report is made without generative AI.
- Codes are assisted by OpenAI GPT-o3, but all the ideas and coding structures are my efforts.