Q5

Motivation and Idea

Most competitive games usually have some best move after accounting for any factors of noise. RPS is an interesting game there is no best move, the goal is to find patterns of behaviour in the opponent that you can exploit while simultaneously trying to be as least exploitable yourself. This leads to a tradeoff between being completely random and deterministic, the former is completely unexploitable but cannot exploit, while the latter can exploit some strategies but if found out is very exploitable.

Since the strategy can only be discovered by looking at the combined history of yours and the opponents actions, we could try building a player-agnostic model that predicts probabilities of what the collective next choice of throws will be. Moreover, the data in RPS is really well-behaved and simple, in that it is just a sequence of tokens representing the three actions, so it is well-posed for our predictive model to use a Transformer encoder base for the most flexibility and to allow us to learn complicated and nonlocal patterns from the population of bots.

Predictor Architecture and Training

To ensure enough expressivity, I decided to use 3 attention layers with a hidden size of 64. To avoid the quadratic growth in compute as the history context increases, I decided that the last 100 moves was a large enough window to gauge the correct patterns of the playing agents, since most of them don't have strategies that look back that far precisely or perform actions with larger horizons.

If we let $a_0, a_1 \in \{0, 1, 2\}$ be the first and second player's move respectively, we can represent a collective action of two players as a number $s = 3a_0 + a_1 \in \{0, ..., 8\}$, which is basically the expansion of a two digit base-3 number. Since we don't always have a history of 100 moves, we can add a padding token 9, so our sequences will have a vocabulary of size 10 that we embed to our hidden size. Let $S = \{0, ..., 9\}^*$ be the class of finite sequences in this vocabulary, given an embedding $E : \{0, .., 9\} \to \mathbb{R}^{64}$, we can map each element in a sequence pointwise to a sequence of vectors $\mathbb{R}^{64*}$ +a positional embedding to prevent our transformer from treating it as a bag of tokens. Then we can add a manually trainable positional embedding to prevent our transformer $T : \mathbb{R}^{64*} \to \mathbb{R}^{64*}$ from treating the input as a bag of words. We then take the last latent $l_{100} = T(E[s_1, ..., s_{100}])_{100}$ and use a linear predictor $L : \mathbb{R}^{64} \to \mathbb{R}^9$ giving us logits $L(l)$ predicting one of the 9 possible outcomes.

I initially thought maybe having another signal from the other latents that tried to predict the agent IDs based on the window would be helpful, so I used the first two latents l_1 and l_2 in the encoding sequence and a single linear ID layer $L_{ID} : \mathbb{R}^{64} \to \mathbb{R}^{43}$ that mapped to 43 logits representing the pre-softmax probabilities of the first $(L_{ID}(l_1))$ and second player $(L_{ID}(l_2))$ being one of the 43 agents respectively. I thought this would help build a profile that could be used as extra useful information along with the regular predictions.

My final predictor model would output the ID latents and the prediction latents as well as the logits after applying the linear transform, in case either the latents or the probabilities would be needed, the former for having a more expressive representation past just the probabilities.

For the first phase of training I first collected 1 episode long games between the first 20 provided agents among themselves, for each pair of IDs in $\{0, ..., 19\}^2$, for a total of 400 games as a baseline offline training set. This was before I had the idea of adding the ID predictions, so these just had next token prediction labels on their rolling windows. Then for every bot, I selected a few probes out of randbot, rockbot, rotatebot, freqbot2, iocainebot and markov5 (usually the first two) to generate a few signature runs for each bot, in this case I did add the ID labels. Afterwards, I also mirrored the dataset, so for every trajectory there was another trajectory with identical moves just with the player order being flipped.

For the training loop, I initially had three different loss contributions, one for the prediction cross-

entropy loss, another for the ID loss, and a symmetry loss, which computed the square mean difference between the logits of the regular and swapped input. At first I started with the symmetry loss scaled by 0.05 and an unscaled ID loss, but I noticed that it did not really contribute much because most of the data had no ID labels. I decided to add dynamic scaling based on what percent of the batch had labelled IDs, so it would provide a large contribution if only a few had ID labels to maintain a consistent signal. I had to scale the base loss down by 0.25 to prevent blowup. I used the standard Adam optimizer for all the training in this assignment.

Using this regime, I managed to reach around $1.65 - 1.7$ loss, which suggested some pattern extraction, but after a bit of testing and removing the ID and symmetry loss altogether, I noticed that they had significantly held back the improvement the model could've made, and I was able to go down to 1.5 CE., at the cost of the ID loss signal becoming degenerate and unusable (for now). This could've been a result of overfitting, but I decided that this problem doesn't really need you to know exactly who the opponent is, just a good gauge of their behavioural patterns is enough.

I initially had kept a low batch size of 512 (where each sample is a window of tokens length 100), which had decent gains, but the noisiness required lower learning rates around 10^{-4}-10^{-5} and made it too slow (at least 50 epochs were needed to reach 1.66 around). Once I removed the other loss terms and increased the batch size to 4096 and the learning rate to 10^{-3}, I started seeing much larger drops. I gauged the model every 200 batches and manually adjusted the learning rate when I needed to, eventually dropping both the batch size to 2048 and the learning rate back down a magnitude to find flatter minima.

Policy Architecture and Training

Once I had a decent predictor model, my initial idea for an agent policy was to do a best response based on what probabilities were predicted. That is, given I have a distribution $P(A_0, A_1)$ over player 0 and player 1's choices given the past history, put myself in player 0's spot and calculate the expected payoff for me playing $A_0 = A_0$:

$$ER(A_0) = \sum_{a_1 \in \{0,1,2\}} P(a_1|A_0)R(A_0, a_1)$$

using the player 0 payoff matrix of RPS for a particular action. Then, I would sample actions w.r.t. $\text{softmax}(ER(0), ER(1), ER(2))$, so higher return actions would be more likely to be picked. I did this because the deterministic argmax choice ended up being very exploitable and was taken advantage of during online evaluation against the other agents. This improvement made it consistently outperform a good portion of the bots without any online learning, but a few complicated ones matched it or beat it consistently, particularly the ones that switched strategies and the Markov models, as well as greenberg and iocainebot.

Separately, I also had the idea of the softmax temperature that is a MLP that depends on the difference between the players' ID probabilities, so we could try indirectly coding cautious behaviour if the opponent looks like one of the stronger bots or a bot that normally beats me easily. To make this more robust, I added a module that would keep a moving average of the belief probabilities for both sides, and then a temperature head that would take in the difference and apply a two layer fully connected network to generate an inverse temperature, which was clamped betweeen 0.1 and 5 to prevent any numerical issues or dominating gradients. This was before I discarded the idea of having the predictions be IDs, but I kept it in because it could be a useful additional way to change the model's behaviour, even if the interpretation is less clear.

I also noted that the payoff matrix only represents the completely uninformed belief of the RPS rewards. Since you don't have any information about the enemy you only would know how good a move is because it wins or loses. In reality, our goal isn't to just win once, so playing a locally non-winning move against a particular agent might actually be better if we can manipulate their behaviour to benefit us long term. I added a dynamic payoff contribution term that took in all the extra latents in the output of our predictor and computed a 2 layer FFN with hidden size 128 and output logit size 9, which is then reshaped into a 3×3 matrix, normalized by tanh and scaled by 0.5 before being added to the static

payoff, so that we don't need to relearn the basic good and bad combinations.

To complete the full training of our agent, I decided to use a manual REINFORCE algorithm with a discount of 0.97. Note that I chose this so that the reward signal from the end of a window is around 1/10 of the closer wins, so it isn't as noisy as 0.99, but not as reductive as lower discounts. The loop would collect a certain number of episodes from a randomly sampled batch of bots (between 10 and 30), and then compute the predictor outputs, giving us a collection of trajectories our policy trains on each round. The policy networks started with a learning rate of 10^{-3} which I gradually brought down to about 10^{-5}. Initially I kept the base predictor frozen, but only got marginal gains, so I introduced a custom optimizer for its parameter with a learning rate much lower at around 10^{-6} so that it would not catastrophically forget past knowledge from the offline dataset. Because the number of training gradients would increase from this, I had to drop the number of episodes to 5 per round.

To gauge the strength of the bots against my agent, I kept a strength parameter for each bot starting at 0, and whenever they played against my agent and won, a moving average with shift of 0.01 would add the negative of my return to it, so if I lost more it would make it more positive and otherwise make it more negative. The sampling probability of each bot was then proportional to the exponential of their strength, so that harder bots would be sampled more frequently during the loop. To keep track of any improvements or changes, I kept a moving average of the returns over the round and printed the strongest bot after that round.

I ran about 1000 rounds of the online training at varying learning rates and batch sizes, and I noticed the strength on simple bots dropped consistently, with small average improvements on the good bots which were primarily mixed_strategy, robertot, foxtrotbot, markov5, markovbails, russrocker4, shofar, peterbot, phasenbott, sunNervebot, greenberg, and iocainebot. The basically random bots like pibot, randbot, debruijn81, and textbot all had wildly oscillating strengths, and would make it more difficult to find patterns where it mattered, so I removed them from the sampling population and increased the probability of the hard bots, focusing especially on greenberg, iocainebot, markovbails, phasenbott, and markov5

After about another 1500 rounds, I got a small edge on greenberg, iocainebot and markov5, so I whittled down the special cases to phasenbott and markovbails. I manged to bring markovbails to the same range as the other strong ones, but phasenbott was much more volatile, sometimes losing a lot and sometimes winning a lot, so I gave up on trying to keep its EMA returns negative.

Running the agent

In the provided MyAgent21217394 folder, I have a .py file containing the necessary classes for the different architecture components as well as an unloaded agent class. My agent wraps this unloaded agent and takes the weights for the predictor and policy models from the provided pred_weights.pt and policy_weights.pt files in the same folder during initialization. If running on a different device other than cpu, modify the agent's device variable when initializing it.