12 SUPPLEMENT ABOUT THE AGENT

12.1 More about the probabilistic policy

Our goal with the probabilistic policy was to prevent games from being deterministic. In particular, during training, the probabilistic property can be used for exploration, but in many applications after training would be replaced with a regular `arg_max` operator. However, doing this in our setting would mean that once the user chose a pair of agents, every game between them would play out the same. To avoid this, we trained our agents using a `Gumbel_softmax`[13] with a temperature parameter, which does not change the *ordering* of the perceived action quality—just the sampling probabilities. We scheduled the temperature to start at 20 and end at 0.1—and continued to use the same 0.1 temperature after training. Our team chose the temperature ranges empirically, running softmax tests directly on hypothetical scoring output instead of using the neural network. The criterion for which we were looking was a sparse action probability matrix, where just a few actions received all the probability mass. To illustrate the importance of sparsity, suppose 35x poor actions have 2% action probability while a single excellent action has all the rest of the probability mass; that would only be 30%!

12.2 Achieving a more optimal agent

Our goal was not to solve the domain, but to get a sufficiently strong agent that we could mutate that agent and see if participants could detect the change. Thus, we employed a simplified version of the architecture outlined in AlphaZero [87], but modified for explanation generation. While our agent is clearly suboptimal, it begs the question of whether its poor performance is the fault of the architecture, the training process, our simplifications, etc. A CNN model should be able to perfectly solve this domain, since the game structure is highly local, grid based, and fully described by a single game snapshot. The architecture of our CNN should have sufficient capacity.

Probably the weakest link in the current structure is the target formation. Currently, we use a procedure best described as "Pure Monte Carlo game search" (PMCGS, described by Russel and Norvig [78] in Chapter 5): For each square on the current board, play random games to the end while recording results. Pack the results into a tensor of outcome probabilities (e.g. if it wins 7 of the 10 games, 0.7 would be the win outcome target value). One simple improvement would be to replace the *random games* with games played by having both players *follow the agent's policy*. However, the downside of policy rollouts is that training takes longer. In fact, the capability to do so is currently in our source, but commented out. One of the downstream consequences of using random rollouts is that the agent does not defend very well. This is because if the agent exposes itself to a kill shot, PMCGS targets can underestimate the threat because both players are treated as random, and thus *unlikely* to take the winning square.

The other obvious weak link is the loss function. We attempted to use the simple, off-the-shelf components where possible, and so did not define our own loss function, but it should be possible to improve upon `L1Loss`. In particular, when using L1Loss, it is just as important to accurately estimate an illegal move, a coin flip move, and the winning move; but it is actually far more important to get the last one right. This is not desirable, because e.g., large errors for terrible moves are tolerable without harming the action selection. Conversely, accurate predictions for the best actions seem much more important than the others if the action selection is to pick them. To do so, one could define a custom loss function assigning weight as an increasing function of win probabilities in the *target* tensor.

[13] https://pytorch.org/docs/stable/generated/torch.nn.functional.gumbel_softmax.html#torch.nn.functional.gumbel_softmax

The last idea that bears mention is to make the agent model-based. Currently, our agent does not get the game model or learn the game model, and so it is essentially model-free. Performing MCTS on the search tree from a model-based agent would likely improve performance, and might be necessary for extending this work into some other domains—but should not be necessary to optimally solve MNK games. However, explaining with this kind of search tree (which explicitly encodes an overwhelming amount of information about the sequential environment) remains an interesting and sparsely explored research area.

To summarize the ideas we suggest trying:

- Use a transformer
- Add an LSTM
- Train longer
- Improve the target formation
- Define a better loss function
- Use a model-based agent