Speaking in Chess: Training Large Language Models to Play Chess

Evan Frick, Tianle Li, Kayla Lee[1]

Abstract

In this paper, we delve into the uncharted territory of chess analytics with Large Language Models (LLMs), transforming the game's representation from static positions to a sequence-to-sequence problem. Leveraging a custom chess move tokenizer, our GPT-2-Small model intakes a series of chess moves and output the next optimal move by learning exclusively from actions and sparsely distributed rewards of a game's outcome. After extensive behavior cloning on over 20 milion high-ELO chess games, this behavior cloning strategy autoregressively generates full-length chess games with reasonable accuracy.

To further enhance the model, we evaluated 6 Reinforcement Learning (RL) strategies, including 3 novel algorithms, Fictitious Self-Play with Short-Term Adversaries, Past-Present Q-Iteration with a Pseudo-Ensemble, and Self-Play with Funnel Searching.

First, we attempted self-play through Policy Gradient and Q-Iteration methods. Although elegant, we found that pure self-play with a single model was too unstable for developing a reliable policy, as the training often deviated towards draws with low precision game-play.

To improve training stability, we implemented Fictitious Self-Play with Past-Present Q-Iteration, yielding a dependable policy. Notably, extensive training iterations demonstrated that the present policy consistently outperformed the past, indicating optimized reward capture. To ensure the current model's superiority was not limited to an adversarial role, we introduced an LLM-based-pseudo-ensemble algorithm. By assigning models different chess openings at each training step, we generated diverse policy conditionals. However, after numerous iterations, the present policy struggled to consistently outpace the past model within this pseudo-ensemble framework, underscoring the complexity of dynamic opening play.

Acknowledging the learning challenges associated with ensemble-based algorithms, we introduced Fictitious Self-Play with Short-Term Adversaries. This algorithm involves training a base model as an adversary, which is subsequently employed to refine the current model's defenses. The adversary is reset each iteration to prevent over-specialization and the defending policy is given few training steps to prevent reverse adversarial strategies. We find that Fictitious Self-Play with Short-Term Adversaries is comparable to Past-Present Q-Iteration. Notably, both the adversary policy and the actual policy trained yield strong and improved policies, even though the adversary policy is reset to the base model after each training iteration.

Observing the training inefficiencies and slow inference speed, we attempted a simple offline training scheme where we employ trajectory loss. We find loss converges, but the policy extensively plays out of distribution moves at test-time.

Finally, our evaluations against the chess engine Stockfish revealed early-game proficiency but mid-game vulnerabilities leading to suboptimal late-game play. We developed Self-Play with Funnel Searching, a strategic exploration method that selectively expands potential moves based on their trajectory probabilities. Funnel Search samples k moves at a given state and pick top m moves from all sampled moves using a customizable k and m scheduler, resulting in "funnel" shaped trajectories trees.

Ultimately, we find that pure single-model self-play is too unstable, even with improve game sampling methods like funnel search. We also find that simple offline methods are not sufficient and are unable to punish out of distribution strategies. We find the dual-model methods protect against divergence, and promote better policies. Past-Present Q-iteration and Fictitious Self-Play with Short-Term Adversaries are able to produce policies better than the behavior cloned base model. Additionally, we find all our implicitly Q-Value altering methods do not yield useful Q-Values after RL training, which prevents the use of true dynamic programming based algorithms.

We conclude that the sparse-reward action history inferred state learning problem is exceedingly difficult. Reward signals are noisy as they are passed down to actions earlier in the trajectory. Moreover, the state space exponentially increases for each action. As such, training successful policies in this spaced is difficult, and requires myriad methods to overcome the inherent instability of the learning problem. We hope that the novel algorithms present can be applicable to more than just chess, and can contribute to a wider range of sparse reward sequence problems.

Our code is publically available at: https://github.com/kaylaclee68/Speaking_in_Chess

1. Preliminaries

1.1. Chess Terms

1.1.1. Move Notation

In algebraic notation, each square is denoted by a coordinate pair with letters 'a-h' indicating columns and numbers '1-8' indicating rows. Moves are represented by the moving piece's initial (excluding pawns) followed by the destination square. Captures add an 'x' before the destination square, checks are noted by '+', and checkmate by ''. Special moves include castling ('O-O' for kingside, 'O-O-O' for queenside) and pawn promotion (e.g., 'e8=Q').

Examples:

- **Nc8-d6**: Knight moves from c8 to d6.
- **Nc8xd6**: Knight captures on d6.
- **Nc8-d6+**: Knight moves to d6, checking the opponent's king.
- **Nc8-d6#**: Knight moves to d6, delivering checkmate.
- **Ra8xb8+**: Rook captures on b8 and checks.

1.1.2. ELO

ELO is a measurement of player strength. Grandmaster players play around 2500 to 2900. The maximum Stockfish ELO is 3500.

1.1.3. Evaluation

The evaluation score for a given state on the board is calculated by a max strength engine and either is Centipawns or Mate in X. One pawn advantage is equal to 100 Centipawns. Mate in X means there is a force checkmate in X moves. Centipawns can be converted in a rough estimate of win probability by dividing by 100 and appying the sigmoid function. All references to "eval" in this paper mean $\text{sigmoid}(\frac{\text{centipawn score}}{100})$.

2. Introduction

The game of chess has always been a key area of focus in artificial intelligence research due to its strategic depth and complexity. Recently, language models (LMs) have shown exceptional abilities in text understanding and generation. The goal is to demonstrate how language models can be effectively applied in chess by predicting moves and formulating strategies through reinforcement learning algorithms. However, chess inherently involves sparse rewards due to the large number of possibilities in any given state, making it difficult to learn effectively using pure reinforcement learning. We will also present three novel reinforcement learning algorithms to combat the sparse nature of chess games, Adversarial Self-Play, Pseudo-Ensemble Self-Play, Self-Play with Funnel Searching, and compare our results across many reinforcement learning schemes.

[1]University of California, Berkeley. Correspondence to: Evan Frick, Tianle Li, Kayla Lee < evanfrick@berkeley.edu, tianleli@berkeley.edu, kaylalee@berkeley.edu>.

2.1. Problem Statement

In this paper, we aim to train an LLM agent through RL to play in a sparse reward and low observation environment. Rewards are received at the end of the game, where the winning player receives reward 1, and the losing player receives reward 0, tying results in 0 for both players. We employed an n-gram strategy in a reinforcement learning context to predict the next action a_{t+1} (outputs) based on the history of previous actions (inputs) $\vec{a}_{0:t}$. Theoretically, the true state s can be reconstructed entirely without loss from $\vec{a}_{0:t}$. As such, our policy becomes $\pi(a_{t+1}|\vec{a}_{0:t})$. We can then autoregressively generate a full game by appending the a_{t+1} on to $\vec{a}_{0:t}$ to create next "state" $\vec{a}_{0:t+1}$. This is possible because chess transition probabilities are known and deterministic when viewed from the perspective of both players. We should consider that this problem formulation in considerably more difficult than traditional approaches which provide the full state at every action. We opted to instead align with how LLMs autoregressively build trajectories.

Considering that LLMs produce their token probabilities through a softmax over a logit layer, we can consider an alternate view of the policy as

$$\arg\max_{a_{t+1}}[Q_\theta(a_{t+1}, \vec{a}_{0:t})] \tag{1}$$

or as

$$\text{soft}\max_{a_{t+1}}[Q_\theta(a_{t+1}, \vec{a}_{0:t})] \tag{2}$$

for a less deterministic policy. We also note that because of the deterministic nature of state-action transitions, and the theoretical existence of an optimal line of play:

$$Q^*(a_{t+1}, \vec{a}_{0:t}) = V^*(\vec{a}_{0:t+1}) \tag{3}$$

It is also unexpectedly true that if there exists an algorithm that solves chess:

$$Q^*(a_t, \vec{a}_{0:t-1}) = Q^*(a_t, \vec{a}_{0:t}) \text{ for all } t. \tag{4}$$

Essentially, an ideal policy would be able to give a Q-Value of 1 for a winning move and a Q-Value of 0 for a losing move.

3. Related Work

3.1. Chess and LLMs

In the realm of chess, language models like GPT-2 exhibit remarkable capabilities. These models delve deep into the intricate dependencies within the vast decision space of the game. Their strength lies in their ability to model conditional probabilities, allowing policies to dynamically adapt based on the specific chess opening being played (Noever et al., 2020). For instance, the probability distribution $p(\text{output}|\text{opening 1})$ is distinct from $p(\text{output}|\text{opening 2})$, illustrating the model's agility in tailoring its strategy to the initial game conditions. This adaptability mirrors the tactical acumen of expert chess players who deftly adjust their plans in response to their opponent's opening moves. These mechanisms enable the model to transcend a static single policy and, instead, cultivate a versatile array of dynamic strategies (Noever et al., 2020).

3.2. Q-Learning with BCE Loss

Q-Learning aims to learn the value of actions without explicitly modeling the environment's dynamics. In this context, Binary Cross-Entropy (BCE) Loss can be interpreted as a way to measure the discrepancy between the predicted Q-values, seen as probabilities of success, and the binary outcomes of actions (success or failure) (Zhang and Sabuncu, 2018).

The advantages of using BCE loss for Q-learning are manifold:

- **Direct Probability Model:** BCE Loss directly models the Q-values as probabilities, which aligns well with the need to predict the likelihood of successful actions.
- **Gradients:** The gradients resulting from BCE Loss are generally well-behaved, providing stable updates during training, which is important for the convergence of Q-learning algorithms.
- **Compatibility with Binary Outcomes:** Many Q-learning scenarios involve binary outcomes (e.g., win/loss), making BCE Loss a natural fit for such problem statements.

3.3. Self-Play (SP)

Self-Play (SP) involves an agent playing against itself during training. This approach establishes a continuous learning process where the agent's skill level aligns with that of its opponent. However, Self-Play can be susceptible to instability due to noisy loss signals, leading to a scenario where the agent might forget how to counter strategies it previously encountered. Idealistically, self-play would iteratively increase the ability of the agent(s) involve, but in an environment with imperfect play, agents may instead diverge, since if one agent gets worse, the other agent plays against a worse opponent, and may also get worse, spiraling out of control. The following algorithms seek to mitigate this problem.(Laterre et al., 2018)

3.4. Fictitious Self-Play (FSP)

Fictitious Self-Play involves training against a uniform mix of past versions, which helps prevent cyclical forgetfulness but can sometimes lead to inefficient learning due to numerous interactions with weaker, outdated opponent strategies.(Heinrich and Silver, 2016)

3.5. Prioritized Fictitious Self-Play (PFSP)

Prioritized Fictitious Self-Play enhances Fictitious Self-Play by matching the current agent (Agent A) with a selected opponent (Agent B) from a pool of past versions (Candidates C), based on a calculated probability that optimizes the learning potential from each match. (Heinrich and Silver, 2016)

3.6. Adversarial Self-Play

In the context of robotic manipulation, two agents operate with distinct objectives: the primary agent is tasked with efficiently executing given tasks with known parameters, akin to achieving victory in a chess game. Meanwhile, the second agent is charged with exploring new and unexplored scenarios. According to (Plappert et al., 2021), this setup emphasizes the importance of robot's adaptability and its ability to innovate beyond preprogrammed instructions, especially when dealing with complex or unpredictable environments (Plappert et al., 2021). The dual objective loss function used in this setting reflects the balance between proficient task execution and the continuous exploration of new robotic manipulation strategies.

$$\mathcal{L}_{\text{ASP}} = \alpha \cdot \mathcal{L}_{\text{win}} + (1 - \alpha) \cdot \mathcal{L}_{\text{exploration}}$$

α is a weighting factor that determines the balance between task proficiency ($\mathcal{L}_{\text{win}}$) and strategic exploration ($\mathcal{L}_{\text{exploration}}$) (Plappert et al., 2021). The task proficiency aspect quantifies the primary agent's effectiveness in completing its designated tasks, while the strategy development component evaluates the adversarial agent's ability to explore new approaches to decision-making. We later explore how a tangential formulation of Adversarial Self-Play can be applicable to chess.

We note that Wang et al. (2023) found adversarial policies were able to beat superhuman GO AI's, especially run without search, despite being poor overall policies. We reason

that non-search AI's in move-based games like chess and GO may suffer from this lack of robustness. Exploiting the simplicity of training adversaries against game-playing RL agents may be a key to new algorithms.

3.7. RL for Chess

- **MCTS vs. Chess Approach:** Unlike MCTS, which builds a tree structure, our Chess approach utilizes a transformer-based model to predict optimal game trajectories for chess, enhancing win chances for both sides. Instead of selectively exploring nodes and simulating outcomes like MCTS, we induce a temperature parameter for exploring multiple trajectories, mimicking Monte-Carlo Search.

Both AlphaGo and our chess model utilize a combination of self-play, policy gradient methods, and bootstrap iterations in their training processes.

4. Methods

4.1. Model

We opt to use a customized version OpenAI's GPT-2, a decoder-only model (Radford et al., 2019). We use the standard GPT-2-small architecture, with 12 layers, 12 attention heads, 768 embedding dimensions. In order to have each token represent a chess move, we use a custom tokenizer. To create the tokenizer, we enumerate every possible Long Algebraic Notation (LAN) move, and 2 turn marking special tokens, totaling 23253 tokens. These tokens are sufficient to express every possible legal chess game in PGN notation. This reduces the model vocabulary, thus requiring full pretraining of GPT-2 from scratch. The custom tokenizer is better for two reasons: **(1)** The action trajectory becomes clearly defined where each non-special token is an action. **(2)** During generation, we can mask out tokens corresponding to illegal moves. This is essential for later steps.

4.2. Behavior Cloning

In order to initialize to a policy that generated reasonable moves instead of completely random actions, we do supervised behavior cloning by pretraining the model via next token prediction on 2300+ ELO games. We find behavior cloning to be extremely successful in this use case. While generation of illegal moves is still possible, we find that once illegal moves are masked, the model is able to play with excellent accuracy.

4.3. RL Loss Choices

There are many choices on how the win/loss signal should be propagated back to moves during a game. Namely, the loss strategies arise from two different interpretations of the problem: 1. The policy gradient interpretation; 2. The Q-iteration interpretation. The first interpretation views our policy as just producing a probabilities over moves for each step. Normally the loss is as follows:

$$L(\theta) = -\mathbb{E}_\tau[R(\tau)log(P(\tau))] \tag{5}$$

However (1) is not sufficient for our case, since we have 2 rewards per trajectory. Thus it should be formulated as follows:

Assume

$$\begin{aligned} &R(\tau_{win}) = 1 \\ &R(\tau_{lose}) = -1 \\ &P(\tau) = P(\tau_{win})P(\tau_{lose}) \end{aligned} \tag{6}$$

Then we have

$$\begin{aligned} L(\theta) &= -\mathbb{E}_\tau[R(\tau_{win})log(P(\tau_{win})) + R(\tau_{lose})log(P(\tau_{lose}))] \\ &= -\mathbb{E}_\tau[log(P(\tau_{win})) - log(P(\tau_{lose}))] \\ &= -\mathbb{E}_\tau[log(\frac{P(\tau_{win})}{P(\tau_{lose})})] \end{aligned} \tag{7}$$

However, policy gradient requires a τ_{win} and τ_{lose} which is not always possible due to drawing. It is unclear how to discourage drawing with this loss scheme.

We instead turn to the Q-iteration perspective where actions are draw from the distribution generated by $\underset{a_{t+1}}{\text{soft}\max}[Q_\theta(a_{t+1}, \vec{a}_{0:t})]$. We then assert that $Q_\theta(a_{t+1}, \vec{a}_{0:t}) = V_\theta(\vec{a}_{0:t+1}) = R(\tau_{result})$ (if the policy was optimal). We can then run supervised loss on the Q-value as follows: $\ell(Q_\theta(a_{t+1}, \vec{a}_{0:t}), R(\tau_{result}))$. However, we do not have explicit access to the Q-values (model logits) because they are not designed to be actual Q-values when behavior cloning, just proportional to them. Thus we run loss on the softmax over the Q-values instead. Intuitively, winning moves should push the Q-value up, and losing moves should push the Q-value down. Thus the actual loss used is $\ell(\text{soft}\max(Q_\theta(a_{t+1}, \vec{a}_{0:t})), R(\tau_{result}))$.

There is a large bias with this method. Consider the following: a chess game is played near optimally, but is lost due to a single mistake mid-late game. The above methods reasons that *every* Q-Value should be pushed down, even though only one move was the problem. We can then adopt a trajectory-level loss, that suggests that the $P(\tau_{win})$ should be 1 and $P(\tau_{lose})$ should be 0, but individually the probabilities are less defined. Notably, this implies that if a game is

lost, only one or more moves must have been wrong. Similar to the previous method, if the game was won, all moves must have been good (a flawed, but unavoidable assumption). We then arrive at the following trajectory level loss equation for some τ_{result} defined by moves $\vec{a}$:

$$\ell(\prod_{t=0;\text{step }2}^{T} \text{soft}\max(Q_\theta(a_{t+1}, \vec{a}_{0:t})), R(\tau_{result})) \quad (8)$$

Since these Q-Value losses have targets between 0 and 1, we use binary cross entropy loss as ℓ and we define τ_s as trajectory from one of the sides.

$$\ell(\tau_s) = -R(\tau_s)\log(p(\tau_s)) - (1 - R(\tau_s))\log(1 - p(\tau_s)) \quad (9)$$

The non-trajectory version is (for one side):

$$\ell(\tau_s) = \sum_{t=0;2}^{T} -R(\tau_s)\log(p(a_t)) - (1 - R(\tau_s))\log(1 - p(a_t)) \quad (10)$$

5. Experiments

5.1. Direct Self-Play

5.1.1. Description

The simplest self-play formulation is as follows:

Algorithm 1 Basic Self-Play

while True **do**
  $\vec{a} \leftarrow$ `[]`
  while game is not over **do**
    $a_{next} \leftarrow \pi_\theta(a_{next}|\vec{a})$
    $\vec{a} \leftarrow$ `concat`$(\vec{a}, a_{next})$
  end while
  $\tau_{win} \leftarrow p(\vec{a}$[`win moves`]$)$
  $\tau_{lose} \leftarrow p(\vec{a}$[`lose moves`]$)$
  loss $\leftarrow \ell(\tau_{win}|win) + \ell(\tau_{lose}|lose)$
  `backward`(loss)

In this case ℓ can adopt several different strategies, see the section on losses for more information.

Self-play is conceptually simple. A model autoregressively generates a game trajectory. We increase the probability of moves that were played by the winning "agent" and we decrease the probability of moves that were played by the losing agent.

Unfortunately, this strategy is also extremely unstable. Consider some move a_t. a_t receive a win/loss reward signal r generated at the last time step of the game T telling it to increase/decrease in probability. However, the reward r is noisy because $\vec{a}_{t'>t}$ is unlikely to be optimal, meaning winning or losing does not actually give an accurate signal on if a_t was a good or bad move. While this variance may be reduced by collecting infinite trajectories, this is completely unrealistic.

We implement self-play with both a policy gradient style loss and a binary cross entropy loss. For more information on losses, see above.

5.1.2. Results

We find that self-play is too unstable to train good policies, and diverges quickly with both the Q-iteration problem formation as well as the policy gradient problem formation.

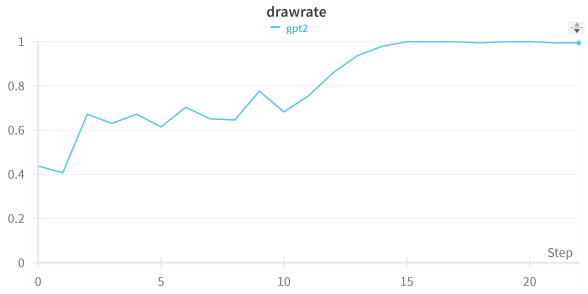

Figure 1. Self-play draw rate approaches 1 very quickly.

We can detect divergence with draw rate. As the policy diverge, it begins drawing every game as moves become random or repetitive. Empirical analysis of the games played shown nonsensical results. This occurred no matter how the objective was formulated, even when heavily penalizing draws.

5.2. Past-Present Q-Iteration

5.2.1. DESCRIPTION

To try and solve the divergence problem, we implemented a simplified version of PFSP, giving probability 1 priority to the most recent past version. This is done due to compute constraints such that maintaining many past versions would be impossible. We maintained two models, the previous self and current self. The current self was trained for 15 interactions, with 64 games per iteration, then the previous self was updated to the current self. This stabilizes the learning, ensuring that the new policy is better than the previous policy. Care for selection k is needed. As $k \to \inf$ the system becomes more and more stable, but π_θ becomes more and more adversarial to π_{past}, which is not necessarily a good overall policy. We use trajectory loss when training.

Algorithm 2 Past-Present Q-Iteration

$\pi_{past} \leftarrow$ `pretrained agent`
$\pi_\theta \leftarrow$ `pretrained agent`
for each training iteration **do**
  `freeze`(π_{past})
  Train π_θ against *opponent* π_{past} for k iterations
  $\pi_{past} \leftarrow \pi_\theta$
end for

5.2.2. RESULTS

We found that Q-iteration trains a reasonable policy, see 12 for more details. Empirical evaluation showed some strange adversarial-like behavior when generating game trajectories at inference time, but evaluation against Stockfish shows otherwise.

We can confirm that the present policy is training well against the previous policy by setting the k, the past-present update period, to infinity.

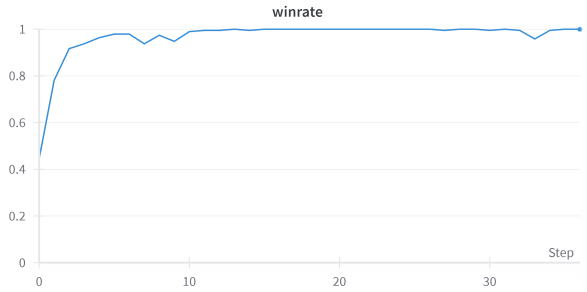

Figure 2. Present win rate over Past as k increases. Win rate approaches the maximum of 1

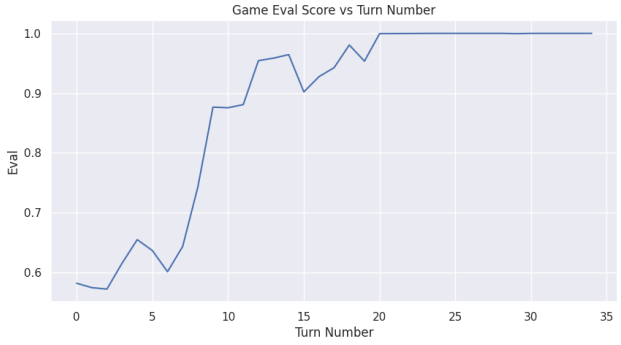

Figure 3. Eval from a winning game against Stockfish at 1500 ELO

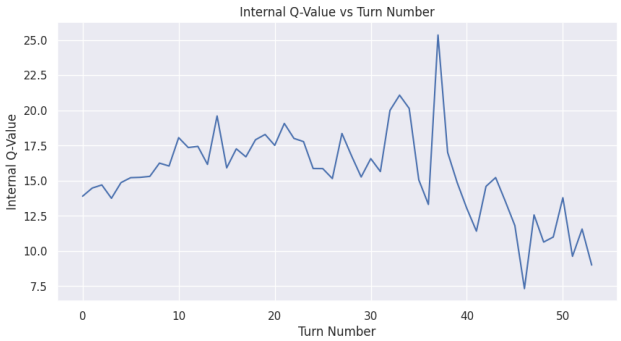

Figure 4. Internal Q-Values from the winning game from Figure 3

We notice that despite the training, the Q-Values do not provide much meaning. The above two figures show the same game, we notice that as the game increase in the policy's favor, the Q-Values do not show a meaningful increase.

5.3. FSP With LLMs as Pseudo-Ensembles

5.3.1. DESCRIPTION

We recognize that only using the current past self is susceptible to cyclical robustness issues. As such, we propose FSP With LLMs as Pseudo-Ensembles, which utilizes LLMs exceptional ability to represent unique conditional distributions along with the extremely divergent nature of chess trajectories. In this algorithm, we use the policies conditioned on a chess opening to create a pseudo-ensemble, thereby forcing the current policy to robustly learn to beat the past policy. By using a diverse array of openings, the current agent cannot simply learn to exploit an adversarial trajectory on the past agent.

Algorithm 3 Pseudo-Ensemble Fictitious Self-Play

$\pi_{past} \leftarrow$ `pretrained agent`
$\pi_\theta \leftarrow$ `pretrained agent`
for each training iteration **do**
  `sample` $O_i \sim \mathrm{U}(O)$
  Train $\pi_{past}(\cdot|O_i)$ against $\pi_{past}(\cdot|O_i)$
end for

We note that intuitively $\pi(\cdot|O_i)$ and $\pi(\cdot|O_j)$ for $i \neq j$ yields very different state and action distributions, thereby forcing the present policy to learn a good overall policy to show improvement. We use an opening book of 480,000 openings (8 to 24 moves, can be subsets) and randomly select and opening each time.

5.3.2. RESULTS

We find that the present policy is unable to improve at all even as k is very large. We consider several reasons for this behavior:

1. The objective loss lacks the information quality needed to improve.
2. The openings are too out of distribution for the policies to play reasonable games.
3. The predominately white-favored openings in the dataset cause issue.
4. The opening set is too large.
5. Not enough training iterations.

Future work should consider exploring these angles.

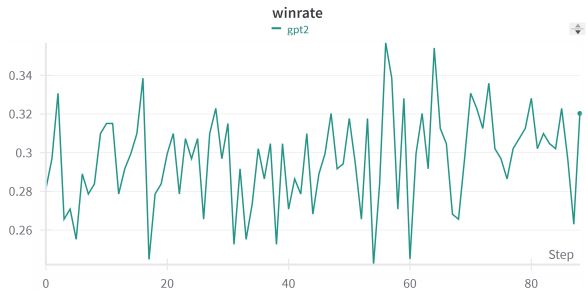

Figure 5. We see that the win rate does not improve over many iterations, although there may be a slight trend upwards.

Given more compute, it would be interesting to explore how this win rate changes for even larger k. Unfortunately, it was clear the policy would not train with a reasonable time window.

5.4. FSP With Short-term Adversarial Agents

Self-play showed diverge due to instability very quickly. Past-Present Q-iteration stabilizes this. However, this would sometimes train and adversarial network to the past network, rather than a good chess policy. While FSP with LLMs as pseudo-ensembles ideally would fix this problem, it was impossible to find converge at all. Considering the inability to train a policy to beat a pseudo-ensemble of frozen policies, yet the clear ability to very quickly train an adversarial policy, we propose FSP with Short-term Adversarial Agents, a novel self-play algorithm that seeks to regularize instability inspired by concepts from Wang et al. (2023) and Plappert et al. (2021). FSP with Short-term Adversarial Agents exploits 2 main ideas: 1. Training an adversarial chess policy with sparse reward is fast; 2. In chess (and other games), an adversarial policy does not have to be a true good policy, however, a true good policy must be adversarially robust. We outline the algorithm below:

Algorithm 4 FSP With Short-term Adversarial Agents

$\pi_{adv} \leftarrow$ `pretrained agent`
$\pi_\theta \leftarrow$ `pretrained agent`
for each training iteration **do**
  `unfreeze`(π_{adv})
  `freeze`(π_θ)
  for each adversarial step **do**
    Train π_{adv} against π_θ
  end for
  `unfreeze`(π_θ)
  `freeze`(π_{adv})
  for each policy step **do**
    Train π_θ against π_{adv}
  end for
  $\pi_{adv} \leftarrow$ `pretrained agent`
end for

The algorithm regularizes the adversarial by forcing it to reset every iteration, meaning the adversarial must train anew with `num adversarial steps`, usually about 9, for a total of $9 * 64$ games. This prevents the adversarial policy from becoming too different from the already good pretrained model. This also allows the adversarial policy to never fully commit to a certain strategy; at each training iteration, the adversarial can learn the new best adversarial against the current policy. The `num policy steps` is purposely less than `num adversarial steps` in order to prevent the policy from learning a reverse adversarial policy. Ultimately, we can think of the algorithm in 2 ways: 1. A policy must learn to defend against custom adversarial agents, forcing it to be adversarially robust; 2. An adversarial policy must continually learn new adversarial strategies against an increasingly adversarially robust policy. Both perspectives imply that one or both of the policies should improve.

5.4.1. RESULTS

We find relative success with FSP With Short-term Adversarial Agents. First, we find that we can train an adversarial very quickly, roughly in 9 iterations.

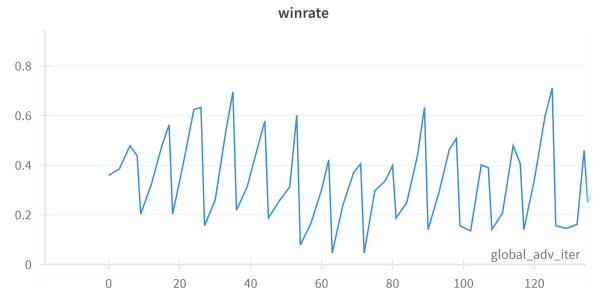

Figure 6. Adversarial win rate vs adversarial iteration (zoomed in): we see that win rate spike quickly each time the new adversarial is trained.

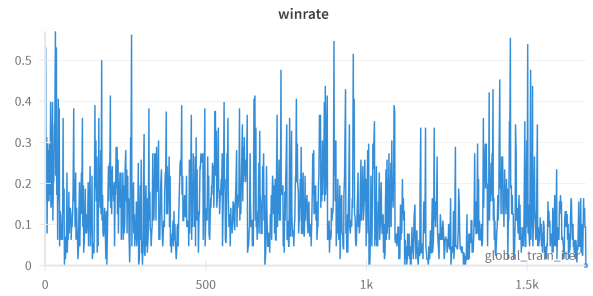

Figure 7. Adversarial win rate vs adversarial iteration (zoomed out): We see that the adversarial is always able to find a strong adversarial policy in 9 steps. We notice that the starting adversarial win rate increases and decrease, but is generally very low.

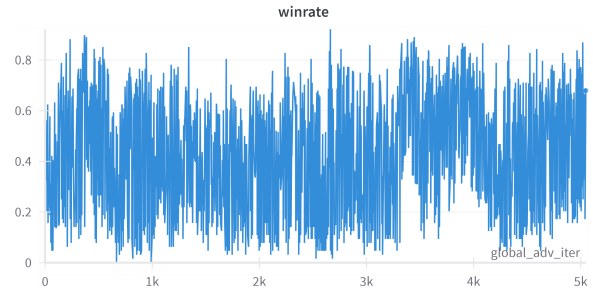

Figure 8. Policy win rate vs policy iteration: We see that the policy generally is able to defen the adversarial policy up to above a 0.33 win rate. We also see that at the end, the policy seems unable to define the adversary.

We saved the adversarial models trained at different steps. We expected the early adversarial policies to be bad policies, but tests against Stockfish show otherwise.

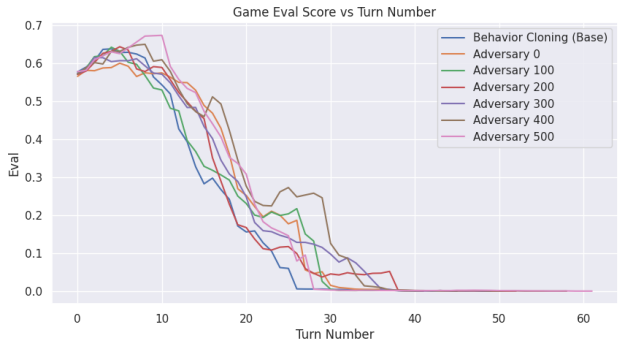

Figure 9. Eval over turn when playing Stockfish at 3000 ELO.

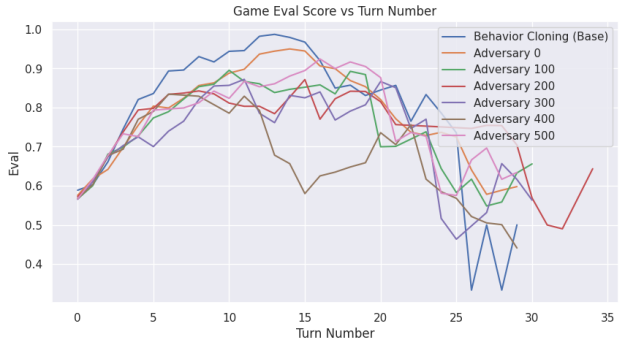

Figure 10. Eval over turn when playing Stockfish at 1500 ELO.

Our figures show that the adversarial policies seem to mildly improve against the baseline. Note that all adversarials are trained with 9 gradient steps from the base policy, improvement is only due to playing against an improved actual policy.

Interestingly, the adversarial policies only improve with time when playing against Stockfish 3000 ELO, and not 1500 ELO, possibly because of out of distribution states. We explore the success of the policy in the overall results section below.

Table 1. Adversarial Self-Play statistics against the adversarial model at iteration 5 (win means adversarial 5 won): the win/loss/draw ratio indicates future adversarial models still remember how to beat past adversarial models.

Policy at Train Step	5	10	560
Win rate	0.60	0.26	0.20
Loss rate	0.22	0.46	0.40
Draw rate	0.18	0.28	0.40

We also play an old adversary against the newest policy and compare the win rate against the old policy trained to

defend against the old adversary policy. In the above table, adversarial 5 is designed to defeat policy 5, policy 10 is trained defend against adversarial 5, and policy 560 remembers how to defend against adversarial 5 many iterations later. Therefore, it seems that we do not have a cyclical effect such that defending against new adversaries makes the policy to forget other previous defense strategies. Instead, the policies retains previous defense strategies naturally.

5.5. Offline Learning

5.5.1. Description

All online methods were exceedingly sample inefficient and slow to run due the cost of inference for decoder models. We opted to try a simple offline method, encouraged by the success of behavior cloning. We devised an offline algorithm to run the trajectory based loss detailed in the RL Loss Choices section. We award trajectories where the agent in the data one, and penalize trajectories where the agent in the data lost. In doing so, we hope to replicate the best moves seen the dataset, rather than the average move seen in the dataset.

5.5.2. Results

We find empirically that the offline learning results in unusable policy, even when fine-tuned from the behavior cloned model. We notice that the offline trained model plays extremely unoptimial openings, unlike the other usable policies. We reason that the model is encouraged to play out of distribution policies because the offline loss has pushed down entire trajectory probabilities, including the beginning moves that may have been good. Unoptimial moves not seen in the dataset are never altered, so during test time those moves have the greatest probability of being played. Future attempts at offline strategies in self-play should consider how to protect against this out of distribution issue.

5.6. Funnel Search

5.6.1. Description

When examining the performance of all models against Stockfish, we discover our models play well during roughly the first 30 moves, perform worse in the mid game, and perform poorly in late game. This is due to the limited variety of early game variations in chess leading to our data being well represented by trajectories with similar openings and early games. However, as trajectories goes into mid and late game, the number of variations approaches near infinity. We believe increasing exploration as games move into later phases will improve our current reinforcement learning schemes.

Hence we present our final novel algorithm, Funnel Search. Funnel Search takes inspiration from Beam Search, an well-known algorithm for Language Models to explore by expanding the most promising nodes in a limited set, or "beam," of candidate solutions at each level to efficiently find optimal solutions (Freitag and Al-Onaizan, 2017). Naively implementing Beam Search in the training loop will yield unoptimal exploration due to the greedy nature of the Beam Search algorithm. Therefore we modify Beam Search to sample trajectories at a given state from its action distribution with a temperature setting. Further, instead of setting a constant k for all time steps, which would blow up too quickly, we calibrate a "k scheduler" and "m scheduler" to determine the number of beams at any state allowing a precise control on the growth of the "funnel".

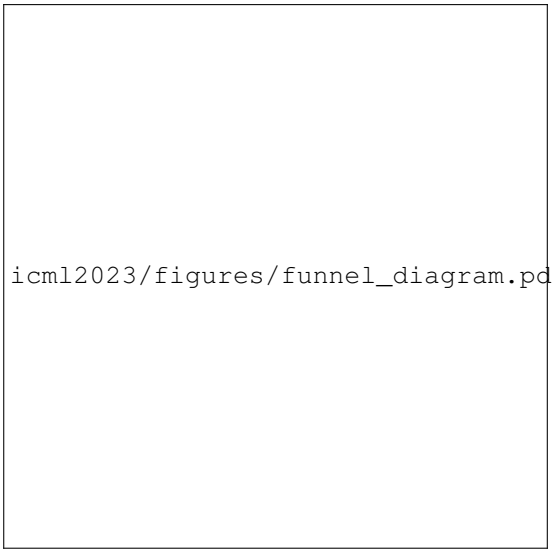

Figure 11. Funnel Search Graph: We sample h number of moves from the action distributions at a given state, where h is 2 with probability k = 0.5, otherwise h = 1. Then from all the sampled moves, we select top m moves. The red arrows are the sampled moves, the blue arrows are sampled moves that aren't in the top m moves, and the black arrows are the moves that aren't sampled.

Algorithm 5 Funnel Search

Require: $k_i, m_i, h'_i \geq 1 \; \forall i \in [0, N]$
  $N \leftarrow$ `max number of moves`
  $B \leftarrow$ `[]`
  for $i \leq N$ **do**
    $L \leftarrow$`[]`
    for each beam b_i **do**
      $h \leftarrow h \sim Pr(H = h') = k_i$ `else 1`
      `sample` h `logits no replacement`
      `logits` $\leftarrow$ `log(softmax(logits))`
      L `extends logits`
    end for
    `pick top` m_i `beams` b' `from` L
    if b'_j terminates **then**
      B `appends` L
    end if
    $b \leftarrow b'$
  end for
  return B

The intuition behind Funnel Search is as follow: we assume the reason why a trajectory loses or draws a game is because the model made a blunder after playing an perfect early game, which puts the trajectory in an unrecoverable state. It is unlikely because the entire trajectory was bad or the latest moves were bad. In chess, it is safe to assume the opponent will take advantage of any mistakes, especially against high elo engines like Stockfish. A blunder, such as losing a queen, will put a trajectory into a state that likely leads to a loss or draw no matter what happens after. Hence, we should increase exploration in the middle and late game to explore more possibles moves and learn to not enter states leading to a loss or draw. In Funnel Search, a blunder in the middle game will branch out into multiple losing and drawing trajectories in the late game, which all trace back to the same blunder. This implies reinforcement learning with Funnel Search heavily disincentivizes blunders and encourages states that often lead to a win. See in Figure 11 for Funnel Search's visualization and in Algorithm 5 for the searching algorithm.

5.6.2. Results

We primarily use Funnel Search to collect trajectories during self-play style training, and then apply BCELoss on the log probabilities of white and black separately with their corresponding reward and back propagate. For more details on the training algorithm see algorithm 6. However, this yields disappointing results: the loss quickly converges to zero within 5 iterations. Upon inspection of the resulting games, the model learned to draw the game using repetition to minimizes the loss, likely due to the single-model training setup with limited reward signals.

In an optimistic attempt to resolve this issue, we convert our loss function to applying BCELoss on each individual action instead of the sum across the trajectory. For the drawing case in the previous loss function, the sum of all log probabilities across the trajectory approaches zero as the game length increases, which leads to a zero overall loss. After using the new loss function, the results remain poor. This is likely due the sparse reward doesn't provide constructive feedback to each action in the trajectory, forcing the model to opt for drawing as an alternative to minimizing loss. A good analogy is a confused student guessing C for all the multiple choice questions on an exam.

From the perspective of computational efficiency, Funnel Search is actually a good exploration policy to collect useful trajectories. It is able to collect over 1000 trajectories within 3-5 minutes when batched on reasonable computing resources, making it the fastest training algorithms in this paper.

Algorithm 6 Self-Play RL with Funnel Search

while train **do**
  τ, reward $\leftarrow$ funnel search $(\pi_\theta, \dots)$
  pred $\leftarrow [\sum_a log(p(\tau_{white})), \sum_a log(p(\tau_{black}))]$
  target $\leftarrow [r(\tau_{white}), r(\tau_{black})]$
  loss $\leftarrow$ BCELoss (pred, target)
  backpropagate loss

6. Overall Results

We detail a comparison between the different algorithms presented that resulted in playable policies.

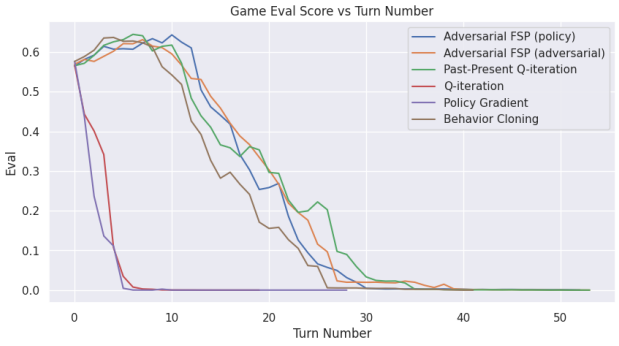

Figure 12. Game Evaluation vs Turn number (higher is better). We play our policies against max strength Stockfish, with our model as white. We observe how long the model can defend against Stockfish, and how much of an advantage it can build during the opening. 1.0 means white is expected to win, 0.0 means white is expected to lose, 0.5 means a draw is expected.

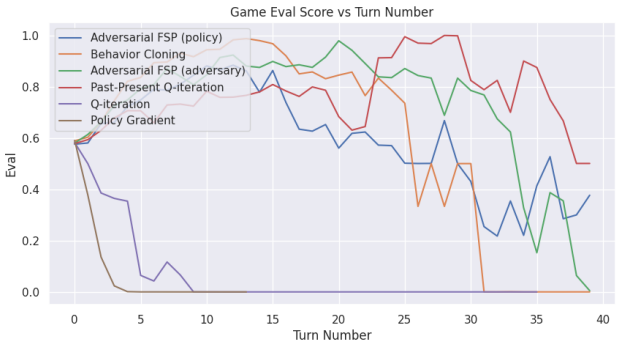

Figure 13. Game Evaluation vs Turn number (higher is better). Same as Figure 12 but Stockfish is set to 1500 ELO

When playing against max strength Stockfish, we find that Adversarial FSP policy, Adversarial FSP adversarial, and Past-Present Q-iteration all result in policies with mild improvement over the baseline Behavior Clone policy. We observe that the Adversarial FSP policy is very strong in the opening. Past-Present Q-iteration has the best survivability. Q-iteration and Policy gradient do not generate use-able policies. We note that longer survivability is exponentially harder as the number of possible states increases exponentially with each move. Therefore, small increases in eval performance imply reasonable increases in policy ability.

When playing against 1500 ELO Stockfish, Past-Present Q-iteration seems strongest, with the Adversarial FSP adversary policy and the baseline model close behind. It is unclear why the policies do not show large improvement. One possibility is out of distribution states.

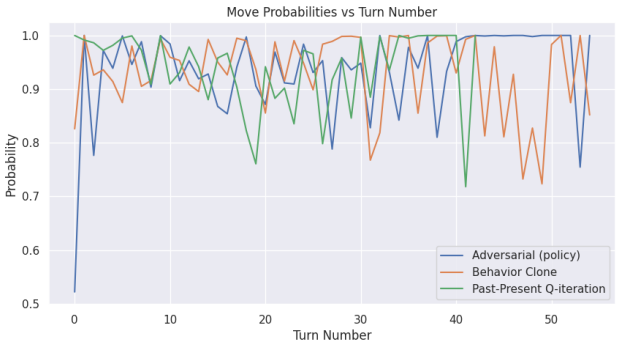

Figure 14. Move Probabilities vs. Turn Number: We can see that adversarial policy is significantly less certain about its move in the opening. Models with RL is also more certain in the late game.

7. Conclusion

We find the sparse-reward, state-inferred, and action-only learning problem to be exceedingly difficult. The divergent nature of chess trajectories and massive amount of states contributes to a highly unstable and difficult to learn environment. Single model self-play is too unstable to learn good policies, even with more optimal trajectory planning through Funnel Search. Dual-model self-play systems utilizing past and present or adversarial and policy greatly stabilize learning and produced improved policies over baseline. The inability to learn in the pseudo-ensemble environment implies that the environment is likely still too difficult to truly learn a global policy.

In the future, we believe there are several possible works could be done to improve our algorithms. First, Dual-Play: initializes two base models and then train one model solely plays white while the other plays black. Second, funnel search could be used to boost mid and late game exploration for any other algorithm used in the paper, especially on dual-model systems such as Adversarial Self-Play which protect against divergence, resulting a new scheme: Adversarial Funnel Search. Pseudo-ensemble could be attempted with varying opening book sizes to increase or decrease difficulty. Adversarial self-play could be attempted on other sequence games, or other multi-agent environments (such as language models in interaction). Additionally, integrating meta-learning and trasfer learning could enable models to quickly adapt to new game scenario and opponent strategies. This could significantly reduce the training time and computational resources required to achieve high levels of play.

References

M. Freitag and Y. Al-Onaizan. Beam search strategies for neural machine translation. *arXiv preprint arXiv:1702.01806*, 2017.

J. Heinrich and D. Silver. Deep reinforcement learning from self-play in imperfect-information games. *arXiv preprint arXiv:1603.01121*, 2016.

A. Laterre, Y. Fu, and Jabri. Ranked reward: Enabling self-play reinforcement learning for combinatorial optimization. *arXiv preprint arXiv:1807.01672*, 2018.

D. Noever, M. Ciolino, and J. Kalin. The chess transformer: Mastering play using generative language models, 2020.

M. Plappert, R. Sampedro, and Xu. Asymmetric self-play for automatic goal discovery in robotic manipulation. *arXiv preprint arXiv:2101.04882*, 2021.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever. Language models are unsupervised multitask learners. 2019. URL https://api.semanticscholar.org/CorpusID:160025533%7D.

T. T. Wang, A. Gleave, T. Tseng, K. Pelrine, N. Belrose, J. Miller, M. D. Dennis, Y. Duan, V. Pogrebniak, S. Levine, and S. Russell. Adversarial policies beat superhuman go ais, 2023.

Z. Zhang and M. R. Sabuncu. Generalized cross entropy loss for training deep neural networks with noisy labels. *arXiv preprint arXiv:1805.07836*, 2018.

A. Contributions

Evan Frick: Evan was primarily responsible for pretraining the GPT-2 model, finetuning, determining the loss functions, Adversarial Self-Play, Past-Present Q-iteration, plotting evaluations, and dataset collection and curation. Evan also experimented with algorithms that did not make it into the final paper: min max q-iteration, direct logit Q-iteration.

Tianle Li: Tim was primarily responsible for Self-Play with Funnel Searching, StockFish evaluation, checkpoint evaluation, and Direct Self-Play. Tim also experimented with algorithms that did not make it into the final paper: Exploration of Value-based methods.

Kayla Lee: Kayla was primarily responsible for Policy Gradient, Q-Iteration, Offline Learning. Kayla also experimented with algorithms that did not make it into the final paper: Beam Search.

Everyone: Everyone contributed to discussion of ideas, implementations, experiments, and research directions. All parties contributed equally to writing the paper. Everyone assisted with derivations of relevant proofs and losses. All parties were involved with research related work. All parties were involved in the empirical evaluation of models.

TAs: We especially thank the TAs Vivek, Kevin, and Joey for providing valuable advice and direction for this project during office hours.

Professor: We thank Professor Sergey Levine for the fantastic project opportunity and advice on self-play.