Transformer-based Generative RL for Tractor

柯绎思 [1]
School of EECS
2400013062@stu.pku.edu.cn

柯怿憬 [2]
School of EECS
2400013060@stu.pku.edu.cn

Abstract

We present a generative reinforcement learning approach for Tractor, a four-player trick-taking card game with imperfect information and large combinatorial action spaces. Unlike traditional methods that treat actions as discrete classification, we formulate the entire game as a sequential token generation problem, modeling trump declaration, card burial, and trick-taking as a unified autoregressive sequence. Our Transformer-based policy network naturally handles variable-length multi-card actions while capturing long-range dependencies across game trajectories. Trained entirely from scratch using PPO with self-play, our agent learns effective strategies without human expert knowledge or heuristics. After training on 5 million rounds of self-play, the final model acquires an effective strategy, demonstrating that end-to-end generative RL can successfully tackle complex imperfect-information games.

1 Introduction

Tractor is a four-player partnership trick-taking card game with imperfect information, partner coordination, and a large combinatorial action space. The game is played with two decks (108 cards total) by four players in fixed positions, with opposite players forming partnerships. The game proceeds through three stages: (1) *Deal stage*, where cards are dealt and players declare trump suit by revealing level cards; (2) *Cover stage*, where the banker who declared trump buries 8 cards from the kitty; and (3) *Play stage*, where players take tricks following suit rules, with the attacking team aiming to capture point cards. The final score determines level advancement based on points captured by attackers.

We present a generative reinforcement learning approach for Tractor with two key contributions:

Generative RL Formulation. Rather than treating actions as discrete classification over a fixed space, we formulate Tractor as a sequential token generation problem. The entire game—including trump declaration, card burial, and trick-taking—is modeled as a unified autoregressive sequence where the agent generates action tokens conditioned on game history. This token-based formulation naturally handles variable-length multi-card actions (singles, pairs, tractors, throws) without combinatorial enumeration, while the Transformer architecture captures long-range dependencies across the full game trajectory.

Learning from Scratch. Our agent learns entirely through self-play from random initialization, without incorporating any human expert knowledge or heuristics. Unlike card game AI systems that rely on hand-crafted features or imitation learning from human data, our model discovers effective

[1] Implemented the model architecture (section 2.3), optimized rollout with KV caching (section 2.4), developed the asynchronous RL framework (section 2.6), and conducted experiments.

[2] Designed the environment-agent interface, the state and action representations (sections 2.1 and 2.2), implemented the training algorithm (section 2.5), and wrote the report (assisted by LLM).

Term Project for Reinforcement Learning course in Peking University (2025 Fall).

strategies for all game stages purely from reward signals, demonstrating that end-to-end reinforcement learning can acquire complex card game strategies without human guidance.

2 Method

We present our generative reinforcement learning approach for Tractor. The key insight is to model the entire game as a sequential decision process where the agent autoregressively generates action tokens conditioned on the game history. This formulation naturally handles the variable-length game trajectories and combinatorial action spaces inherent in Tractor.

2.1 State Representation

We represent the game state as a sequence of tokens, where each token encodes either a game event or an action taken by a player. The token vocabulary is divided into several categories:

Card Tokens (0–53). Each unique card is assigned a token ID based on its suit and rank. Cards are encoded relative to the current level to normalize the representation:

- Normal cards: tokens 0–47 (12 ranks $\times$ 4 suits, excluding level cards)
- Level cards: tokens 48–51 (4 suits)
- Small joker: token 52
- Big joker: token 53

Output Tokens (54–163). These tokens represent cards played during the trick-taking stage. Since two decks are used, there are 108 possible card instances plus 2 special tokens for the killing and end-of-action marker.

Level Tokens (164–170). These encode the current game level, discretized into 7 categories representing different scoring thresholds.

Trump Tokens (171–175). These indicate the declared trump suit (spades, hearts, clubs, diamonds) or no-trump.

Special Tokens (176–177).

- Cover token (176): Marks the transition to the card burial stage
- Trick token (177): Marks the boundary between tricks

Each token is paired with a player ID (0–4, where 0 indicates no specific player) to indicate which player is associated with the event. The player IDs are encoded relative to the current player's perspective to maintain permutation equivariance.

2.2 Action Representation

Actions are represented as discrete tokens from a vocabulary of 175 possible actions, partitioned by game stage:

Deal Stage Actions (0–10). During dealing, players can:

- Declare trump with a single level card: actions 0–3 (by suit)
- Declare trump with a pair of level cards: actions 4–7 (by suit)
- Declare trump with joker pairs: actions 8–9
- Pass (do nothing): action 10

Cover Stage Actions (11–64). During card burial, the banker selects 8 cards to bury. Each action corresponds to selecting one of the 54 possible card types.

Play Stage Actions (65–174). Same as output tokens (see section 2.1).

For each stage, valid actions are constrained by an action mask computed by the game engine, which enforces game rules (e.g., following suit when possible, playing valid combinations).

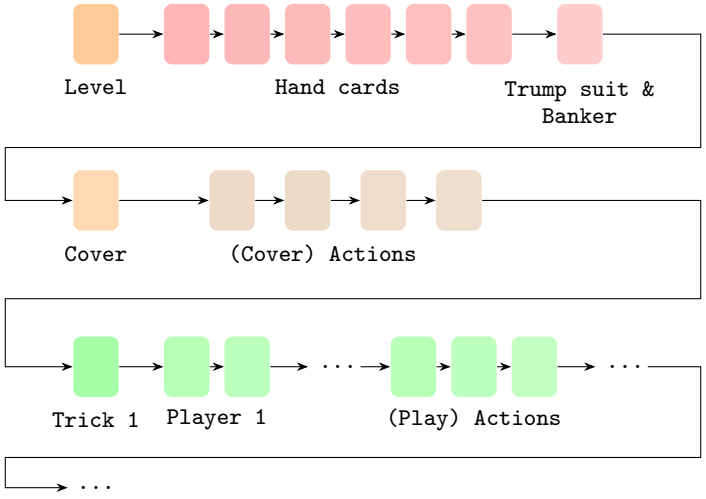

Figure 1: State and action tokenization.

2.3 Model Architecture

Our policy network is based on the Transformer architecture [6], modified for efficient autoregressive inference in the RL setting.

Input Embedding. The input consists of two components, token embedding and player embedding, mapping token IDs and player IDs to d-dimensional vectors. The final input representation is the sum of these two embeddings.

Transformer Backbone. We use a decoder-only Transformer with pre-norm layer normalization, multi-head self-attention with causal masking, and SwiGLU feed-forward networks [4]. We employ Rotary Position Embedding (RoPE) [5] for position encoding. Detailed architecture specifications are provided in appendix A.

Output Heads. The Transformer output is fed to three separate linear heads:

- **Policy head**: Projects to the action space, producing logits for action selection
- **Value head**: Projects to a scalar value estimate for the current state
- **Mask prediction head**: Auxiliary head that predicts the valid action mask (used only during training)

2.4 Rollout

During rollout, the model autoregressively generates actions given the current game state:

1. Encode the new observations as tokens and append to the sequence
2. Compute the policy logits using the Transformer with KV-caching
3. Apply the action mask to zero out invalid actions
4. Sample an action from the masked distribution (or take the argmax for deterministic play)
5. Execute the action and receive the next observation

For multi-card actions (e.g., playing a tractor or throw), the model generates multiple tokens sequentially until it outputs the end-of-action marker.

2.5 Training

We train the policy using Proximal Policy Optimization (PPO) [3] with a self-play framework.

Self-Play with Model Pool. To ensure training stability and prevent overfitting to a single opponent strategy, we adopt a self-play framework similar to OpenAI Five [1], maintaining two model pools:

- **Latest model**: Contains the most recent model checkpoint
- **Checkpoint pool**: Contains up to 16 historical checkpoints sampled periodically

During each episode, we pair two teams:

- Team 1 (learning agents): Use the latest model
- Team 2 (opponents): Use a model selected as follows:
  - 60% probability: The latest model (self-play)
  - 40% probability: Uniformly sampled from the checkpoint pool

This strategy balances learning against the current best policy while maintaining robustness against diverse historical strategies. It prevents the agent from overfitting to a single opponent's strategy and encourages the development of more generalizable playing skills.

PPO Objective. The policy is optimized using the clipped PPO objective:

$$L^{\text{PPO}}(\theta) = -\mathbb{E}\left[\sum_t \min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{1}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the importance sampling ratio and $\hat{A}_t$ is the generalized advantage estimate (GAE) [2]. The PPO objective also includes a value loss (mean squared error between predicted and target values) and an entropy bonus that encourages exploration by penalizing low-entropy policies.

Representation Learning. In addition to the standard PPO losses, we include an auxiliary mask prediction loss: binary cross-entropy for predicting valid action masks, which improves the model's understanding of game rules.

The total loss is:

$$L = L^{\text{PPO}} + c_v L^{\text{value}} + c_e L^{\text{entropy}} + c_a L^{\text{aux}} \tag{2}$$

where c_v, c_e, and c_a are hyperparameters (see appendix A for details).

Reward Shaping. We use a combination of:

- **Immediate rewards**: Points captured during each trick (scaled by 0.01)
- **Penalty**: Points lost due to invalid throw attempts (scaled by 0.01)
- **Final reward**: Based on the game outcome (levels won/lost)

2.6 Asynchronous RL Framework

To achieve high training throughput, we employ an asynchronous actor-learner architecture that decouples trajectory generation from policy optimization. Our framework consists of two types of processes running concurrently across 6 GPUs. Figure 2 illustrates the overall architecture.

Actor Processes. We deploy 4 actor processes, each utilizing a dedicated GPU for policy inference during rollout. The actors asynchronously generate game trajectories by:

1. Loading the latest policy weights from a shared memory pool
2. Simulating complete Tractor games using the current policy (with opponents sampled according to the strategy described in section 2.5)
3. Computing advantages using GAE and packaging trajectories into experience tuples

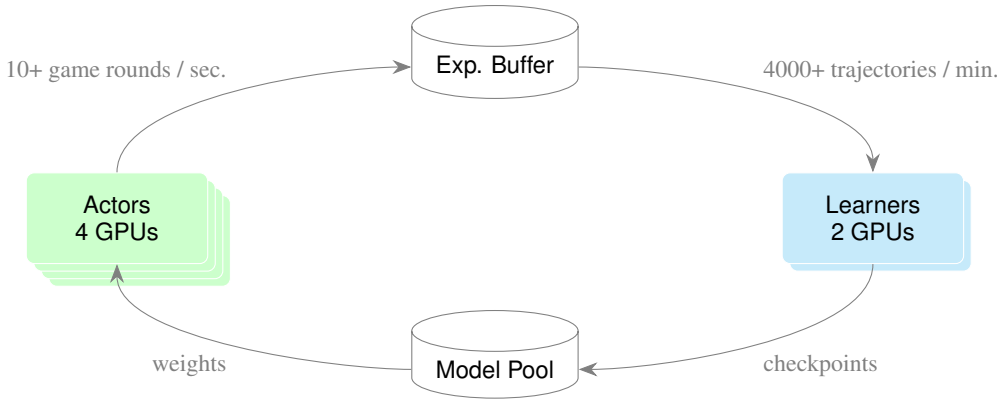

Figure 2: The asynchronous actor-learner training framework.

4. Pushing completed trajectories to a shared experience buffer

The asynchronous design allows actors to continuously generate experience without blocking on learner updates, maximizing GPU utilization for inference.

Learner Processes. We deploy 2 learner processes, each utilizing a dedicated GPU for gradient computation. The learners operate in a synchronous data-parallel fashion:

1. A batch of experiences is popped from the shared experience buffer
2. The batch is scattered across the 2 learners for data-parallel training
3. Each learner processes its portion of the batch in mini-batches, computing gradients locally
4. Gradients are synchronized across both learners via all-reduce operations
5. The synchronized gradients are applied to update the policy parameters

This data-parallel training strategy enables efficient scaling while maintaining gradient consistency.

Synchronization and Checkpointing. The actors and learners communicate through two mechanisms:

- **Weight synchronization**: Actors periodically fetch the latest policy weights from shared memory, ensuring they generate trajectories using a reasonably up-to-date policy while tolerating minor staleness for throughput
- **Checkpoint pool updates**: Every 200 training iterations, the learners save the current policy to the checkpoint pool (see section 2.5), maintaining a diverse set of historical opponents for robust self-play training

This architecture achieves a balance between sample efficiency and computational throughput, enabling the collection of millions of game trajectories required for learning effective Tractor strategies.

3 Results

We trained the agent for 10^4 training iterations, corresponding to approximately 5 million game trajectories generated through self-play. The training process took around 48 hours on our asynchronous RL framework utilizing 6 GPUs.

3.1 Evaluation Setup

To evaluate the effectiveness of our approach, we established a baseline model for comparison. The baseline consists of an earlier trained model with a smaller model dimension trained using only immediate rewards without the final outcome-based rewards.

We evaluated 5 checkpoints from our training run. For each checkpoint, we conducted 2048 evaluation games against the baseline model, where our checkpoint models played as one team and the baseline model controlled the opposing team. We recorded both the win rate and the average final score for each checkpoint.

3.2 Performance Against Baseline

Figure 3 presents the results of our evaluation. The checkpoint models show consistent improvement throughout training. The average final score follows a similar trend, demonstrating that our model not only wins more frequently but also achieves larger margins of victory.

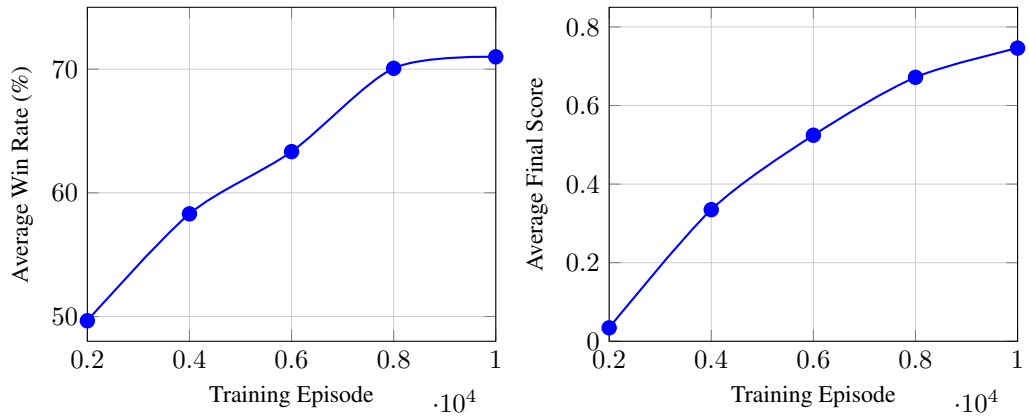

Figure 3: Learning curve showing average win rate and final score against the baseline model over training episodes.

The results demonstrate that our generative reinforcement learning approach successfully learns effective strategies for Tractor through self-play. The final checkpoint achieves a strong 70% win rate against the baseline, suggesting that the combination of our Transformer-based architecture, reward shaping, and self-play training framework is effective for this complex card game domain.

4 Conclusion

We presented a generative reinforcement learning approach for the card game Tractor that formulates the entire game as a sequential token generation problem. By treating actions as autoregressive sequences rather than discrete classifications, our Transformer-based architecture naturally handles the variable-length, combinatorial action space while capturing long-range dependencies across game trajectories. The agent learns entirely from scratch through self-play using PPO, acquiring effective strategies for all game stages without human expert knowledge. This work demonstrates that generative RL with modern Transformer architectures can successfully tackle complex imperfect-information games.

References

[1] OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique Pondé de Oliveira Pinto, Jonathan Raiman, Tim Salimans, Jeremy Schlatter, Jonas Schneider, Szymon Sidor, Ilya Sutskever, Jie Tang, Filip Wolski, and Susan Zhang. Dota 2 with large scale deep reinforcement learning. *arXiv preprint arXiv:1912.06680*, 2019.

[2] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. In *International Conference on Learning Representations*, 2016.

[3] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[4] Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

[5] Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[6] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, 2017.

A Hyperparameters

Table 1: Training hyperparameters

Hyperparameter	Value
Discount factor γ	0.99
GAE parameter λ	0.95
PPO clip parameter ϵ	0.2
Value loss coefficient c_v	0.5
Entropy coefficient c_e	0.005
Auxiliary loss coefficient c_a	0.05
Learning rate	10^{-5}
Adam ϵ	10^{-5}
Rollout batch size per GPU	128
Training batch size per GPU	512
Training mini-batch size	64
PPO epochs	2
Gradient clipping	1.0

Table 2: Model architecture

Component	Specification
Token vocabulary size	178
Action vocabulary size	175
Model dimension d	384
Number of Transformer layers	16
Number of attention heads	12
Hidden dimension (per head)	64
Maximum sequence length	320
FFN hidden dimension	~ 1024 (SwiGLU)
Position encoding	RoPE
Normalization	Pre-LayerNorm