IRSTI 28.23.25 https://doi.org/10.26577/jpcsit4220261

**Dinara Zhussupova**

Institute of Mathematics and Mathematical Modeling, Astana, Kazakhstan
e-mail: zhus.dinara@gmail.com

# APPLYING SELF-PLAY REINFORCEMENT LEARNING TO KAZAKH NATIONAL BOARD GAMES: A CASE STUDY ON TOGYZKUMALAK

**Abstract.** The paper presents the results of a study on training agents for competitive board games, using the Kazakh national Togyzkumalak game, a zero–sum game, as a case study. As demonstrated by Alpha Zero, it is possible to train a champion without human expert knowledge or supervised learning. However, each game has unique characteristics that require specific research to develop an excellent player. In this study, two approaches were used to train agents: model-free and model-based. The self-play technique was used effectively during the training. As a result, the agents developed the ability to play Togyzkumalak at a competitive level.

**Keywords:** Multi-agent reinforcement learning, cumulative reward, Q-learning, zero-sum game, Monte Carlo tree search.

## 1. Introduction

Multi-agent learning uses reinforcement learning algorithms, but it is a significantly more complex area of agent learning. In reinforcement learning (RL), the agent's goal during interaction with the environment is to maximize the reward assigned to it at each moment in time [1]. The reward signal helps the agent determine whether it is acting correctly in the environment or if it needs to adjust its policy moving forward.

In multi-agent reinforcement learning (MARL), the problem is complicated by the fact that multiple agents in the environment may have different goals and varying knowledge about the environment. In addition, each agent's reward may depend on other players' actions and policies.

The concept of cumulative reward helps to evaluate the future rewards of actions taken at the current time. This type of reward informs the agent about how promising a particular action is in the short term and how optimal it is over the long term. The concepts of cumulative reward and value function are central to most reinforcement learning methods.

Reinforcement learning for multi-agent environments [2] is based on RL, where agents learn the optimal policy through trial and error, aiming to maximize the received reward. While single-agent RL focuses on learning the optimal policy for an individual agent, MARL addresses learning optimal policies for multiple agents and the unique challenges that arise in this process.

An important feature of MARL is the non-stationarity caused by the constantly changing policies of agents during learning. This non-stationarity can lead to a moving target problem, as each agent adapts to the policies of others, whose policies are also continuously adjusting in response. This can result in cyclical and unstable learning dynamics.

The ability to robustly bandle such non-stationarity is thus often a critical aspect of MARL algorithms and has been the focus of extensive research [3], [4], [5].

In RL, an agent's policy is considered optimal if it yields the highest reward in each state. In a multi-agent environment, however, the optimal policy for one agent depends on the policies of other agents due to their competitiveness or cooperativeness. Therefore, a more comprehensive approach is necessary. In Chapter 4 of the book [2] there are various approaches to setting the problem and solving them for several agents interacting in one environment. Additionally, unlike in RL, where the optimal policy consistently provides the maximum reward, in MARL the situation is more complex, since an agent's reward also depends on the actions of other agents. The main goal of MARL is to develop learning methods that lead to policies that merge to a solution.

MARL is essentially RL applied to multi-agent game models to learn optimal policies for agents. Thus, MARL is based on both RL theory and game theory. Agents interact within an environment to achieve certain goals. This concept can be defined

using models of multi-agent interaction. These models, based on game theory, are commonly referred to as games.

Minimax is a solution concept used in two-agent zero-sum games where the reward of one agent is negative compared to the reward of the other. The existence of a minimax solution for normal-form games is discussed in Neumann [6], [7]. The minimax problem can be solved using linear programming algorithms, such as the simplex method [8].

Independent learning often uses the self-play technique [9], [10], [11]. Although independent learning usually implicitly assumes self-play (e.g., in independent Q-learning), it is not strictly required as agents can use different algorithms within this approach. Self-play has been used effectively to train Alpha Zero AI [12] and to develop competitive agents capable of learning complex motor skills such as wrestling and ball-playing [13]. Recent work has introduced several multi-agent tasks with competing goals in a 3D world with simulated physics in the MuJoCo framework [14], where agents must learn complex motor skills to thrive in a competitive environment.

A recent breakthrough in MARL was the victory of the OpenAI team-trained agents in the game DOTA2 [15] against a team of professional players. However, it is important to note that this victory was not entirely unrestricted, as there were limitations on role selection, locations, and some features for players.

The next advancement in this field was to train agents without any prior knowledge of the game.

MuZero AI was presented by Google DeepMind researchers in the work [16].

This paper explores methods for training agents to play Togyzkumalak. Structure of the paper:

- Section 2 contains information about the game and the rules;
- Section 3 outlines the challenges of training an agent for Togyzkumalak, including search complexity, game depth, chaotic mechanics, and the need for extensive self-play;
- Research methods are described in Section 4;
- Section 5 describes the development tools used in the research;
- Section 6 presents the results for two algorithms;
- Future research and interesting conclusions from the obtained results are described in Section 7.

## 2. About game

Togyzkumalak is one of the traditional Kazakh national games. It is a strategic board game played on a special board with 162 stones. In 2020, UNESCO included Togyzkumalak in the Representative List of Intangible Heritage.

### *2.1 Description*

The game of Togyzkumalak is played between two players on a special board. The board features two accumulation holes and 18 game holes (see Figs. 1 and 2). During the game, each player controls one accumulation hole. At the start of the game, 162 stones are distributed across the 18 game holes with 9 stones in each hole.

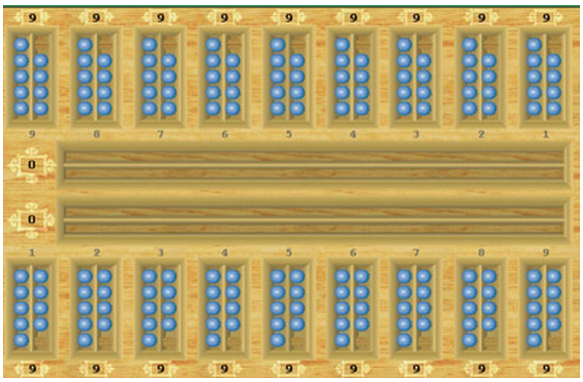
The player who makes the first move is called ”Bastaushy (Beginner)”, while the player who makes the subsequent move is called ”Kostaushy (Continuer)”. Sometimes, the terms ”white side” and ”black side” are used to refer to the Beginner and Continuer, respectively.

The general structure of the board is shown in Figs. 1 and 2.

In Fig. 2, the player’s game holes are positioned closer to them, while the accumulation hole, called ”Kazan”, is located opposite the opponent. The number of stones in the Kazan is indicated at the bottom of the cell, and the total number of stones it contains there is also displayed.

**Figure 1**
*Wooden board with ornaments*

**Figure 2**
*The standard game board*



*2.2 Basic rules of the game*

The game is played by two competing players. Togyzkumalak is a zero-sum game, i.e. the stones won by one player are effectively subtracted from the other player's total.

Before the game starts, the board is set up in the starting position. Here are the basic rules of the game.

1) Description of a player's move.

- The player takes all the stones except one from one hole and places one stone in each cell in a counterclockwise direction. The move must be made only from a non-empty hole; if all the holes are empty, this position is called "Atsyrau" (see 3).

- If the last hole contains an even number of stones and is on the opponent's side, the player captures these stones and places them in their accumulation hole.

Exception. If there is only one stone in the current hole, it is moved to the adjacent hole on the right.

Exception. You cannot move stones from a hole captured by your opponent, referred to as from Tuzdyk.

2) Getting a Tuzdyk. Tuzdyk is a hole captured from your opponent. Let us look at cases where a player gets a Tuzdyk:

a) if the last stone placed does not land in the opponent's final cell and the number of stones in that hole becomes 3, the player captures as a Tuzdyk

b) it is not allowed to capture a Tuzdyk in the same numbered hole as the opponent's Tuzdyk (for example, if your opponent has captured hole number 3, you cannot capture a hole with the same number).

3) Atsyrau. A situation in the game when a player cannot make any move because there are no stones left in their game holes.

4) Winning the game.

First case: A player wins if they collect more than 81 stones in their accumulation hole.

Second case: A player also wins if their opponent reaches Atsyrau, in which case all remaining stones in the opponent's game holes are transferred to the player's accumulation hole.

## 3. Problem formulation

The difficulty of training an agent for the Togyzkumalak game can be described in terms of combinatorial complexity, game depth, and state-space dimensionality.

*3.1 Game tree size*

At the beginning of the game, a large number of possible moves significantly increase the size of the search tree.

In chess, the average branching factor is ≈ 35. In Togyzkumalak:

- at the start, there are 9 possible moves (choosing from one of the 9 holes);

- after a few moves, Tuzdyks appear, introducing new rules;

- in the middle of the game, there are typically 5–7 meaningful moves.

Although the number of possible branches in the initial position may be lower than in chess, it does not narrow significantly due to the complex rules of stone redistribution.

*3.2 Depth of play – length of matches*

Games in Togyzkumalak can last more than 100 moves, which is significantly longer than in chess, where the average game length is around 40–50 moves. In Togyzkumalak, the games often exceed 100 moves.

The average game length in Go is about 200-250 moves, but can vary greatly. If both players fight for every territory until the end, a game can extend beyond 300 moves.

Although Go games are much longer than those in Togyzkumalak, the branching factor (number of possible moves) is lower due to the game's territorial nature.

Deep games make deep learning more challenging because mistakes made early on may only become apparent 50–60 moves later.

*3.3 Determinism vs. stochasticity*

Togyzkumalak is a deterministic game, but the redistribution of stones after a move creates complex effects that are difficult to predict.

In chess, a player simply moves a piece. In Togyzkumalak, a single move can immediately redistribute 10+ stones.

This characteristic introduces a level of chaos, making position evaluation more challenging for a neural network. The same action can lead to completely different results depending on the distribution of stones on the board.

*3.4 Position evaluation*

In chess, piece differences can be used as a simple heuristic.

At the beginning of a Togyzkumalak game, material balance has little impact since both players have the same number of stones.

Key moments in the game revolve around Tuzdyk and Kazan management, which are harder to encode as a simple function. A trained neural network must understand strategic concepts such as:

- control over Tuzdyk
- advantageous stone redistribution
- defending and attacking the opponent's game hole.

Therefore, deep neural network architectures are needed to evaluate positions effectively.

*3.5 Self-play training problem*

At the beginning of training, the agent plays poorly and many games end randomly.

Compared to chess, where more natural patterns emerge (such as center control and piece development), mistakes in Togyzkumalak can appear random (e.g., losing a Tuzdyk early in the game).

As a result, significantly more self-play games are required before the agent begins to learn meaningful strategies.

## 4. Methods

While independently training agents for the Togyzkumalak game, two RL approaches can be used: model- free and model-based algorithms. The neural network architectures were selected on the basis of the experiment results. Since the representation of the board – the state of the game – differs between the methods, the descriptions are provided separately in Subsections 4.1 and 4.2.

*4.1 Deep Q-network*

The game board represents data on the number of stones in each game hole and the accumulation holes, as well as the number of Tuzdyks. Each game state is therefore described by a 23-dimensional vector, where the first 18 components represent the number of stones in the game holes, the 19th and 20th components represent the number of stones collected in the accumulation holes by the first and second players, respectively, and the 21st and 22nd components represent the numbers of holes won as Tuzdyks by each player. If no Tuzdyk has been captured, the value is set to -1. The last coordinate of the vector indicates the player's number.

In the game, the state space is discrete, a tensor of dimension 23 × 1, and the action space is also discrete, a tensor of dimension 9 × 1. According to the rules, a player cannot move from a hole that has no stones. Therefore, when implementing the game environment, action masking is used.

A game with a person requires displaying the state after each move. For this purpose, it is convenient to represent the state in graphical form Figure 3.

**Figure 3**
*Representation of the board in the starting position*

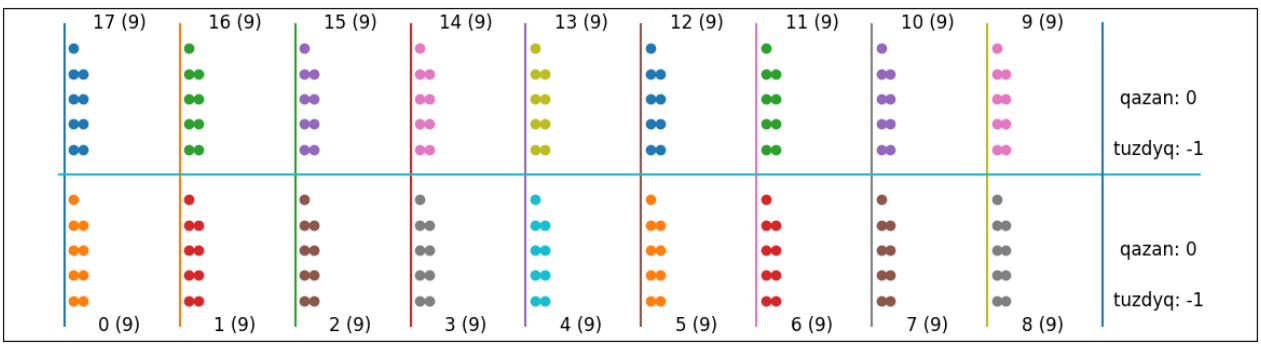
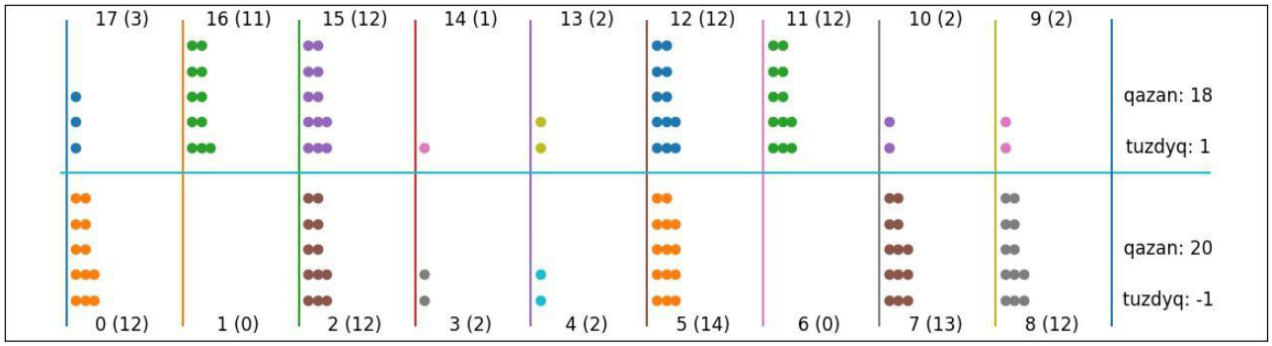


**Figure 4**
*Representation of the board during the game*




The environment for the Togyzkumalak game was implemented using the PettingZoo framework [17]. Figures 3 and 4 show graphical representations of the initial and playing positions of the game. The number of stones in the accumulation holes and the number of Tuzdyk cells are indicated on the right.

The Deep Q-Learning algorithm, also known as Deep Q-network [18] was used to train the agents by estimating the Q-value, which represents the expected reward. Thus, the Q-value corresponds to the expected reward for performing the selected action in the current state, following a particular policy. Deep Q-Learning uses a deep neural network to approximate the Q-values for each possible action-state pair. To optimize the learning process, a modified version, Double Deep Q-Learning [19] was used instead of the standard DQN.

For the Double DQN algorithm, it is very important to choose the most optimal neural network architecture. In the self-play technique described in Section 1, the weights are loaded from the previous model, so the choice must be made at the beginning.

The following MLP architectures were considered: NET ARCHS = [[128, 256, 256, 128], [256, 512, 256], [256, 512, 512, 256], [512, 1024, 512], [1024, 2048, 2048, 1024], [2048, 4096, 4096, 2048], [2048, 4096, 8192, 4096, 2048]].

ReLU activation function is used between the linear layers. The exploration and subsequent selection were carried out using neural network training with random policies as adversaries.

The results and details of the training are given in Subsection 6.1.

### *4.2 Monte Carlo Tree Search*

The Monte Carlo Tree Search (MCTS) algorithm was originally developed to train agents to play Go [12]. The AI, called AlphaZero Go, used a deep neural network. The algorithm was further developed in [20], and its application was extended to other games, such as Chess and Shogi. The neural network used in the algorithm allowed for a reduction in the number of roll-outs during gameplay by two orders of magnitude. Specifically, AlphaZero searches only 80,000 positions per second in Chess and 40,000 in Shogi, compared to 70 million for Stockfish [21] and 35 million for Elmo. AlphaZero compensates for the smaller number of evaluations by using its deep neural network to focus much more selectively on the best option. Since AlphaZero AI defeated the previous champions in each of the games it completed in (AlphaZero Go in Go, Stockfish and Elmo in Chess and Shogi respectively), it is currently considered the world champion.

AlphaZero was trained solely through self-play.

The MCTS algorithm in this study was applied to the Togyzkumalak game. The game board is implemented as in Figure 5. The first line indicates the number of stones in the players' accumulation holes, while lines 2 through 10 show the number of stones in the game holes. The left side belongs to the first player and the right side to the second player.

In the board representation, "o" denotes stones belonging to Bastaushy and "x" denotes stones belonging to Kostaushy. Uppercase letters ("O" and "X") represent ten stones each to reduce board size. This notation helps in visualizing the game state more compactly while maintaining clarity.

For ease of play with a human opponent, the number of stones and the game hole numbers are displayed on the edges of the board.

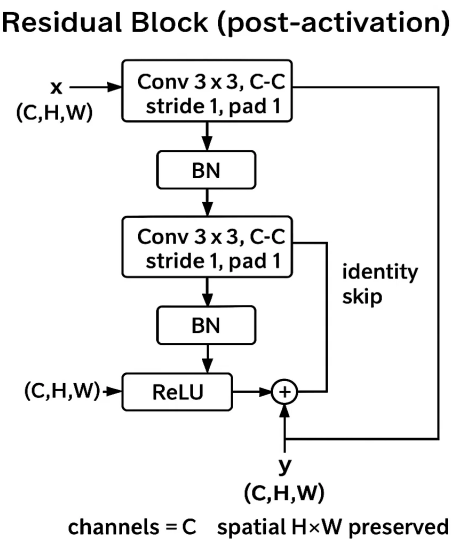
**Figure 5**
*Representation of the board in the starting position*

```
Turn  1 Player  1
============================================
000|0|--------------------------------|0|000
009|1|ooooooooo-------------xxxxxxxxx|1|009
009|2|ooooooooo-------------xxxxxxxxx|2|009
009|3|ooooooooo-------------xxxxxxxxx|3|009
009|4|ooooooooo-------------xxxxxxxxx|4|009
009|5|ooooooooo-------------xxxxxxxxx|5|009
009|6|ooooooooo-------------xxxxxxxxx|6|009
009|7|ooooooooo-------------xxxxxxxxx|7|009
009|8|ooooooooo-------------xxxxxxxxx|8|009
009|9|ooooooooo-------------xxxxxxxxx|9|009
============================================
Full board stones: 162.0
```

When Tuzdyk is captured, the cell is marked with a symbol representing the corresponding player. For example, in Figure 7, the Kostaushy has captured cell number 3.

During training, the board is presented as a 2D image of dimension boardx ×boardy (10 × 32) with one channel. The neural network follows the AlphaZero framework, utilizing a convolutional backbone with residual blocks and separate policy and value heads. During feature extraction, the input board state is processed by a convolution with batch normalization and ReLU activation. A sequence of residual blocks captures complex board dynamics.

**Figure 6**
*Basic residual block used in the MCTS network*




Each block shown in Figure 6 consists of:

- two convolutional layers with batch normalization;
- a skip connection adding the input to the output before ReLU activation.

The model features a dual-head architecture with separate outputs for policy and value estimation. Outputs action probabilities via:

- a convolution (128 channels) with batch normalization and ReLU;
- a fully connected layer mapping to the action space;
- log softmax activation for move probabilities.

Predicts board evaluation through:

- a convolution (64 channels) with batch normalization and ReLU;
- a fully connected layer (256 neurons);
- a final layer with tanh activation producing a value head.

This architecture balances expressiveness and efficiency, ensuring stable learning through residual blocks and separate optimization for policy and value functions.

The results obtained are described in Subsection 6.2.

**Figure 7**
*Representation of the board with Tuzdyk won by the Kostaushy*

```
Turn  51 Player  1
============================================
026|0|OOooooooo-----------------xxxXX|0|023
003|1|ooo-------------------------xxx|1|003
008|2|ooooooooo---------------------x|2|001
000|3|x-----------------------xxxxxx|3|006
001|4|o-------------------------xxxxx|4|005
002|5|oo-----------------------------xx|5|002
007|6|ooooooo--------------------xxxx|6|004
005|7|ooooo-----------------xxxxxxxXX|7|027
005|8|ooooo-----------------xxxxxxxxx|8|009
002|9|oo-------------------------xxxXX|9|023
============================================
Full board stones: 162.0
```

During training AlphaZero-style self-play pipeline is used. MCTS generates training tuples ($s_t$, $\pi_t$, $z$) where $s_t$ is the board state, $\pi_t$ is the MCTS policy (visit-count distribution) at step $t$, and $z \in \{-1, 0, 1\}$ is the game outcome from the current player's perspective. All tuples are stored in a replay buffer with a fixed capacity of 200 000 examples. When the buffer is full, the oldest examples are evicted.

During each training phase, mini-batches are drawn uniformly at random from the buffer (no prioritization). We also persist previously collected examples across runs to ensure continuity and reproducibility. Exploration follows the standard temperature schedule: MCTS policies are sampled stochastically up to move 20 and greedily thereafter. Search strength is controlled by 300 simulations per move.

Model selection is performed via an arena of 40 games between the new checkpoint and the incumbent, with each game starting from a randomized legal position. The challenger replaces the incumbent only if it wins at least 60% of games. Under this protocol, successive checkpoints consistently outperform earlier versions, evidencing a steady increase in playing strength. More details can be found in [22].

## 5. Development of tools

MARL is a diverse and highly active research area. With the introduction of deep learning in MARL, interest in these problems has grown significantly. As a result, publications and conference reports on this topic indicate the rapid development of the field.

Therefore, MARL researchers also require tools for conducting experiments and reproducing theoretical developments. Recently, numerous frameworks and interfaces have been developed for creating single- and multi-agent environments of various types. Naturally, there are also libraries for training agents, which are especially popular among robotics specialists, as RL is one of the most effective tools for training robots to perform specific tasks.

The source code is written using the frameworks and libraries described in this section and is available at [22].

*5.1 Framework for creating an environment*

PettingZoo [17] is a simple Pythonic interface designed to represent common RL and MARL problems. PettingZoo includes a wide range of reference environments, useful utilities, and tools to create custom user environments.

*5.2 Framework for training agents*

Tianshou [23] is a RL framework based on PyTorch. Tianshou provides a high-speed environment and a Pythonic API to build a deep reinforcement learning agent.

It is important to note that not all frameworks support MARL. For example, CleanRL [24], which is widely used in RL, is difficult to adapt for multi-agent environments. However, RLlib [25] is an excellent library for training agents in both single- and multi-agent environments, providing a wide selection of prebuilt algorithms and tools for customizing the training process.

*5.3 Library for training an agent based on the MCTS algorithm*

To train an AI to play the Togyzkumalak game, the Monte Carlo Tree Search algorithm was employed, which was previously used to train AlphaZero. The environment and training code were developed using the [26] library.

The implementation of the algorithm in this library is easily adaptable to any competitive board game. It is highly flexible and straightforward to implement self-contained reinforcement learning [27]. The library is designed for ease of use in any competitive two-player game and allows for building various DL neural network architecture.

## 6. Results

This section presents the main results of agent training using DQN and Monte-Carlo Tree Search algorithms. Although the two approaches have differences in the implementation of the environment, the rules of the game embedded in the environment remain consistent.

*6.1 Results of Deep Q-Learning*

For deep learning, it is necessary to select the neural network architecture, which is a non-trivial task as the success of training depends on it. In the study, the selection criterion was based on the results of the training of various architectures against a random policy. After training, all models played matches against this random policy, and the number of wins was a key factor in choosing the main architecture.

Here are the training details and the main characteristics of the computer resources:

- maximum number of epochs: 2000,
- number of steps per epoch: 1000,
- number of steps for testing: 50,
- batch size: 256,
- discount factor: 0.99.

The Adam optimizer with a learning coefficient of $lr = 3e - 04$ showed the highest results in obtaining a reward.

Details of the training process:

- number of hours: 65,

- GPU: P100 (Kaggle),
- CPU and GPU RAM: 1.5 gb and 243 mb.

The rewards statistics during training for the example of four architectures are shown in Figure 8. Analyzing the graph data, it can be stated that the efficiency of agent training is achieved with the selected training parameters.

Here, vertically, is the average reward value and horizontally, is the step number in each epoch.

To assess out-of-sample playing strength, we use two metrics: win rate against a random policy and win rate against the built-in opponent from The Togyz Qumalaq app at difficulty levels 1–4.

During testing, agents trained using the DQN algorithm were able to defeat only level 1 and level 2 computer opponents out of 4 difficulty levels. The trained policy exhibited short-sighted behavior, focusing on collecting as many stones as possible without considering the defense of game holes.

**Figure 8**
*Dynamics of change in received rewards during training against random policy*

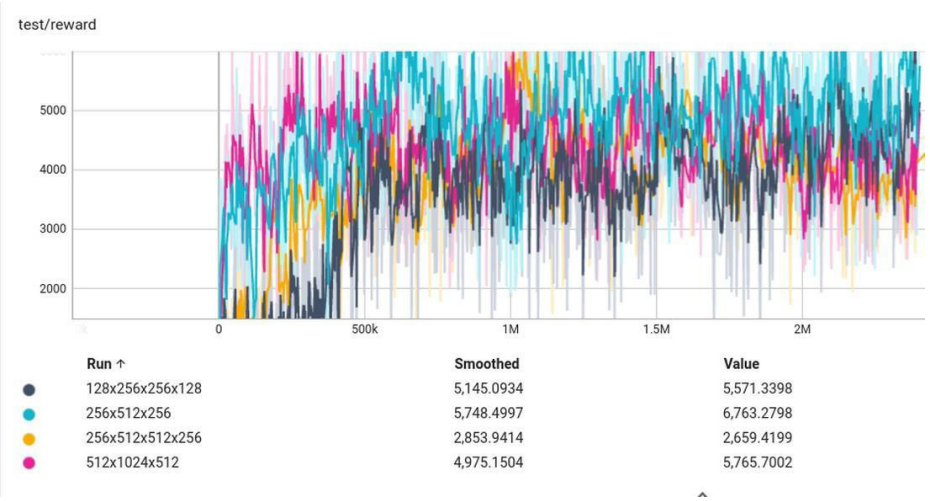
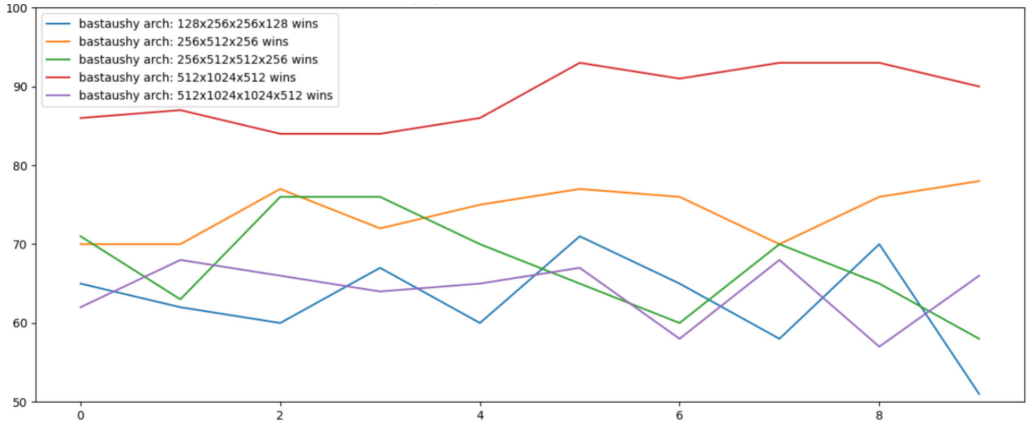


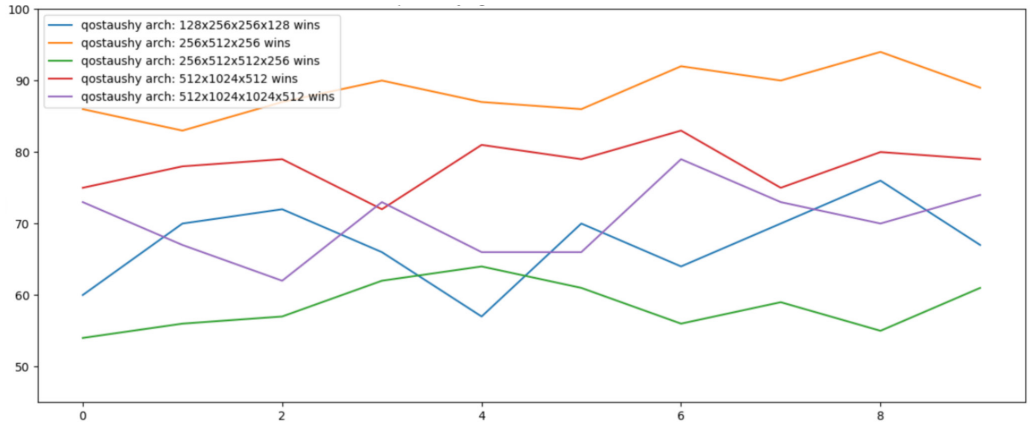

Figure 9 shows win rates of trained agents with five network architectures evaluated against a random policy, separately for the first move (Bastaushy) and the second move (Qostaushy). The 512–1024–512 MLP consistently dominates: as Bastaushy it stays around the 88–93% band across evaluation rounds (Fig. 9), and as Qostaushy it remains in the 78–85% band with peaks above 90% (Fig. 10).

**Figure 9**
*Bastaushy move. Evolution of win rate against a random opponent for different architectures*



Smaller models (e.g., 128–256–256–128 or 256–512–256) plateau between ~60% and ~78%. These results guided this choice of 512–1024–512 as the default architecture for subsequent experiments.

**Figure 10**
*Qostaushy move. Evolution of win rate against a random opponent for the same architectures*




*6.2 Results of Monte Carlo Tree Search algorithm*

Overcoming challenges described in Section 3 required finding a suitable representation of the board and constructing a customized neural network architecture for the algorithm.

The neural network architecture is described in Subsection 4.2. We will provide the training details and describe the implementation using the self-play technique.

During the experiments, the optimal number of residual blocks was found to be 30. One of the key training parameters was numMCTSSims, which determines the number of game moves simulated by MCTS. A trade-off study between execution time and stable learning through self-play resulted in a value of 300. A fixed learning rate did not yield good results, so the training utilized a CosineAnnealingLR scheduler and a momentum SGD optimizer. More details on the training process can be found in [22].

Main training parameters:

- number of channels in layers: 256,
- number of residual blocks: 30,
- learning rate: 0.001 → 0.0001,
- dropout probability: 0.3,
- number of epochs: 20,
- mini-batch size: 64.

The network consists of an input stem Conv3×3 (1→256) + BN, a stack of 30 residual blocks, a policy head Conv1×1 (256→128) + BN + FC ($128 \times board_x \times board_y \rightarrow 9$), and a value head Conv1×1 (256→64) + BN + FC ($64 \times board_x \times board_y \rightarrow 256$) + FC ($256 \rightarrow 1$). Here, BN denotes Batch Normalization and FC denotes a fully connected (linear) layer with bias. The model contains 41,100,362 trainable parameters (weights + biases + BN affine).

During training, the open source library [27] was used, which is available at [26]; more details are given in Subsection 5.3. After training, the model plays several matches against the previously version of itself; if > 60% wins, a new model is accepted; otherwise, it is rejected.

Details of the training process:

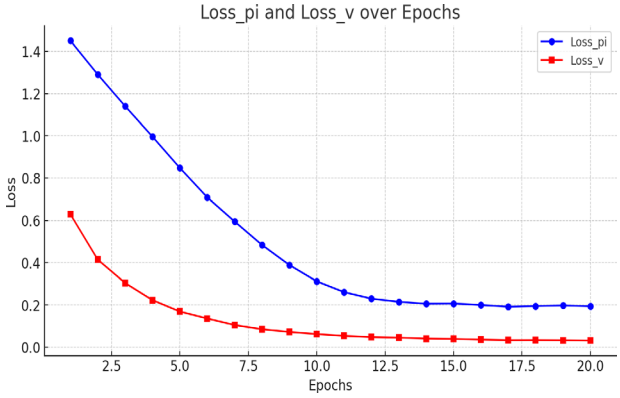
- number of hours: 172,
- GPU: GeForce RTX 4060 Max-Q / Mobile,
- CPU and GPU RAM: 16 gb and 8 gb.

The dynamics of dual head loss during most iterations followed a consistent pattern as in Figure 9. Trained agents can still win against novices and a random policy; however, to outperform professional players, more computing resources are required for training and experimentation. Unfortunately, there was insufficient memory and processing power for a wider or deeper network, which could be crucial to defeating a professional player.

However, the training dynamics in the first dozens of iterations indicated that this neural network has the potential to become a champion in Togyzkumalak. As the number of training examples increased, the agent consistently outperformed its previous versions, demonstrating a steady improvement in its playing strength.

Training dynamics of the MCTS policy – value network is shown in Figure 11. Policy loss (blue; cross-entropy to MCTS visit distribution) and value loss (red; MSE to game outcome) versus training epoch. The value loss converges quickly and stabilizes near 0.04; the policy loss decreases steadily and plateaus around 0.19 after ≈14 epochs, indicating overall convergence of the network.

**Figure 11**
*Dual-head loss dynamics during one iteration*




To assess how the convolutional policy – value network used with MCTS translates into practical playing strength, we evaluated the final checkpoint against The Togyz Qumalaq mobile app. The MCTS agent consistently defeated levels 1–3 (as Bastaushy), while level 4 remains an open challenge. For context, the best DQN baseline reliably beats only levels 1–2, reflecting the limits of a flat MLP without spatial inductive bias.

## 7. Discussion and future work

The main goal of this work was to develop trained agents that play the Togyzkumalak game effectively. To achieve this, a multi-agent environment was prepared, and research work was carried out to design a reward system, select the appropriate architecture, and choose a learning algorithm.

In addition to the algorithms described in Section 4, experiments were carried out with other methods, such as Proximal Policy Optimization and Rainbow. However, based on the comparison of results from these experiments, the Deep Q-Learning and Monte Carlo Tree Search algorithms were selected.

While DQN is effective in many control and Atari-like benchmarks, it is not ideal for combinatorial board games with sparse rewards and long-horizon tactics. Accordingly, we treat DQN as a baseline and adopt a planning-based approach (policy-value networks with MCTS-guided self-play), which enables the model to discover board patterns and apply state-specific policies with deep lookahead from each position.

The AlphaZero algorithm uses information about the game's specifics. For example, the main property of all the listed board games (Go, Chess, Shogi and Togyzkumalak) is symmetry, i.e. the board can be symmetrically reflected along the main axis; in the case of the Togyzkumalak game, this is the vertical axis. In future research, there is interest in applying the methods used in the training of MuZero AI [16].

This research achieved promising results in practical applications.

**Acknowledgments**

The author would like to thank the Institute of Mathematics and Mathematical Modeling for their technical and organizational support during this study.

**Funding**

This research has been funded by the Science Committee of The Ministry of Science and Higher Education of the Republic of Kazakhstan (Grant No. AP23488302).

**Conflicts of Interest**

The author declares no conflict of interest.

**References**

1. R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed., ser. Adaptive Computation and Machine Learning. Cambridge, MA, USA: MIT Press, 2018.
2. S. V. Albrecht, F. Christianos, and L. Schäfer, *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. Cambridge, MA, USA: MIT Press, 2024.
3. K. Zhang, Z. Yang, and T. Başar, "Policy optimization provably converges to Nash equilibria in zero-sum linear quadratic games," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

4. C. Daskalakis, D. J. Foster, and N. Golowich, "Independent policy gradient methods for competitive reinforcement learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, pp. 5527–5540.

5. C.-Y. Wei, C.-W. Lee, M. Zhang, and H. Luo, "Last-iterate convergence of decentralized optimistic gradient descent/ascent in infinite-horizon competitive Markov games," in *Proc. 34th Conf. Learning Theory (COLT)*, ser. Proceedings of Machine Learning Research, vol. 134, M. Belkin and S. Kpotufe, Eds., Aug. 2021, pp. 4259–4299.

6. J. von Neumann, "Zur Theorie der Gesellschaftsspiele," *Mathematische Annalen*, vol. 100, no. 3, pp. 295–320, 1928.

7. J. von Neumann, O. Morgenstern, and A. Rubinstein, *Theory of Games and Economic Behavior* (60th Anniversary Commemorative Edition). Princeton, NJ, USA: Princeton University Press, 1944.

8. R. J. Vanderbei, *Linear Programming: Foundations and Extensions*. Cham, Switzerland: Springer, 2020.

9. M. Bowling and M. Veloso, "Multiagent learning using a variable learning rate," *Artificial Intelligence*, vol. 136, no. 2, pp. 215–250, Apr. 2002.

10. B. Banerjee and J. Peng, "Performance bounded reinforcement learning in strategic interactions," in *Proc. 19th National Conf. Artificial Intelligence (AAAI)*, San Jose, CA, USA, 2004, pp. 2–7.

11. R. Powers and Y. Shoham, "Learning against opponents with bounded memory," in *Proc. 19th Int. Joint Conf. Artificial Intelligence (IJCAI)*, San Francisco, CA, USA, 2005, pp. 817–822.

12. D. Silver, J. Schrittwieser, K. Simonyan, *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, pp. 354–359, 2017.

13. T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2018.

14. E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, 2012, pp. 5026–5033.

15. C. Berner, G. Brockman, B. Chan, *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

16. J. Schrittwieser, I. Antonoglou, T. Hubert, *et al.*, "Mastering Atari, Go, chess and shogi by planning with a learned model," *Nature*, vol. 588, pp. 604–609, 2020.

17. J. K. Terry, B. Black, A. Hari, *et al.*, "PettingZoo: Gym for multi-agent reinforcement learning," *arXiv preprint arXiv:2009.14471*, 2020.

18. V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing Atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

19. H. van Hasselt, "Double Q-learning," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 23, 2010.

20. D. Silver, T. Hubert, J. Schrittwieser, *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

21. T. Romstad, M. Costalba, and J. Kiiski. "Stockfish chess engine." Accessed: Mar. 27, 2025. [Online]. Available: https://stockfishchess.org/

22. zhus-dika. "Togyzkumalak agent." Accessed: Mar. 27, 2025. [Online]. Available: https://github.com/zhus-dika/togyz-qumalaq-agent

23. Tianshou Team. "Tianshou: A library for deep reinforcement learning in PyTorch." Accessed: Mar. 27, 2025. [Online]. Available: https://tianshou.org/en/stable/index.html

24. S. Huang, R. F. J. Dossa, C. Ye, J. Braga, D. Chakraborty, K. Mehta, and J. G. M. Araújo, "CleanRL: High-quality single-file implementations of deep reinforcement learning algorithms," *Journal of Machine Learning Research*, vol. 23, no. 274, pp. 1–18, 2022.

25. Ray. "RLlib: Scalable reinforcement learning." Accessed: Mar. 27, 2025. [Online]. Available: https://docs.ray.io/en/latest/index.html

26. S. Nair. "Alpha-Zero-General implementation." Accessed: Mar. 27, 2025. [Online]. Available: https://github.com/suragnair/alpha-zero-general

27. S. Thakoor, S. Nair, and M. Jhunjhunwala, "Learning to play Othello without human knowledge," Stanford Univ., Stanford, CA, USA, Tech. Rep., 2016.

***Information about the authors:***

*Dinara Zhussupova – PhD. Dr. Dinara Zhussupova is a Leading Researcher at the Institute of Mathematics and Mathematical Modeling. She received her PhD in Mathematics from L.N. Gumilyov Eurasian National University. Her academic and research work focuses on mathematical modeling, numerical methods, artificial intelligence, and reinforcement learning. She has experience in interdisciplinary research at the intersection of mathematics, computer science, and intelligent systems (Almaty, Kazakhstan, e-mail: zhus.dinara@gmail.com. ORCID iD: 0000-0002-1860-0843).*

*Submission received: 10 April, 2025*
*Revised: 17 February, 2026*
*Accepted: 17 February, 2026*