Int. J. Adv. Eng. Pure Sci. 2025, 37(1): <27-35>
DOI: 10.7240/jeps.1620319

RESEARCH ARTICLE / ARAŞTIRMA MAKALESİ

Adapting the AlphaZero Algorithm to Pawn Dama: Implementation, Training, and Performance Evaluation

Erdem PEHLİVANLAR [1], Alperen GÖNÜL [1], Cem GÜLEÇ [1], Muhammet ŞERAMET [1], Mehmet Kadir BARAN [1]

[1]Marmara University, Faculty of Engineering, Computer Science Departmant, İstanbul, Turkey

Abstract

This research uses deep reinforcement learning techniques, notably the AlphaZero algorithm, to construct an artificial intelligence system that can play Pawn Dama at a level that surpasses human players. Pawn dama, a simplified variant of Dama, is a perfect platform to explore AI's ability to think strategically and make decisions. The primary goal is to develop an AI that can use self-play to develop sophisticated strategies and comprehend the game's dynamics and regulations. The project incorporates MCTS to improve decision-making during games and uses a Convolutional Neural Network (CNN) to enhance the AI's learning capabilities. Creating an intuitive graphical user interface, putting the reinforcement learning algorithm into practice, and testing the system against real players are steps in the development process. The accomplishment of this project will contribute to the field of strategic game AI research by providing insights that may be applied to other domains and spurring further advancements in AI-driven game strategies.

Keywords: Deep Reinforcement Learning, Deep Learning, AlphaZero Algorithm, Pawn Dama, Monte Carlo Tree Search (MCTS), Convolutional Neural Network (CNN)

I. INTRODUCTION

Board games provide a good platform for AI research due to their controlled settings, which facilitate examining strategic thinking. AlphaZero algorithm, introduced by Deepmind in 2017, has profoundly transformed AI's approach to board games. In contrast to the previous approaches, the AlphaZero algorithm achieved a high level of competence in playing pawn dama without prior domain knowledge, relying solely on self-play and general-purpose learning. Its success showcased the potential of reinforcement learning combined with neural networks and Monte Carlo Tree Search, setting a new standard for AI in strategic decision-making and game theory.

This project aims to adapt the AlphaZero algorithm to Pawn Dama, a simplified version of Turkish Dama. The rules for Pawn Dama are given in the Appendix. The AI will learn the rules and strategies of Pawn Dama through self-play and extensive training, ultimately reaching a high level of competence in playing Pawn Dama. For this purpose, it is necessary to rewrite the two core components of the AlphaZero algorithm, neural networks, and the Monte Carlo Tree Search (MCTS) for Pawn Dama. A snapshot of the game is given in Figure 1 and Figure 2.

Corresponding Author: MEHMET KADİR BARAN, **Tel: :** +90 532 243 85 49 **, E-mail:** crom.baran@gmail.com

Submitted: 15.01.2025, **Revised:** 14.02.2025, **Accepted:** 20.02.2025

Figure 1. View of the Game

Figure 2. View of the Game

1.1. Aims of the Project

This research has two primary objectives: To develop a simple framework for applying the AlphaZero algorithm tailored to limited computational budgets and to lay the groundwork for the eventual full implementation of Turkish Dama.

By accomplishing these objectives, the project also aims to contribute to the broader area of AI research by showcasing the use of modern AI methods in culturally significant games and encouraging further advancements in AI-driven strategic gaming.

The main technical contributions of this article can be collected under two headings:

- The adaptation of the CNN to pawn dama and the design of its action space (Sect. 3.1.3).
- Developing an MCTS algorithm for pawn dama. (Sect. 3.1.1)

In summary, the inclusion of a new game in AlphaZero's repertoire enhances its coverage and applicability. This game presents a complexity level that is intermediate between simple games like Tic-Tac-Toe or Connect Four and computationally demanding games such as Chess or Go thus filling a gap in the spectrum of games suitable for training.

II. RELATED WORK

2.1 Standard Reinforcement Learning Algorithm

AlphaZero uses a modified version of the Reinforcement Learning algorithm (RL), a key method to train an agent in decision-making within a given environment. An environment is composed of numerous states, and at every moment, the agent takes actions based on the environmental factors. The quality of each action is determined by the reward it generates, prompting the agent to adjust its strategy to maximize cumulative rewards. One classical RL algorithm is Q-learning, which relies on creating a Q-table. The rows of this table represent possible environmental states, while the columns reflect potential actions and the rewards they produce. As the agent interacts with its environment, it updates the Q-table with reward values. Eventually, the agent can use the completed table to choose actions that maximize its rewards in any given scenario. However, despite its effectiveness in specific scenarios, Q-learning has limitations when applied to more complex environments. For example, in chess, the number of possible positions is approximately 10^{120}. Completing and storing a Q-table for such a vast number of states is impractical. The development of deep reinforcement learning algorithms, such as AlphaZero, overcame the limitations of classical reinforcement learning and enabled superhuman performance in chess.

2.2 Deep Reinforcement Learning on Strategy Games

Until the advent of deep RL learning, board game-playing algorithms were mainly based on minimax algorithms enhanced by alpha-beta pruning [1, 2]. This approach peaked in Deep Blue for chess [3] and in Chinook for checkers [4], which achieved superhuman performance by successfully defeating the reigning world champions.

Starting in 1990, Reinforcement Learning [27] made its appearance in board game-playing algorithms with algorithms like TD-Gammon [5, 6].

The introduction of MCTS revolutionized bard game-playing algorithms by using random simulations to estimate the potential outcomes of moves. [7, 8] This method balanced exploration and exploitation without exhaustive search and was applied to Go within a year of its development [9] and then to Kriegspiel in 2010 [10]. For a survey of the applications of MCTS for game playing, see [11].

Finally, in 2016, a synergy between MCTS-based RL methods and neural network-based deep learning methods was achieved in the Alphago algorithm by Deepmind. [12]. The game of Go, long considered a formidable challenge to AI, was first conquered by Alphago, making it the first program to beat a professional Go player.

AlphaGo initially trained its neural network on a large dataset of games played by human experts. After this supervised learning phase, AlphaGo trained its neural net further through self-play**.**

The following algorithm from DeepMind, AlphaGo Zero (2017) [13], eliminated the use of human knowledge and trained its neural network solely through self-play, therefore learning strategies and tactics from first principles.

Alphago Zero was explicitly developed for the game of Go. The next algorithm of Deepmind, Alphazero (2017) [14], was a more generalized version of Alphago zero. It was capable of mastering multiple games (e.g., Go, chess, shogi) without any domain-specific modifications, thus demonstrating that the same algorithm can achieve superhuman performance across different environments.

Deepmind published two more algorithms: Alphastar (2019) [15] and Muzero (2020) [16]. Alphastar was developed for real-time strategy games like StarCraft II and achieved superhuman performance. MuZero is a deep reinforcement learning algorithm that combines a model-based approach with policy and value networks, learning both a model of the environment and optimal strategies without relying on prior knowledge of the environment's dynamics. Alphastar and Muzero are not directly relevant to our research and are mentioned here solely for the purpose of completeness. For more recent developments, see [23-26]

III. METHOD

The primary approach employed in developing an algorithm capable of playing pawn dama involved tailoring the AlphaZero algorithm specifically to this game. AlphaZero algorithm combines reinforcement learning with Monte Carlo Tree Search (MCTS) and neural networks to excel in strategic games such as Chess, Go, and Shogi. Unlike traditional engines, it learns solely through self-play, starting without pre-existing game knowledge. AlphaZero algorithm has two components: (1) A neural network embedded within the AlphaZero evaluates board positions and predicts optimal moves, while (2) MCTS efficiently explores possible outcomes by simulating future states of the game. As AlphaZero plays, its strategies are continuously improved by adjusting its neural network parameters based on game results. In the end, it achieves superhuman performance.

3.1 Alpha-Zero on Pawn Dama

In the following sections, we detail the modifications we have made to the AlphaZero algorithm to adapt it to pawn dama. As previously noted, the AlphaZero algorithm comprises two key components: Monte Carlo Tree Search (MCTS) and a neural network. We will briefly describe the functionality of these components and outline the adjustments necessary to tailor them for pawn dama.

3.1.1. Monte Carlo Tree Search (MCTS)

In its classical form, Monte Carlo Tree Search (MCTS), is an artificial intelligence algorithm designed to determine the optimal move based on the current game state without incorporating any learning mechanisms. It operates through four fundamental steps in each simulation iteration:

a) Selection: Starting from the root node, the algorithm traverses the tree by selecting child nodes based on a selection policy, often the **Upper Confidence Bound (UCB)** formula.

$$UCB(i) = \frac{W_i}{N_i} + C\sqrt{\frac{ln\, N_p}{N_i}},$$

where N_i is the visit count of the node i, N_p is the visit count of its parent, and W_i is the total reward accumulated in node i at the backpropagation steps. C is a hyperparameter that balances exploration with exploitation. When traversing down the tree, the child node with the highest UCB score is selected.

b) Expansion: Once a promising node is selected, the algorithm checks if it has unexplored child nodes. If so, it expands the tree by adding one or more child nodes to represent possible moves.

c) Simulation: From the newly expanded node, the algorithm plays a simulation (also called a "playout") by performing random moves until the game reaches a terminal state (e.g., a win, loss, or draw).

d) Backpropagation: The simulation result is propagated back through the path of nodes leading to the root. [8]

The four steps of MCTS are illustrated in Figure 3.

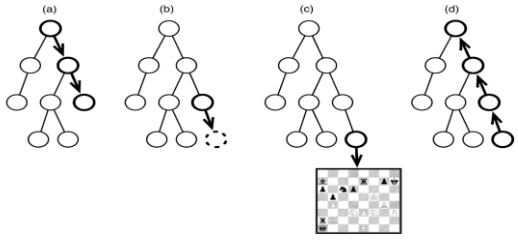

Figure 3. MCTS Structure, from [10]

These four steps are repeated for a predefined number of iterations or until a computational budget (e.g., time or resource limit) is reached, after which the algorithm determines the best move to make from the root node based on the data collected during the search.

AlphaZero makes two key differences from the classical MCTS algorithm described in Figure 3:

- It does away with step 3 (simulation), replacing the value returned from the simulation with the value vector of the neural network. With this update, rather than simulating the game randomly, the value vector of the CNN model is used to predict the actual result of the game.

- Another update is applied to the UCB formula, where an additional factor is introduced: the winning probability $P(i)$ for node i. This new factor serves as a multiplier in the UCB formula, helping the algorithm make better decisions when selecting moves.

$$\frac{W_i}{N_i} + CP(i)\frac{\sqrt{N_p}}{1+N_i}$$

P(ı) is ith component of the policy vector of the ith node's parent. In our work, the constant C, which balances exploration with exploitation, is taken as 1.

Policy and value vectors of a neural network will be explained in the next section.

In this study, we adapted the Monte Carlo Tree Search (MCTS) algorithm as it is used by AlphaZero to the specific rules of pawn dama. This adaptation was straightforward because pawn dama lacks repetition, as each move produces a unique board configuration; otherwise, additional measures would have been necessary.

3.1.2. Convolutional Neural Network (CNN)
The algorithm uses a convolutional neural network (CNN) [20]. It takes 3 layers of information as input: board size, whose turn it is, and a representation of the game board. It produces two outputs: a policy and a value vector, which are used in MCTS. For an NxN input board, the policy vector, output by the policy head, describes the winning probability of each possible legal move in the space of all possible legal moves, the action space. If the CNN generates probabilities for some illegal moves, we mask them by clearing their corresponding probabilities, ensuring that only legal moves are considered in the decision-making process. The representation chosen for the action space is critical and will be explained in detail in the next subsection. The value vector is a prediction of the outcome of the given game board. Once the model training is complete, the algorithm returns a value of 1, 0, or -1, indicating a loss, draw, or win. The Convolutional Network (CNN) structure used is illustrated in Figure 4.

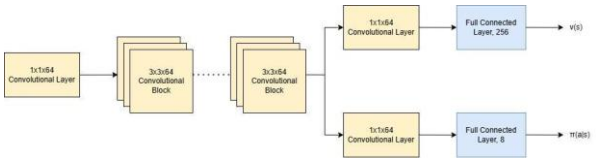

Figure 4. Two-headed Convolutional Neural Network (CNN) Architecture, from [18]

Table 1. Layer details of CNN used

Layer	Type	Input Shape	Output Shape	Parameters
conv1	Conv2D	(B, 1, X, Y)	(B, C, X, Y)	3x3 kernel, stride=1, padding=1
bn1	BatchNorm2D	(B, C, X, Y)	(B, C, X, Y)	-
conv2	Conv2D	(B, C, X, Y)	(B, C, X, Y)	3x3 kernel, stride=1, padding=1
bn2	BatchNorm2D	(B, C, X, Y)	(B, C, X, Y)	-
conv3	Conv2D	(B, C, X, Y)	(B, C, X-2, Y-2)	3x3 kernel, stride=1, no padding
bn3	BatchNorm2D	(B, C, X-2, Y-2)	(B, C, X-2, Y-2)	-
conv4	Conv2D	(B, C, X-2, Y-2)	(B, C, X-4, Y-4)	3x3 kernel, stride=1, no padding
bn4	BatchNorm2D	(B, C, X-4, Y-4)	(B, C, X-4, Y-4)	-
flatten	Reshape	(B, C, X-4, Y-4)	(B, C*(X-4)*(Y-4))	-
fc1	Linear	(B, C*(X-4)*(Y-4))	(B, 1024)	-
bn_fc1	BatchNorm1D	(B, 1024)	(B, 1024)	-
fc2	Linear	(B, 1024)	(B, 512)	-
bn_fc2	BatchNorm1D	(B, 512)	(B, 512)	-
fc3 (Policy)	Linear	(B, 512)	(B, action_size)	-
fc4 (Value)	Linear	(B, 512)	(B, 1)	-

Below are key notations used in Table
- B: Batch size (64)
- C: Number of channels (512)
- X, Y: Board dimensions

We applied ReLU activation and dropout with a rate of 0.3 after each BatchNorm. Finally policy vector is transformed by log_softmax function and value prediction is scaled using tanh activation.

3.1.3. Action Space
Action space is the set of all possible actions (moves) that can be taken in a given state of the game. It is a fundamental concept of reinforcement learning, and its design defines CNN's policy vector.

The rules of Pawn Dama allow each piece to have three possible moves after it has already moved and four possible moves during its first move.

Consequently, for an NxN board, an action space of dimension $4 * 2N + 3(N-2)^2$ would be sufficient to describe all possible moves. However, after some consideration, it was decided to use an action space of dimension $N^4 = N^2 * N^2$ instead.

In this design, the first N^2 represents the piece's initial position, and the second N^2 is the target position. A move is thus depicted as $(x_1,\ y_1) \rightarrow (x_2,\ y_2)$, meaning there are four indices in total to specify each action. This description of the action space, while having higher dimensionality, provides the flexibility to incorporate pieces with arbitrary movement rules in future versions of the algorithm without significantly changing the structure of CNN, thereby increasing its adaptability.

After discussing the dimensionality of the action space, it becomes evident that the policy vector is a single vector of dimension N^4. Each potential move $(x_1, y_1) \rightarrow (x_2, y_2)$, on the board is represented by a single element of this vector determined by the following index:

$$x_1 + (y_1 * N) + (x_2 * N^2) + (y_2 * N^3)$$

The output of the CNN stores the winning probability of the corresponding move into this element.

3.1.4. Self-Play and Training

After these modifications to MCTS and CNN components of the AlphaZero algorithm are made, the training process begins. Initially, the CNN is initialized with random weights. The algorithm then engages in 100 MCTS self-play games, experiencing wins, losses, and draws. During this competition, every position encountered is recorded. The recorded data consists of a triplet: board position, which move is made, and which side eventually won the game. This accumulated data is then used to train the CNN model for 100 epochs. Once training is completed, the process moves to the arena phase, where each new model plays 30 games against the previous model. If the new model's win threshold fraction is 60% higher than the previous one, it is accepted; otherwise, it is rejected. These three stages (MCTS, CNN training, and Arena play) are called an iteration. Numerous iterations were carried out in a loop. The algorithm is set up to run for 300 iterations. But during a run, we follow the loss value and stop the algorithm manually when it stabilizes around zero. Figure 5 is the flowchart of the AlphaZero algorithm.

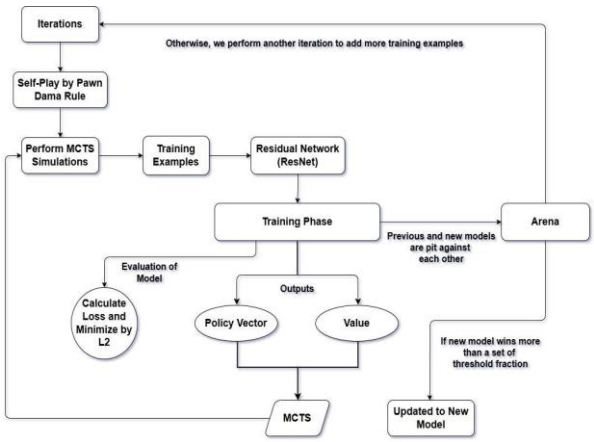

Figure 5. Flow of self-play based on the game rules

IV. EXPERIMENTS

4.1. Overview

We evaluated the performance of our algorithm using three different methods. First, we tracked the loss values at the end of each epoch during the training process. At the conclusion of the training, we visualized these values in a graph to analyze the model's learning progress and overall development. Second, at the end of each iteration, we tested the newly trained model against the previous version in 30 matches. We observed how the model improved over iterations by analyzing the number of wins, losses, and draws. Finally, we tested the trained model against approximately 100 human players through an interactive interface. Remarkably, no human player (some of them competitive chess players) managed to defeat the model. These three approaches provided a comprehensive understanding of our algorithm's training process and performance in real-world scenarios.

4.2 Decrease in Training Error

To evaluate our model's learning progression, we analyzed the training errors for two different board sizes: 5x5 and 6x6. We tracked each configuration's policy vector and value losses throughout the training process. The training was conducted on NVIDIA A100 and NVIDIA GeForce RTX 3050 GPUs, with losses recorded at the end of each epoch. The following subsections present detailed analyses for each board size.

4.2.1. 5x5 Board Experiments and Results

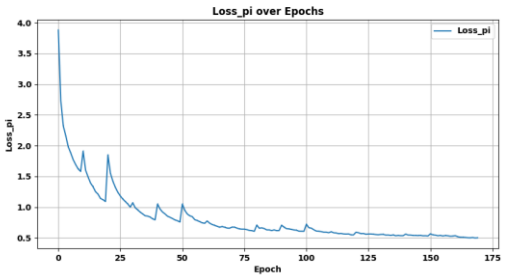

Figure 6. 5x5 Board Policy Vector Losses

Figure 6 shows the decrease in the loss of the policy network over 175 epochs in a 5x5 board. This graph was essential for understanding the learning process of the AlphaZero algorithm when applied to our Pawn Dama game. The plot indicates that the training process for the policy network was successful. The rapid initial decrease in training losses, followed by a gradual and stable convergence, shows that the model effectively learned to make policy predictions. The overall performance suggests that the model was well-trained and capable of generalizing its learned strategies to new game scenarios.

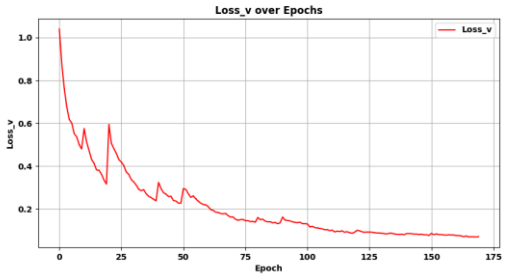

Figure 7. 5x5 Board Value Losses

Figure 7 shows the decrease in the loss of the value network over 175 epochs for a 5x5 board. Looking at the graph, we can tell that the training process was successful for the value network. We saw a sharp decrease in training loss at the beginning, followed by a steady and stable drop, which suggests that the model learned to make good value predictions.

The spikes in the loss function of both graphs indicate points at which one iteration finishes and a new iteration starts (See 3.1.4). Each iteration brings its own training set, and we can assume that these new training sets bring some new information (i.e., previously unseen game situations) that is not contained in the old training sets. At that point, the neural net, solely trained by the old training sets, struggles to handle this new information, hence the sudden jump in error. However, in time, it absorbs this new information into its weights via backpropagation, and the errors drop. Also note that as the game progresses, it becomes harder to surprise neural networks, and the size of the spikes decreases.

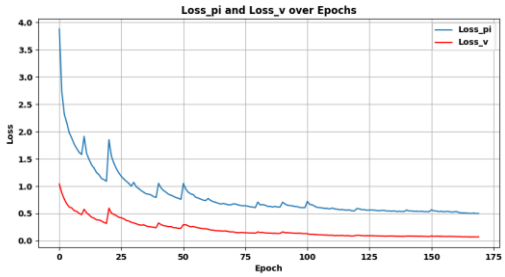

Figure 8. 5x5 Board Policy Vector and Value Losses

As shown in Figure 8, we display both the policy vector and value losses on a single graph for the 5x5 board. This side-by-side presentation helps visualize the learning progress of the policy and value networks simultaneously. The training was conducted on an NVIDIA A100 GPU, a high-performance computing unit known for handling complex deep-learning tasks. The model was trained for approximately 18 hours, allowing it to reach an effective level of performance. This training duration was sufficient to capture the essential patterns and nuances of the game while ensuring the model had enough time to refine its learning.

4.2.2. 6x6 Board Experiments and Results

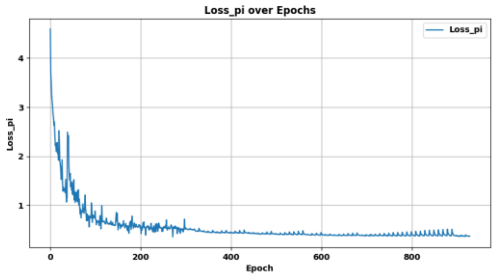

Figure 9. 6x6 Board Policy Vector Losses

Figure 9 shows the policy loss for a 6x6 board, helping us understand how the model learns to select moves in the pawn game. Here, the convergence is slower than on the 5x5 board. The loss values were relatively high initially but dropped quickly during the first 100 epochs. By reaching 600 epochs, the loss had settled at its minimum level and remained steady. The reasons for the spikes are the same as discussed for the 5x5 board. The consistent decline and eventual stabilization of the loss values indicate that the model successfully learned the key strategies of the game.

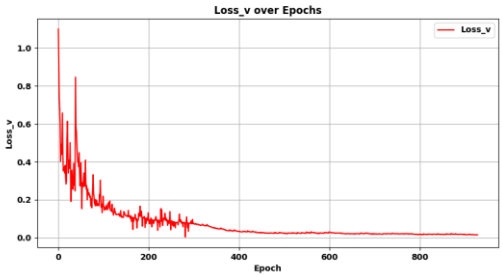

Figure 10. 6x6 Board Value Losses

In Figure 10, the value loss curve shows how the model improved its ability to evaluate positions in the 6x6 pawn game. Again, the loss was high initially but dropped quickly during the first 100 epochs. This rapid decrease tells us that the model learned the basics of position evaluation early in the training. After this point, the loss continued to decline more gradually, showing that the model was fine-tuning its understanding of more complex situations.

By the 1000th epoch, the loss reached a stable and low value, staying consistent for the rest of the training. Unlike the policy loss, we do not see any significant spikes here, which shows that the training process for the value network was smooth and reliable.

The final result confirms that the model successfully learned to evaluate positions accurately, predicting game outcomes with confidence. This stability tells us that once the core ideas of position evaluation were understood, the model could consistently apply them to new scenarios, showing a clear understanding of pawn structures and their impact on gameplay.

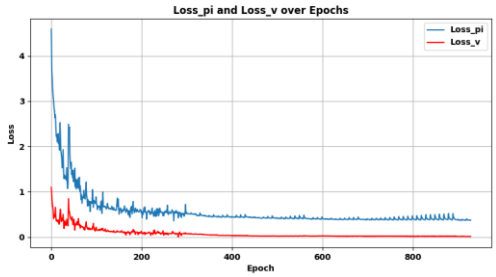

Figure 11. 6x6 Board Policy and Value Vector Losses

Figure 11 presents the policy vector and value losses for the 6x6 board. By displaying both losses simultaneously, we can assess the model's performance in both areas more efficiently. The training process required significant computational power, and we used an NVIDIA A100 GPU for this task. The training for the 6x6 board took approximately 48 hours. This extended training period allowed the model to learn more complex strategies and game dynamics effectively.

4.3 Self-Play Results

In this section, we discuss the results obtained from the self-play games, a critical step in the training process of our model. Self-play allowed the model to improve iteratively by competing against itself, refining its strategies, and correcting mistakes over time. The model was set to play 40 matches throughout training at the end of each iteration, where the newly trained version competed against the model from the previous iteration. The new model consistently improved and outperformed its predecessor as the training progressed. However, in the later stages, the matches between iterations consistently resulted mostly in draws, suggesting that the model had achieved its maximum potential and was operating at an optimal level, unable to further improve against itself. These findings highlight the model's ability to reach a high level of performance autonomously through self-play. This iterative process not only ensured strategic improvement but also validated the robustness of the training methodology.

4.4 Playing Against Humans

To evaluate our model's performance in real-world scenarios, we conducted tests by allowing humans to play directly against the trained AI. Using the interactive interface we developed, over 100 games were played against various individuals with varying experience levels, including skillful players who frequently play chess at well-known online platforms such as lichess or chess.com. Impressively, the AI model remained undefeated throughout these matches, consistently demonstrating its ability to adapt to human strategies. This experience highlighted the AI's strategic depth and robustness, as it effectively handled diverse human gameplay styles. Additionally, feedback from the participants revealed that the AI not only played competently but also provided a challenging and engaging experience. These results confirmed the strength of the model and its capability to perform reliably outside of controlled testing environments, marking a significant milestone in its development.

V. CONCLUSIONS

In this study, we ran AlphaZero on a new game that had never been explored. By integrating Monte Carlo Tree Search (MCTS) with neural networks, we adapted AlphaZero to this novel environment. The updates to the MCTS algorithm, particularly in the simulation phase and UCB formula, allowed for more effective data generation and evaluation. Leveraging the CNN model to predict game outcomes and optimize move selection, the algorithm improved iteratively, achieving superior performance. The self-play and evaluation process ensured that only models with significant improvements were accepted. This showcases the strength of combining MCTS and deep learning, as AlphaZero successfully adapted to our custom game and demonstrated advanced strategic decision-making. This study is the first step toward fully implementing Turkish Dama in the AlphaZero algorithm.

VI. APPENDIX

6.1 Rules of Pawn Dama

1. The board must have at least four rows. Other than this, they can be of any side.

2. At the start of the game, white pawns occupy the first two rows of the board. Black pawns occupy the last two rows of the board.

3. Pawns can move one square forward if that square is not occupied.

4. If this is their first move, pawns have the option to move two squares forward if both squares are unoccupied.

5. Pawns capture diagonally.

6. The side whose pawn reaches the opposite end of the board first wins.

VII. CODE AVAILABILITY

Our project's complete source code implementation is available as open-source at: https://github.com/erdemphl/solving-pawn-dama-with-alphazero. Our code builds upon and extends the code presented in [18].

REFERENCES

[1] Shannon, C.E. (1950). Programming a Computer for Playing Chess. Philosophical Magazine, 41(314), 256-275.

[2] Knuth, D.E., & Moore, R.W. (1975). An Analysis of Alpha-Beta Pruning. Artificial Intelligence, 6(4), 293-326.

[3] Newborn, M. (1997). Kasparov versus Deep Blue: Computer Chess Comes of Age. Springer.

[4] Schaeffer, J. (1997). One Jump Ahead: Challenging Human Supremacy in Checkers. Springer-Verlag.

[5] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. Communications of the ACM, 38(3), 58-68.

[6] Samuel, A.L. (1959). Some studies in machine learning using the game of checkers. IBM Journal of Research and Development, 3(3), 210-229.

[7] Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In Proceedings of the 5th International Conference on Computers and Games (pp. 72-83).

[8] Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. Machine Learning, 282, 282-293.

[9] Gelly, S., & Silver, D. (2007). Combining Online and Offline Knowledge in UCT. In Proceedings of the 24th International Conference on Machine Learning (273-280).

[10] Ciancarini, P., & Favini, G.P. (2010). Monte Carlo tree search in Kriegspiel. Artificial Intelligence, 174, 670-684.

[11] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. Artificial Intelligence Review, 34(1), 1-49.

[12] Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., & Lanctot, M. (2016). Mastering the game of Go with deep neural networks and tree search. Nature, 529(7587), 484-489.

[13] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., & Hassabis, D. (2017). Mastering the game of Go without human knowledge. Nature, 550(7676), 354-359.

[14] Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. Science, 362(6419), 1140-1144.

[15] Vinyals, O., Babuschkin, I., Czarnecki, W.M., Mathieu, M., Dudzik, A., Chung, J., Choi, D., Powell, R., Ewalds, T., Georgiev, P., & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature, 575(7782), 350-354.

[16] Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., Guez, A., Lockhart, E., Hassabis, D., Graepel, T., & Silver, D. (2020). Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model. Nature, 588(7839), 604-609.

[17] Dong, H., Ding, Z., & Zhang, S. (2020). Fundamentals, Research and Applications. In Deep Reinforcement Learning (pp. 391-414).

[18] Thakoor, S., Nair, S., & Jhunjhunwala, M. (2016). Learning to play othello without human knowledge. Stanford University.

[19] Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. ArXiv. Retrieved from https://arxiv.org/abs/1712.01815

[20] LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. Proceedings of the IEEE, 86(11), 2278-2324.

[21] Ioffe, S., & Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Proceedings of the International Conference on Machine Learning.

[22] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. Journal of Machine Learning Research, 15(1), 1929-1958.

[23] Nair, V., & Hinton, G.E. (2010). Rectified linear units improve restricted boltzmann machines. In Proceedings of the International Conference on Machine Learning.

[24] Tomašev, N., Paquet, U., Hassabis, D., & Kramnik, V. (2020). Assessing game balance with AlphaZero: Exploring alternative rule sets in chess. arXiv preprint arXiv:2009.04374. https://arxiv.org/abs/2009.04374

[25] Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering Atari games with limited data. Advances in Neural Information Processing Systems, 34, 14917–14929.

[26] Schmid, M., Moravčík, M., Burch, N., Kadlec, R., Davidson, J., Waugh, K., Bard, N., Timbers, F., Lanctot, M., Holland, G. Z., Davoodi, E., Christianson, A., & Bowling, M. (2023). Student of Games: A unified learning algorithm for both perfect and imperfect information games. Science Advances, 9(45), eadg3256. https://doi.org/10.1126/sciadv.adg3256

[27] Sutton, R. S., & Barto, A. G. (2018). Reinforcement learning: An introduction (2nd ed.). MIT Press.