University of Camerino

School of Science and Technology

Master Degree in Computer Science

Clustered Federated Deep Reinforcement Learning with Selective Aggregation

A Framework for Chess Playstyle Preservation

Supervisor

Prof. Massimo Callisto De Donato

Co-Supervisor

PhD. Student Martina Zannotti

Student

Francesco Finucci

Academic Year 2024-2025

Abstract

Federated learning enables collaborative model training across distributed nodes while preserving data privacy, but its application to reinforcement learning presents unique challenges, particularly the tension between collaborative learning and behavioral diversity. In chess, different playing styles (tactical vs. positional) represent valuable strategic diversity that traditional federated averaging would homogenize into a single global model.

This thesis presents a novel clustered federated deep reinforcement learning framework that maintains specialized chess engines while enabling knowledge transfer. We implement a three-tier hierarchical aggregation system: (1) local training on distributed nodes, (2) intra-cluster federated averaging within playstyle groups, and (3) selective inter-cluster aggregation that shares only low-level feature extraction layers while preserving cluster-specific policy and value heads. Our approach employs an AlphaZero-style neural network architecture with a 119-plane board representation and dual policy-value heads, combining Monte Carlo Tree Search (MCTS) for move exploration with self-play reinforcement learning. The system is bootstrapped using playstyle-filtered Lichess databases and tactical puzzle datasets before transitioning to self-play training.

The system comprises tactical and positional clusters, each containing four federated nodes that collaboratively learn cluster-specific strategies. By sharing only generic feature extractors (convolutional and early residual blocks) while maintaining separate decision-making layers (policy and value heads), our selective aggregation preserves strategic diversity while accelerating convergence through knowledge transfer. We evaluate trained models against Stockfish across multiple ELO levels and measure cluster diversity through specialized metrics.

Experimental results demonstrate that clustered federated learning successfully balances collaboration and specialization, producing distinct chess engines that maintain playstyle characteristics while benefiting from distributed training. This framework extends federated learning to reinforcement learning domains requiring behavioral diversity, with applications beyond chess to multi-agent systems, personalized AI assistants, and distributed robotics.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The rise of distributed computing and machine learning has created unprecedented opportunities for collaborative AI systems, yet also introduced fundamental challenges in how these systems learn and share knowledge. Federated learning has emerged as a paradigm that enables multiple agents to collaboratively train models while preserving data privacy and locality. However, when applied to reinforcement learning, particularly in domains requiring diverse behavioral strategies, traditional federated approaches face a critical tension: the trade-off between collaborative learning efficiency and the preservation of strategic diversity.

This thesis addresses this challenge in the context of chess, a domain where strategic diversity is not merely desirable but essential. Different playing styles, from aggressive tactical combinations to patient positional maneuvering, represent distinct approaches that have value in different game contexts. While traditional federated averaging would blend these approaches into a homogeneous strategy, we propose a clustered architecture that maintains this diversity while still enabling knowledge transfer across distributed agents.

1.1 Motivation

The success of deep reinforcement learning in complex domains like chess, Go, and Atari games has demonstrated the potential for AI systems to achieve superhuman performance through self-play and iterative improvement. AlphaZero, in particular, revolutionized computer chess by combining deep neural networks with Monte Carlo Tree Search, learning entirely from self-play without human knowledge. However, these achievements typically rely on massive centralized computational resources and homogeneous training data, limiting their applicability in distributed settings where data and computation are naturally partitioned.

Federated learning addresses some of these limitations by enabling collaborative model training across distributed nodes without centralizing data. This approach offers several advantages: preservation of data privacy, reduced communication overhead, and the ability to leverage diverse computational resources. In the context of reinforcement learning for chess, federated approaches could allow multiple training agents to share knowledge while maintaining local control over their training processes and data.

Yet traditional federated learning, designed primarily for supervised learning tasks, faces a fundamental challenge when applied to reinforcement learning in strategic domains. The standard federated averaging algorithm converges toward a single global model, effectively homogenizing the strategies learned by different agents. In chess, this homogenization is problematic. Human chess has evolved numerous distinct playing styles, each with strengths in different positions and game phases. Tactical players excel at calculating concrete variations and exploiting immediate opportunities, while positional players specialize in long-term strategic maneuvering and structural advantages. These diverse approaches are not simply different paths to the same solution; they represent fundamentally different strategic philosophies that have coexisted and enriched the game for centuries.

The tension between collaboration and diversity becomes acute in federated reinforcement learning. While agents benefit from sharing knowledge about general chess principles and pattern recognition, forcing them to converge to identical strategies eliminates the very diversity that makes chess rich and complex. A purely tactical model may miss subtle positional nuances, while a purely positional model may overlook sharp tactical opportunities. An ideal system would preserve these distinct strategic identities while still enabling agents to learn from each other's experiences.

Furthermore, maintaining strategic diversity has practical benefits beyond chess. In multi-agent systems, personalized AI assistants, and distributed robotics, diverse behavioral strategies enable systems to adapt to different contexts, user preferences, and environmental conditions. A framework that can balance collaborative learning with behavioral preservation addresses a broader challenge in distributed artificial intelligence: how to build systems that are both cooperative and specialized.

1.2 Research Questions

This thesis investigates the following research questions:

1. **How can federated learning be adapted to preserve strategic diversity in reinforcement learning domains?** Traditional federated averaging produces a single global model, but many domains benefit from maintaining distinct behavioral strategies. We investigate whether a clustered federated architecture can balance knowledge sharing with playstyle preservation.

2. **What aggregation mechanisms enable knowledge transfer without homogenizing agent behaviors?** We explore selective aggregation strategies that share low-level feature representations while maintaining cluster-specific decision-making layers. The question is whether this approach can accelerate learning while preserving the distinct characteristics of different playing styles.

3. **Can playstyle-specific training data effectively bootstrap distinct strategic identities in a federated setting?** We investigate whether filtering training data by chess opening classifications (ECO codes) and puzzle types can establish and maintain tactical versus positional specializations throughout federated training rounds.

4. **How can we measure and quantify strategic diversity in federated chess engines?** Beyond standard performance metrics like ELO ratings, we need methods to assess whether cluster-specific models maintain distinct strategic characteristics or converge toward homogeneous play.

5. **Does clustered federated learning provide performance benefits compared to isolated training?** We examine whether the proposed three-tier aggregation system (local training, intra-cluster averaging, inter-cluster selective sharing) improves learning efficiency and final playing strength compared to agents training independently.

These questions guide our exploration of clustered federated deep reinforcement learning, with chess serving as a concrete testbed for principles applicable to broader distributed AI systems requiring both collaboration and specialization.

1.3 Contributions

This thesis makes the following contributions:

1. **A novel clustered federated learning architecture for reinforcement learning.** We introduce a three-tier hierarchical aggregation system that maintains multiple cluster-specific models rather than converging to a single global model. This architecture enables collaborative learning while preserving behavioral diversity.

2. **Selective inter-cluster aggregation mechanism.** We design and implement a selective weight-sharing strategy that aggregates only low-level feature extraction layers across clusters while maintaining separate policy and value heads for each playstyle. This mechanism enables knowledge transfer without homogenizing strategic characteristics.

3. **Playstyle-aware data filtering methodology.** We develop a systematic approach for establishing distinct strategic identities using ECO opening code classification and puzzle type filtering, demonstrating how domain-specific data curation can initialize and maintain behavioral diversity in federated settings.

4. **Complete federated AlphaZero implementation for chess.** We provide an end-to-end system integrating AlphaZero-style deep reinforcement learning with federated infrastructure, including cluster management, asynchronous communication, distributed aggregation, and both supervised bootstrapping and self-play training phases.

5. **Evaluation framework for strategic diversity.** We develop metrics to quantify and track the preservation of cluster-specific playing styles throughout federated training, providing tools for assessing whether models maintain distinct characteristics or undergo homogenization.

6. **Empirical analysis of collaboration-diversity tradeoffs.** Through systematic experiments, we provide insights into the benefits and limitations of clustered federated learning compared to isolated training and traditional federated averaging.

1.4 Thesis Structure

The remainder of this thesis is organized as follows:

Chapter 2: Background provides the theoretical foundation for this work. We review deep reinforcement learning and the AlphaZero algorithm, including neural network architectures, Monte Carlo Tree Search, and self-play training. We then introduce federated learning principles, covering federated averaging and its applications. Finally, we discuss related work in distributed reinforcement learning and behavioral diversity preservation.

Chapter 3: Methodology presents our clustered federated learning framework. We describe the three-tier hierarchical aggregation system, detailing local training, intra-cluster federated averaging, and selective inter-cluster aggregation. We explain the selective weight-sharing mechanism that preserves strategic diversity while enabling knowledge transfer. We also present our playstyle-aware data filtering methodology using ECO codes and puzzle classifications.

Chapter 4: Implementation describes the technical realization of our system. We detail the AlphaZero-style neural network architecture with its 119-plane board representation and dual policy-value heads. We explain the server architecture for cluster management and distributed aggregation, the client-side training infrastructure, and the data processing pipeline for Lichess databases and puzzle datasets.

Chapter 5: Experiments outlines our experimental setup and evaluation methodology. We describe the cluster topology with tactical and positional specializations, training configurations, and hyperparameters. We present our evaluation framework, including Stockfish-based performance testing, diversity metrics for measuring playstyle preservation, and baseline comparisons against isolated training and traditional federated averaging.

Chapter 6: Results presents our empirical findings. We analyze training convergence across clusters, evaluate playing strength through ELO estimation, and measure strategic diversity preservation throughout federated rounds. We compare clustered federated learning against baseline approaches and provide insights into the collaboration-diversity tradeoff.

Chapter 7: Conclusion summarizes our contributions and findings. We discuss the implications of our work for federated reinforcement learning and distributed AI systems. We acknowledge limitations of the current approach and propose directions for future research, including extensions to other domains, alternative aggregation strategies, and scalability improvements.

Chapter 2

Background and Related Work

This chapter provides the theoretical foundation for our clustered federated deep reinforcement learning framework. We begin by describing our literature search methodology to ensure transparency and reproducibility. We then provide an overview of reinforcement learning fundamentals and the AlphaZero algorithm that forms the basis of our chess engine. We introduce federated learning principles and discuss how they can be adapted to reinforcement learning settings. Finally, we review related work in distributed reinforcement learning, behavioral diversity preservation, and federated learning applications.

2.1 Literature Review Methodology

To ensure a comprehensive and systematic review of relevant literature, we conducted a multi-stage search process across academic databases, preprint repositories, and technical documentation. This section details our search strategy, inclusion criteria, and the tools used to identify and synthesize relevant work.

2.1.1 Search Strategy

We performed systematic searches across multiple academic databases and repositories between September 2024 and January 2025. The primary sources included:

- **Google Scholar:** Broad coverage of computer science literature and citation tracking
- **arXiv.org:** Recent preprints in machine learning (cs.LG, cs.AI, cs.MA)
- **ACM Digital Library:** Conference proceedings (NeurIPS, ICML, ICLR, AAAI)
- **IEEE Xplore:** Systems and distributed computing literature
- **Semantic Scholar:** AI-powered search and paper recommendations

2.1.2 Search Terms and Queries

Our literature search employed combinations of core terms, connected with Boolean operators. Table 2.1 shows the primary and secondary search queries used across different databases.

We also performed backward citation tracking (reviewing references of key papers) and forward citation tracking (identifying papers that cite foundational work) to ensure coverage of relevant literature.

Table 2.1: Literature Search Queries

Category	**Search Query**
Primary Queries	
Federated RL	"federated learning" AND "reinforcement learning"
Clustered FL	"clustered federated learning" OR "personalized federated learning"
Distributed Chess AI	"AlphaZero" AND ("federated" OR "distributed")
Behavioral Diversity	"behavioral diversity" AND "multi-agent"
Chess Playstyle	"chess AI" AND ("playing style" OR "playstyle")
Selective Aggregation	"selective aggregation" AND "federated learning"
Distributed MCTS	"Monte Carlo Tree Search" AND "distributed"
Secondary Queries	
Heterogeneous FL	"heterogeneous federated learning"
Non-IID Data	"non-IID federated learning"
Transfer Learning	"transfer learning" AND "deep reinforcement learning"
Distributed Self-Play	"self-play" AND ("distributed" OR "federated")
Quality Diversity	"quality diversity algorithms"
Model Divergence	"model divergence" AND "federated"

2.1.3 Inclusion and Exclusion Criteria

Papers were included if they met the following criteria:

Inclusion criteria:

- Published between 2015 and 2025 (with exceptions for seminal earlier work)
- Directly relevant to federated learning, reinforcement learning, or chess AI
- Peer-reviewed or from reputable preprint repositories (arXiv)
- Available in English
- Sufficient technical detail to understand methodology

Exclusion criteria:

- Purely theoretical work without implementation insights
- Domain-specific applications unrelated to game-playing or multi-agent systems
- Duplicate publications or superseded versions
- Insufficient detail on methods or results

2.1.4 AI-Assisted Literature Discovery

In addition to traditional database searches, we leveraged AI tools to assist with literature discovery and synthesis. Table 2.2 shows the AI-assisted queries used for research assistance.

These AI-assisted searches were particularly useful for:

Table 2.2: AI-Assisted Literature Discovery Queries

Tool	**Query / Purpose**
Claude (Anthropic)	
Federated RL Overview	"Summarize recent advances in federated reinforcement learning, focusing on methods that handle heterogeneous agents"
Behavioral Diversity	"What are the key challenges in maintaining behavioral diversity in multi-agent systems?"
Selective Aggregation	"Compare different approaches to selective aggregation in federated learning"
Distributed AlphaZero	"Find papers that combine AlphaZero-style training with distributed or federated approaches"
Semantic Scholar	
Recommendations	AI-powered paper recommendations based on citation graphs and content similarity
Connected Papers	
Citation Networks	Visualizing citation networks and identifying research clusters

1. Quickly understanding the landscape of a new research area
2. Identifying terminology variations (e.g., "behavioral diversity" vs "policy diversity" vs "strategic heterogeneity")
3. Discovering connections between seemingly disparate research communities (e.g., federated learning and chess AI)
4. Generating additional search terms based on paper abstracts

2.1.5 Documentation and Synthesis

We maintained a structured database of reviewed papers using reference management software, tracking:

- Paper metadata (authors, venue, year)
- Key contributions and findings
- Methodological approaches
- Relevance to our research questions
- Gaps or limitations identified

This systematic approach ensured comprehensive coverage of relevant literature while maintaining focus on our core research questions about clustered federated learning for reinforcement learning with behavioral diversity preservation.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm where an agent learns to make decisions by interacting with an environment to maximize cumulative rewards [24]. Unlike supervised learning, where correct answers are provided, RL agents must discover effective strategies through trial and error, receiving only sparse feedback about the quality of their actions.

2.2.1 Markov Decision Processes

Reinforcement learning problems are typically formalized as Markov Decision Processes (MDPs) [24]. An MDP is defined by a tuple (S, A, P, R, γ) where:

- S is the set of possible states the environment can be in
- A is the set of actions the agent can take
- $P(s'|s, a)$ is the transition probability of reaching state s' after taking action a in state s
- $R(s, a, s')$ is the reward received when transitioning from state s to s' via action a
- $\gamma \in [0, 1]$ is the discount factor that determines how much future rewards are valued relative to immediate rewards

The agent's behavior is determined by a policy $\pi(a|s)$ that specifies the probability of taking action a in state s. The goal of reinforcement learning is to find an optimal policy π^* that maximizes the expected cumulative discounted reward, known as the return:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

The value function $V^\pi(s)$ represents the expected return when starting in state s and following policy π:

$$V^\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.2}$$

Similarly, the action-value function $Q^\pi(s, a)$ represents the expected return when taking action a in state s and then following policy π:

$$Q^\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{2.3}$$

2.2.2 Deep Reinforcement Learning

Traditional RL algorithms use tabular representations to store value functions, which becomes impractical for large state spaces. Deep reinforcement learning addresses this limitation by using neural networks as function approximators to estimate value functions and policies. This enables RL to scale to complex domains like video games, robotics, and board games.

Deep Q-Networks (DQN) [15] pioneered this approach by using convolutional neural networks to approximate the action-value function $Q(s, a)$ for Atari games. The key innovations included experience replay, where transitions are stored in a buffer and sampled randomly for training, and a separate target network that stabilizes learning.

Policy gradient methods [26] provide an alternative approach by directly parameterizing the policy $\pi_\theta(a|s)$ with neural network parameters θ. The policy gradient theorem allows us to compute gradients of the expected return with respect to these parameters:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^{\pi_\theta}(s, a)] \tag{2.4}$$

Actor-critic methods combine these approaches by maintaining both a policy network (actor) and a value network (critic). The critic evaluates the quality of the actor's actions, providing lower-variance gradient estimates.

2.2.3 AlphaZero and Monte Carlo Tree Search

AlphaZero [21] represents a breakthrough in deep reinforcement learning for board games, achieving superhuman performance in chess, Go, and shogi through pure self-play learning without human knowledge. The algorithm combines three key components: a deep neural network for position evaluation, Monte Carlo Tree Search for move planning, and reinforcement learning for continuous improvement.

Neural Network Architecture

The AlphaZero neural network takes the current board position as input and produces two outputs:

- A **policy head** $p = f_\theta^p(s)$ that outputs a probability distribution over legal moves
- A **value head** $v = f_\theta^v(s)$ that outputs a scalar value estimating the probability of winning from the current position

The network uses a deep residual architecture with convolutional layers to process spatial patterns on the board. This dual-headed design allows the network to both suggest promising moves and evaluate position quality, which are used together during search.

Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [3] is a best-first search algorithm that builds a search tree incrementally through random sampling. Unlike traditional minimax search used in classical chess engines, MCTS focuses computational effort on the most promising variations.

Each node in the search tree represents a board position and stores statistics about visits and values. The search proceeds through four phases:

1. **Selection:** Starting from the root, choose child nodes that balance exploration (trying less-visited moves) and exploitation (following moves with high estimated value) using the PUCT formula:

$$UCT(s,a) = Q(s,a) + c_{puct} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1 + N(s,a)} \tag{2.5}$$

   where $Q(s,a)$ is the mean action value, $P(s,a)$ is the prior probability from the neural network, and $N(s,a)$ is the visit count.

2. **Expansion:** When a leaf node is reached, evaluate the position using the neural network to get policy priors and value estimate.

3. **Simulation:** In AlphaZero, this phase is replaced by direct neural network evaluation rather than random rollouts.

4. **Backpropagation:** Update statistics along the path from leaf to root, incrementing visit counts and updating action values.

After running many MCTS simulations (typically 800 for AlphaZero), the final move is selected based on visit counts, which represent a refined estimate of move quality informed by deep search.

Self-Play Training

AlphaZero improves through iterative self-play. The current neural network generates training games by playing against itself using MCTS-guided move selection. Each position in these games provides training data:

- The **policy target** is the distribution of MCTS visit counts π, which represents an improved policy compared to the raw network output
- The **value target** is the final game outcome $z \in \{-1, 0, 1\}$ (loss, draw, win)

The network is trained to minimize a combined loss function:

$$L = (z - v)^2 - \pi^T \log p + c||\theta||^2 \tag{2.6}$$

This loss function encourages the network to predict game outcomes accurately (value loss) and match the improved MCTS policy (policy loss), with L2 regularization to prevent overfitting.

The key insight of AlphaZero is that MCTS can be viewed as a policy improvement operator [22]. By repeatedly training the network on self-play games where moves are selected by MCTS, the network gradually improves, which in turn makes future MCTS searches more effective. This creates a positive feedback loop that leads to continuous improvement without requiring any domain knowledge beyond the game rules.

2.3 Federated Learning

Federated learning is a distributed machine learning paradigm that enables multiple participants to collaboratively train a shared model while keeping their data decentralized [13]. Unlike traditional centralized training where all data is aggregated in one location, federated learning brings the model to the data rather than the data to the model. This approach addresses privacy concerns, reduces communication costs, and enables learning from data that cannot be easily centralized due to legal, technical, or practical constraints.

2.3.1 Federated Averaging Algorithm

The foundational algorithm for federated learning is Federated Averaging (FedAvg), proposed by McMahan et al. [13]. FedAvg coordinates distributed training across multiple clients through a central server that aggregates local model updates.

The basic FedAvg procedure consists of several rounds of communication between the server and clients:

1. **Model Distribution:** The server sends the current global model parameters w_t to a subset of clients
2. **Local Training:** Each selected client k trains the model on their local dataset D_k for E epochs, producing updated parameters w_t^k
3. **Aggregation:** The server collects the updated models and computes a weighted average:

   $$w_{t+1} = \sum_{k=1}^{K} \frac{n_k}{n} w_t^k \tag{2.7}$$

   where $n_k = |D_k|$ is the size of client k's dataset and $n = \sum_k n_k$ is the total data size
4. **Iteration:** This process repeats for multiple rounds until convergence

The key advantage of FedAvg is that it performs multiple local optimization steps before communication, significantly reducing the number of communication rounds needed compared to sending gradients after each batch. This is crucial since communication is often the bottleneck in distributed settings.

2.3.2 Challenges in Federated Learning

While federated learning offers many benefits, it also introduces several challenges compared to centralized training [8]:

Non-IID Data: In typical federated settings, data across clients is not independently and identically distributed [12]. Different clients may have data from different distributions, different label distributions, or different amounts of data. This heterogeneity can slow convergence and lead to suboptimal models.

Communication Efficiency: Since clients may have limited bandwidth or intermittent connectivity, minimizing communication rounds is essential. Techniques like gradient compression, quantization, and local optimization help reduce communication costs.

System Heterogeneity: Clients may have varying computational capabilities, storage capacity, and availability. Some clients may be able to train quickly on powerful hardware while others are limited by mobile device constraints.

Privacy and Security: While federated learning keeps raw data decentralized, model updates can still leak information about training data. Differential privacy and secure aggregation techniques can provide stronger privacy guarantees but add computational overhead.

2.3.3 Personalization in Federated Learning

A key limitation of vanilla FedAvg is its assumption that all clients should converge to a single global model. However, in many real-world scenarios, different clients or groups of clients may benefit from specialized models tailored to their specific data distributions or preferences.

Personalized federated learning addresses this by allowing some degree of model customization per client or client group while still leveraging collaborative learning. Several approaches have been proposed:

Fine-tuning: Clients start with a global model but continue training locally after federation completes, adapting the model to their specific data.

Multi-task Learning: The model is split into shared and personalized layers [1]. Shared layers learn common representations across all clients, while personalized layers adapt to individual client characteristics.

Clustered Federated Learning: Clients are grouped into clusters based on data similarity or other criteria [20]. Each cluster maintains its own model through federated averaging within the cluster, while potentially sharing knowledge across clusters.

Meta-Learning: The global model is trained to be easily adaptable to new clients with minimal fine-tuning [6], using techniques like Model-Agnostic Meta-Learning (MAML).

These personalization techniques recognize that a one-size-fits-all global model is not always optimal, especially when client data distributions are significantly different. Our work builds on the clustered federated learning approach, extending it to reinforcement learning settings where behavioral diversity is not just a result of data heterogeneity but a desired outcome.

2.4 Chess Engines and AI

Computer chess has been a central domain for artificial intelligence research since the field's inception. The evolution of chess engines reflects broader trends in AI, from symbolic rule-based systems to search algorithms to modern deep learning approaches.

2.4.1 Classical Chess Engines

Traditional chess engines rely on three core components: board representation, move generation, and position evaluation combined with tree search.

Minimax and Alpha-Beta Pruning: Classical engines use minimax search to explore the game tree, assuming both players play optimally. The algorithm recursively evaluates positions by assuming the maximizing player wants the highest score while the minimizing player wants the lowest. Alpha-beta pruning [10] dramatically reduces the search space by eliminating branches that cannot affect the final decision.

Hand-Crafted Evaluation Functions: Classical engines evaluate positions using carefully designed functions that consider material balance, piece activity, pawn structure, king safety, and other strategic factors. These evaluation functions encode centuries of human chess knowledge into numerical scores.

Stockfish [23], currently the strongest classical chess engine, represents the pinnacle of this approach. It combines sophisticated search algorithms, aggressive pruning techniques, and finely tuned evaluation heuristics to search billions of positions per second. Despite being based on traditional methods, Stockfish remains competitive with neural network engines in many positions. Earlier milestones in classical chess AI include Deep Blue [4], which famously defeated world champion Garry Kasparov in 1997.

2.4.2 Neural Network Chess Engines

The introduction of AlphaZero [21] in 2017 demonstrated that neural networks trained through self-play could achieve superhuman chess performance without domain knowledge. Unlike classical engines that use hand-crafted evaluation functions, AlphaZero learned position evaluation and move selection entirely from self-play.

The success of AlphaZero inspired several open-source projects, most notably Leela Chess Zero (LC0) [11], which reimplemented the AlphaZero approach using distributed training across thousands of volunteer computers. LC0 has evolved to match and sometimes exceed Stockfish's playing strength, particularly in positions requiring long-term strategic planning.

Neural network engines exhibit different playing characteristics compared to classical engines. They tend to favor positional understanding and long-term planning over tactical calculation depth. This has enriched computer chess by introducing more varied and sometimes more human-like playing styles.

2.4.3 Playing Style in Chess

Chess players, both human and computer, exhibit distinct playing styles that reflect different strategic philosophies. Two broad categories often used to characterize playing style are:

Tactical Play: Emphasizes concrete calculation, immediate threats, and combinative play. Tactical players excel at spotting forcing sequences, sacrifices, and sharp variations. They prefer dynamic positions with many pieces on the board where calculation depth determines the outcome.

Positional Play: Focuses on long-term strategic advantages like pawn structure, piece coordination, and space control. Positional players excel at gradual maneuvering, prophylaxis, and converting small advantages into wins. They prefer positions where understanding trumps calculation.

In human chess, players typically develop preferences for certain opening systems and strategic themes that align with their style. Mikhail Tal exemplified tactical brilliance with his sacrificial attacks, while Anatoly Karpov demonstrated the power of refined positional technique. Most strong players can play both styles but show preferences and strengths in certain types of positions.

For chess engines, playing style has traditionally been less pronounced. Classical engines tend toward tactical play due to their search depth, while neural network engines often display more positional understanding. Our work explores whether distinct playing styles can be deliberately cultivated and maintained in federated learning settings, creating specialized engines rather than homogeneous ones.

2.5 Related Work

Our work draws on several areas of research: distributed reinforcement learning, federated learning applications to RL, behavioral diversity in multi-agent systems, and clustered federated learning.

2.5.1 Distributed Reinforcement Learning

Distributed training has become essential for reinforcement learning in complex domains due to the computational demands of both environment interaction and neural network training.

Parallel Experience Collection: Many RL systems use multiple actors to collect experience in parallel, dramatically increasing sample efficiency. A3C (Asynchronous Advantage Actor-Critic) [14] introduced asynchronous updates from multiple workers to a shared model. IMPALA (Importance Weighted Actor-Learner Architecture) [5] separates experience collection from learning, using importance sampling to handle the resulting off-policy data.

Distributed AlphaZero: The original AlphaZero training [21] used distributed self-play, with many workers generating games in parallel while a central learner updates the neural network. This architecture enables the massive scale of training required for superhuman performance, AlphaZero played nearly 5 million games during training.

However, these distributed RL approaches still rely on centralized aggregation and aim for a single global model. They distribute computation for efficiency but do not address the challenge of maintaining behavioral diversity or training multiple specialized models collaboratively.

2.5.2 Federated Reinforcement Learning

Applying federated learning to reinforcement learning is an emerging research area [9, 16]. While traditional supervised federated learning deals with fixed datasets, federated RL must handle the added complexity of exploration, temporal dependencies, and non-stationary data distributions as policies improve.

Policy-Based FedRL: Some approaches extend FedAvg directly to policy gradient methods, aggregating policy network parameters across agents. However, this faces challenges when agents experience different environments or have different reward functions, as policies optimized for different MDPs may not meaningfully average.

Value-Based FedRL: Other work focuses on sharing value function estimates or Q-functions across agents. This can be effective when agents share the same environment but experience different parts of the state space.

Exploration vs. Exploitation Trade-offs: Federated RL introduces unique challenges for exploration. If all agents follow similar exploration strategies, they may collectively fail to explore the state space adequately. Some work addresses this through coordinated exploration strategies or by encouraging diversity in local training.

Most federated RL research has focused on settings where agents face different but related tasks, aiming to share knowledge across task distributions. Our work differs by considering agents working on the same task (chess) but seeking to maintain distinct behavioral strategies rather than converging to a single solution.

2.5.3 Behavioral Diversity in Multi-Agent Systems

Maintaining diversity in multi-agent systems has been studied in several contexts, motivated by applications in team behavior, robust learning, and ensemble methods.

Quality Diversity Algorithms: MAP-Elites and related algorithms [18] explicitly optimize for both performance and behavioral diversity. They maintain archives of solutions that exhibit different behaviors, even if some are suboptimal, creating a diverse collection of strategies.

Diversity-Driven Exploration: In multi-agent RL, some work [17] uses diversity objectives to encourage agents to explore different parts of the state space or learn different policies. This can improve collective exploration efficiency and robustness.

Emergent Communication and Specialization: Research in multi-agent communication has shown that agents can spontaneously develop specialized roles when working toward common goals, with different agents handling different subtasks.

Our work differs from these approaches in that we seek to maintain diversity not just during training but in the final models, and we do so in a federated setting where agents cannot directly observe each other but must coordinate through aggregation.

2.5.4 Clustered Federated Learning

Clustered federated learning [7, 20] recognizes that in heterogeneous settings, forcing all clients to converge to a single global model may be suboptimal. Instead, clients are grouped into clusters, with each cluster maintaining its own model.

Automatic Clustering: Several methods [7] propose to automatically discover clusters during training. Clients are initially assigned to clusters randomly or based on data characteristics, then cluster membership is refined based on model similarity or gradient alignment. This allows the system to discover natural groupings in the data distribution.

Multi-Center Federated Learning: Some approaches maintain multiple global models and allow clients to contribute to the model that best matches their data. This creates a form of competitive federated learning where models specialize to different data distributions.

Hierarchical Aggregation: Similar to our approach, some work [2] uses hierarchical aggregation where updates are first aggregated within clusters, then partial aggregation occurs across clusters. This balances the benefits of local specialization with global knowledge sharing.

However, existing clustered federated learning work focuses on supervised learning tasks with heterogeneous data distributions. The clustering emerges from data heterogeneity rather than being designed to preserve behavioral characteristics. Our work extends these ideas to reinforcement learning where we explicitly initialize and maintain clusters based on strategic playing style, and we introduce selective layer-wise aggregation to balance knowledge transfer with behavioral preservation.

2.5.5 Transfer Learning in Deep RL

Transfer learning in reinforcement learning [25] aims to leverage knowledge from one task to accelerate learning on related tasks. This is relevant to our selective aggregation mechanism.

Progressive Neural Networks: Freeze previously learned networks and add new capacity for new tasks [19], allowing lateral connections. This prevents catastrophic forgetting but increases model size.

Fine-Tuning and Layer Freezing: Standard practice is to fine-tune pretrained networks on new tasks, often freezing early layers that learn general features while adapting later layers to task specifics.

Multi-Task Learning: Training a single network on multiple related tasks can improve performance on all tasks by learning shared representations. However, this typically assumes tasks are sufficiently similar that a shared representation helps rather than hinders.

Our selective inter-cluster aggregation draws on these insights. We share early feature extraction layers across playing style clusters, analogous to sharing general features in transfer learning, while keeping decision-making layers cluster-specific to preserve strategic differences.

2.5.6 Gaps in Existing Work

While the related work provides valuable insights and techniques, several gaps remain that our research addresses:

1. **Federated RL with Intentional Diversity:** Existing federated RL work focuses on handling unavoidable data heterogeneity, not deliberately maintaining behavioral diversity as a goal.

2. **Selective Layer Aggregation:** While some clustered FL work uses hierarchical aggregation, selective layer-wise aggregation based on functional role (feature extraction vs. decision making) is underexplored.

3. **Playing Style Preservation:** Chess AI research has not addressed how to maintain distinct playing styles in collaborative training settings where the default would be homogenization.

4. **Comprehensive Framework:** No existing work provides an end-to-end system combining clustered federated learning, selective aggregation, and self-play RL for behavioral diversity preservation.

Our work addresses these gaps by developing a framework specifically designed to balance collaborative learning with behavioral preservation in reinforcement learning domains, using chess as a concrete and measurable testbed.

Chapter 3

Methodology

This chapter presents our clustered federated deep reinforcement learning framework for chess with selective layer aggregation. We begin by formally defining the problem as an extension of the standard reinforcement learning formulation, incorporating the unique challenges of maintaining strategic diversity in a federated setting. We then describe the three-tier hierarchical aggregation system that enables knowledge transfer across clusters while preserving playstyle-specific characteristics. The chapter proceeds to detail the neural network architecture, the selective aggregation mechanism, the playstyle-aware data filtering strategy, and the complete training pipeline from supervised bootstrapping through self-play reinforcement learning. Finally, we present the evaluation methodology that will be used to validate our approach in Chapter 5.

3.1 Problem Formulation

This section establishes the formal mathematical framework for our clustered federated reinforcement learning approach to chess. We begin by formulating chess as a Markov Decision Process, then extend this formulation to incorporate the dual objectives of performance optimization and strategic diversity preservation within a federated learning setting.

3.1.1 Markov Decision Process Formulation

We formulate chess as a Markov Decision Process (MDP) [24], defined by the tuple $\mathcal{M} = (S, A, P, R, \gamma)$. The state space S represents all possible board configurations, including piece positions, castling rights, en passant opportunities, and move history. Each state $s \in S$ encodes a complete chess position along with the relevant game state information required to determine legal moves. The action space $A(s)$ contains all legal moves available from state s, varying in size depending on the position but typically containing between 20 and 80 possible moves in non-trivial positions.

The transition function $P(s'|s, a)$ is deterministic in chess, as each legal move a from state s results in a unique next state s'. The reward function $R(s, a, s')$ provides feedback about the quality of moves and game outcomes. In our implementation, rewards are primarily assigned at terminal states, with $R = +1$ for winning positions, $R = -1$ for losing positions, and $R = 0$ for draws. The discount factor $\gamma \in [0, 1]$ determines the relative importance of immediate versus future rewards, though in chess with deterministic transitions to terminal states, this factor plays a less critical role than in many other RL domains.

The objective in the single-agent setting is to learn a policy $\pi : S \to A$ that maximizes the expected cumulative reward. In chess, this corresponds to finding a policy that maximizes win probability while minimizing loss probability across all possible game trajectories. The policy is typically parameterized by a neural network with parameters θ, yielding $\pi_\theta(a|s)$, which outputs a probability distribution over legal actions given the current state.

3.1.2 Strategic Diversity Objective

Traditional federated learning [13] aims to train a single global model by aggregating knowledge from distributed clients. However, our framework introduces a fundamentally different objective: we seek to train multiple distinct models, each specialized for a different strategic approach to chess, while still enabling beneficial knowledge transfer between them. This dual objective creates a tension between convergence and divergence that standard federated learning algorithms are not designed to handle.

Formally, we partition our set of N clients into K clusters $C_1, C_2, \ldots, C_K$, where each cluster C_k is associated with a distinct playstyle characteristic. In our primary experiments, we focus on $K = 2$ clusters representing tactical and positional playing styles. Each cluster maintains its own cluster-specific model with parameters θ_k, rather than converging to a single shared model θ_{global} as in standard federated learning.

The strategic diversity objective can be expressed as a bi-objective optimization problem. First, we seek to maximize the performance of each cluster-specific model on its designated playstyle:

$$\max_{\theta_k} \mathbb{E}_{s,a\sim\pi_{\theta_k}}[R(s,a)] \quad \forall k \in \{1, \ldots, K\} \tag{3.1}$$

Second, we aim to maintain measurable divergence between cluster models, ensuring that strategic specialization is preserved:

$$\text{Divergence}(\theta_i, \theta_j) \geq \delta_{\min} \quad \forall i \neq j \tag{3.2}$$

where $\delta_{\min}$ represents a minimum threshold for model differentiation, and the divergence metric can be quantified through various measures such as L2 distance between model parameters, distributional differences in move selection, or behavioral metrics like move type distributions.

This formulation differs fundamentally from standard federated learning, where the objective is typically $\min_\theta \sum_{i=1}^{N} \mathcal{L}_i(\theta)$, seeking a single θ that minimizes the aggregate loss across all clients. Instead, we seek a set of parameters $\{\theta_1, \ldots, \theta_K\}$ that balance individual cluster performance with cross-cluster knowledge transfer while maintaining strategic differentiation.

3.1.3 Federated Learning Constraints

Our problem formulation must satisfy several constraints inherent to federated learning systems. The privacy preservation constraint requires that raw training data remains local to each client. Clients train on their own datasets D_i and only share model parameters or gradients with the central server, never exposing individual game records or training examples. This constraint is particularly relevant in scenarios where training data might contain proprietary opening preparation or game analysis.

The communication efficiency constraint limits the frequency and size of model updates transmitted between clients and the server. Let T denote the total number of training rounds and B the bandwidth available per round. Each communication round t involves clients uploading local model parameters and downloading aggregated cluster models, with total communication cost proportional to the model size and number of participating clients. We denote the communication cost per round as $C_{\text{comm}}(t)$ and require $\sum_{t=1}^{T} C_{\text{comm}}(t) \leq B \cdot T$.

The heterogeneity constraint acknowledges that clients may have different computational capabilities, data distributions, and training objectives. Unlike traditional federated learning settings [12] that treat heterogeneity as an obstacle to convergence, our framework explicitly leverages this heterogeneity to maintain strategic diversity. Clients within cluster C_k train on data that emphasizes playstyle k, creating non-IID data distributions across clusters by design.

Finally, the asynchronous participation constraint allows clients to join and leave training rounds dynamically. Not all clients participate in every round, and we denote the set of participating clients in round t as $S_t \subseteq \{1, \ldots, N\}$. The aggregation mechanism must be robust to varying participation rates while ensuring that each cluster maintains sufficient representation in each training round.

3.2 Clustered Federated Learning Framework

This section describes the overall architecture of our clustered federated learning system for chess. We present the high-level framework design, explain the rationale for our cluster organization, detail the client-server infrastructure, and specify the communication protocols that enable distributed training.

3.2.1 Framework Overview

Our clustered federated learning framework extends traditional federated learning [13] by introducing multiple cluster-specific models rather than a single global model. The system consists of a central aggregation server and a distributed set of training clients organized into playstyle-based clusters. Unlike standard federated averaging [13], where all clients contribute to a unified global model, our framework maintains separate models for each cluster while enabling selective knowledge transfer between them.

The framework operates through three hierarchical levels of aggregation. At the lowest level, individual clients perform local training through self-play reinforcement learning, generating training experiences and updating their local model parameters. At the intermediate level, clients within the same cluster periodically synchronize their models through standard federated averaging, creating a cluster-specific model that captures the shared knowledge of that playstyle group. At the highest level, selective inter-cluster aggregation shares specific layers across clusters while keeping others cluster-specific, enabling knowledge transfer of fundamental chess understanding without homogenizing strategic preferences.

Figure 3.1 illustrates the overall system architecture. The central server coordinates training across multiple distributed clients, which are organized into tactical and positional clusters. Each cluster maintains its own model, and the selective aggregation mechanism enables controlled knowledge sharing between clusters while preserving their distinct characteristics.

This hierarchical design addresses the core challenge of balancing collaboration and specialization. By aggregating within clusters, we enable efficient knowledge sharing among clients with similar objectives. By selectively sharing across clusters, we transfer generalizable representations while maintaining cluster-specific strategic characteristics. The result is a system that leverages the full training data across all clients while producing multiple distinct models optimized for different playing styles.

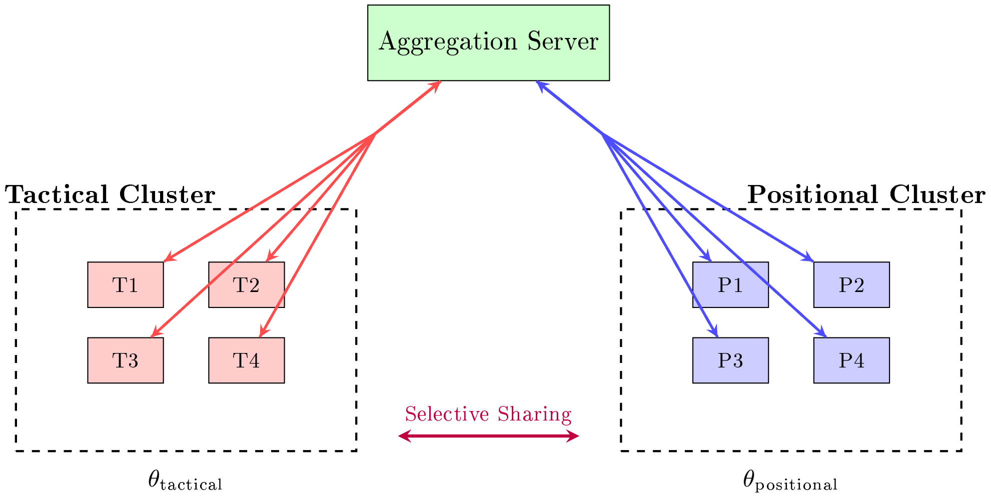


Figure 3.1: System architecture showing the central aggregation server coordinating two clusters of distributed clients. Tactical cluster clients (red) and positional cluster clients (blue) upload model updates to the server and download cluster-specific models. Selective inter-cluster aggregation enables knowledge transfer between clusters.

3.2.2 Cluster Design

We organize clients into two primary clusters based on chess playstyle: tactical and positional. This binary clustering reflects a fundamental dichotomy in chess strategy that has persisted throughout the game's history. Tactical players prioritize concrete calculation, forcing moves, and immediate threats. Positional players emphasize long-term strategic planning, structural advantages, and prophylactic thinking. While these represent endpoints on a spectrum, they provide a clear organizational principle for cluster assignment and a testable hypothesis about strategic diversity preservation.

The tactical cluster trains primarily on sharp, forcing positions with concrete tactical themes. Training data for this cluster is filtered to emphasize openings with early confrontation, games featuring high capture rates, and tactical puzzles requiring precise calculation. The positional cluster trains on strategic positions with long-term planning requirements. Filtered data includes solid openings with emphasis on structure, games with lower exchange rates, and positional puzzles focusing on prophylaxis and planning.

Cluster assignment follows a semi-supervised approach. Initial assignment uses playstyle scores computed from each client's early training games, placing clients into clusters based on their observed tactical tendencies. However, cluster membership is not fixed. Every 20 training rounds, we recompute playstyle scores and allow clients to migrate between clusters if their playing style has shifted significantly. This dynamic reassignment handles the evolution of client behavior during training while maintaining sufficient stability for meaningful cluster-specific learning.

The decision to use two clusters rather than a larger number balances several considerations. Two clusters provide clear interpretability and testability for our core hypotheses about diversity preservation. The tactical-positional dichotomy has well-established foundations in chess theory, making results easier to validate and interpret. Computational overhead scales with the number of clusters, and two clusters allow us to thoroughly evaluate the selective aggregation mechanism without excessive resource requirements. Future work could extend the framework to finer-grained clustering schemes, such as organizing clients by specific opening repertoires or endgame specializations.

3.2.3 Client-Server Architecture

The distributed system architecture follows a star topology with a central aggregation server coordinating multiple training clients. The server maintains authoritative copies of cluster-specific models, orchestrates training rounds, aggregates client updates, and distributes updated models. Clients perform local training through self-play, compute model updates, and communicate with the server to exchange parameters.

Each training round proceeds through a well-defined protocol. The server first selects a subset of participating clients for the current round, ensuring balanced representation from each cluster. Selected clients download the current cluster-specific model corresponding to their assigned cluster. Clients then perform local training for a fixed number of epochs, generating self-play games, collecting training experiences, and updating model parameters through stochastic gradient descent. After completing local training, clients upload their updated model parameters to the server.

The server collects updates from all participating clients and performs aggregation at two levels. First, intra-cluster aggregation computes the weighted average of model parameters within each cluster, where weights typically correspond to the number of training examples processed by each client. This produces updated cluster-specific models that incorporate the collective knowledge of all participating clients in each cluster. Second, selective inter-cluster aggregation shares specified layers across clusters through federated averaging while leaving other layers cluster-specific. The server then stores the updated models and metrics, and the process repeats for the next round.

This architecture provides several advantages over fully peer-to-peer alternatives. Centralized aggregation simplifies coordination and ensures consistent model versions across clients. The server can implement sophisticated aggregation strategies that would be difficult to coordinate in a decentralized setting. Fault tolerance is enhanced, as individual client failures do not disrupt the overall training process. The architecture also facilitates monitoring and evaluation, with the server maintaining comprehensive logs of training metrics and model checkpoints.

3.2.4 Communication Protocol

Communication between clients and the server follows an asynchronous protocol that balances training efficiency with network constraints. The protocol is designed to minimize communication overhead while ensuring that aggregation occurs frequently enough to enable effective knowledge transfer.

Model transmission uses parameter differencing to reduce bandwidth requirements. Rather than transmitting full model parameters each round, clients compute and transmit only the difference between their locally updated model and the initial model they downloaded. For a parameter vector θ_{new} after local training and initial parameters θ_{old}, the client transmits $\Delta\theta = \theta_{\text{new}} - \theta_{\text{old}}$. The server reconstructs updated parameters as $\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta$. This significantly reduces transmission size, particularly in early training rounds when parameter changes are small.

The protocol handles network unreliability through timeout mechanisms and retry logic. Clients set a maximum time limit for upload and download operations. If a client fails to receive an acknowledgment within the timeout period, it retries the transmission up to a maximum number of attempts. If all attempts fail, the client skips the current round and attempts to rejoin in the next round. The server similarly implements timeouts when waiting for client uploads, proceeding with aggregation using only the subset of clients that successfully transmitted their updates.

Synchronization between training rounds follows a flexible schedule that accommodates varying client availability. The server does not require all clients to participate in every round. Instead, it waits for a minimum threshold of clients from each cluster before proceeding with aggregation. This minimum threshold ensures that cluster-specific models benefit from sufficient data diversity while allowing training to proceed even when some clients are offline. If the threshold is not met within a specified time window, the server proceeds with aggregation using available clients, though this situation is logged for monitoring purposes.

Security and privacy considerations are addressed through parameter-level aggregation rather than data sharing. Clients never transmit raw training games or board positions to the server. All communication consists solely of model parameters or parameter differences, which do not directly reveal individual training examples. While sophisticated attacks could potentially extract some information from parameter updates, the aggregation of multiple client updates provides a degree of privacy protection similar to standard federated learning systems.

3.3 Neural Network Architecture

This section describes the deep neural network architecture used by all agents in our federated learning framework. We employ an AlphaZero-style convolutional residual network [21] that takes board positions as input and produces dual outputs: a policy distribution over legal moves and a scalar value estimation. The architecture is designed to support selective layer aggregation, with distinct functional groups that can be shared across clusters or maintained cluster-specifically (see Figure 3.2). We detail the input representation, the residual network structure, the dual output heads, and the layer grouping scheme that enables our selective aggregation strategy.

3.3.1 Input Representation

The neural network receives chess positions as a structured tensor representation encoding the board state, game rules, and move history. Rather than using a simple 8×8 grid with piece identifiers, we employ a rich multi-plane encoding that provides the network with comprehensive positional information while maintaining spatial structure.

The input tensor has shape 8×8×119, representing 119 feature planes over the standard chessboard grid (Figure 3.2). The first 12 planes encode the current piece positions using one-hot encoding, with separate planes for each piece type and color: white pawns, white knights, white bishops, white rooks, white queens, white king, and the corresponding black pieces. Each plane is a binary matrix where a 1 indicates the presence of that piece type at the corresponding square.

To provide temporal context, we include piece positions from the previous seven board states, using an additional 84 planes (12 planes per historical position × 7 time steps). This history enables the network to recognize repetitions, understand pawn structure changes, and track piece mobility patterns across recent moves. The historical encoding is essential for positions where the optimal move depends on the sequence of preceding moves rather than the current position alone.

The remaining 23 planes encode game state information that cannot be inferred from piece positions alone. One plane indicates whose turn it is to move, with all squares set to 1 for white to move and 0 for black to move. Four planes encode castling rights, with separate planes for white kingside, white queenside, black kingside, and black queenside castling availability. One plane marks en passant target squares when applicable. The final 17 planes encode the halfmove clock using a thermometer encoding, representing the number of moves since the last pawn advance or capture, which is critical for the fifty-move rule.

This 119-plane representation provides the network with complete information about the position while preserving the 2D spatial structure of the board. Convolutional layers can exploit translational patterns across the board, learning features that apply regardless of whether a tactical pattern appears on the kingside or queensside. The representation is compatible with the standard AlphaZero approach while containing all information necessary to determine legal moves and evaluate positions according to chess rules.

3.3.2 Residual Network Structure

Following the input representation, the network applies an initial convolution block to transform the 119 input planes into a higher-dimensional feature space. This input convolution consists of a 3×3 convolutional layer with 256 output channels, followed by batch normalization and a ReLU activation function. The use of 256 channels provides sufficient representational capacity for the network to learn rich feature representations while remaining computationally tractable for distributed training across multiple clients.

The core of the architecture consists of 19 residual blocks, each implementing the standard residual connection pattern introduced by He et al. Each residual block contains two 3×3 convolutional layers with 256 channels, with batch normalization and ReLU activation applied after each convolution. The residual connection adds the block's input directly to its output before the final activation, enabling gradient flow through the deep network and facilitating the learning of incremental refinements to the feature representation.

The choice of 19 residual blocks balances network depth with training efficiency. Deeper networks can learn more sophisticated positional patterns but require more computational resources and training data. Our 19-block configuration matches the architecture scale used in moderate-strength AlphaZero implementations [21] and proves sufficient for learning chess at an advanced amateur level. Each block operates on $8 \times 8 \times 256$ feature maps, progressively refining the internal representation through the depth of the network.

The residual blocks are grouped functionally into three categories based on their position in the network: early blocks (blocks 1-6), middle blocks (blocks 7-13), and late blocks (blocks 14-19), as shown in Figure 3.2. This grouping reflects the hierarchical nature of feature learning in deep networks. Early blocks tend to learn low-level spatial patterns and piece configurations. Middle blocks learn tactical motifs and multi-piece coordination. Late blocks integrate high-level strategic concepts and complex positional evaluations. This functional division becomes important for our selective aggregation strategy, as different layer groups may benefit differently from cross-cluster sharing.

3.3.3 Policy and Value Heads

After the 19 residual blocks, the feature representation branches into two separate output heads: a policy head that predicts move probabilities and a value head that estimates position evaluation (Figure 3.2). This dual-head architecture enables the network to learn both which moves to consider and how to evaluate the resulting positions, supporting the Monte Carlo Tree Search algorithm used during move selection.

The policy head transforms the $8 \times 8 \times 256$ feature maps into a probability distribution over possible moves. It consists of a 1×1 convolution that reduces the channel dimension from 256 to 2, followed by batch normalization and ReLU activation. The resulting $8 \times 8 \times 2$ tensor is flattened to a vector of length 128, which is then passed through a fully connected layer with 4672 output units. This output dimension corresponds to the maximum number of possible chess moves in the standard representation: 64 source squares × 73 possible destination patterns (including underpromotions and all move types). A softmax activation produces the final policy distribution π, with illegal moves masked to zero probability based on the current position.

The value head estimates the expected outcome from the current position. It applies a 1×1 convolution to reduce the channel dimension from 256 to 1, followed by batch normalization and ReLU activation. The resulting $8 \times 8 \times 1$ tensor is flattened to a 64-dimensional vector and passed through a fully connected layer with 256 hidden units and ReLU activation. A final fully connected layer with a single output unit, followed by a tanh activation, produces the value estimation $v \in [-1, 1]$, where -1 represents a certain loss, $+1$ represents a certain win, and 0 represents an even position.

Both heads are trained simultaneously using a combined loss function. The policy head is trained with cross-entropy loss against the improved policy distribution produced by MCTS during self-play, encouraging the network to predict the same moves that tree search identifies as strong. The value head is trained with mean squared error against the actual game outcomes, learning to predict position evaluation directly. The dual-head design shares the representational work of the residual tower while specializing the final layers for their distinct prediction tasks.

3.3.4 Layer Grouping for Selective Aggregation

To enable selective parameter sharing in our federated learning framework, we partition the neural network into functionally distinct layer groups. Each group represents a coherent set of parameters that can be independently chosen for cluster-specific or cross-cluster aggregation. This grouping reflects both the hierarchical structure of the network and the hypothesis that different layers may benefit differently from exposure to diverse playing styles.

We define five layer groups (Figure 3.2). The **input block** comprises the initial 3×3 convolutional layer, batch normalization, and activation that transforms the 119-plane input representation into 256-channel feature maps. This group contains the parameters that process raw board encodings into a learned feature space. The **early residual blocks** group includes residual blocks 1 through 6, which learn fundamental spatial patterns and piece relationships. The **middle residual blocks** group contains blocks 7 through 13, which learn tactical patterns and multi-piece coordination. The **late residual blocks** group encompasses blocks 14 through 19, which integrate strategic concepts and high-level position evaluation.

The final two groups separate the output heads. The **policy head** group contains all parameters involved in move prediction, including the policy-specific convolution, fully connected layers, and softmax activation. The **value head** group contains all parameters for position evaluation, including the value-specific convolution, hidden layer, and final value output. This separation allows independent decisions about whether move selection patterns and position evaluation should be shared across playstyle clusters.

This five-group partition provides sufficient granularity to test hypotheses about which network components benefit from cross-cluster knowledge transfer. Early layers that learn universal chess patterns might benefit from aggregation across all clients regardless of playstyle. Middle and later layers that encode tactical and strategic preferences might require cluster-specific aggregation to preserve distinct playing styles. The policy and value heads might show different sensitivity to cross-cluster aggregation, as move preferences may be more playstyle-dependent than outcome predictions. The grouping enables these hypotheses to be tested empirically through controlled experiments with different selective aggregation configurations.

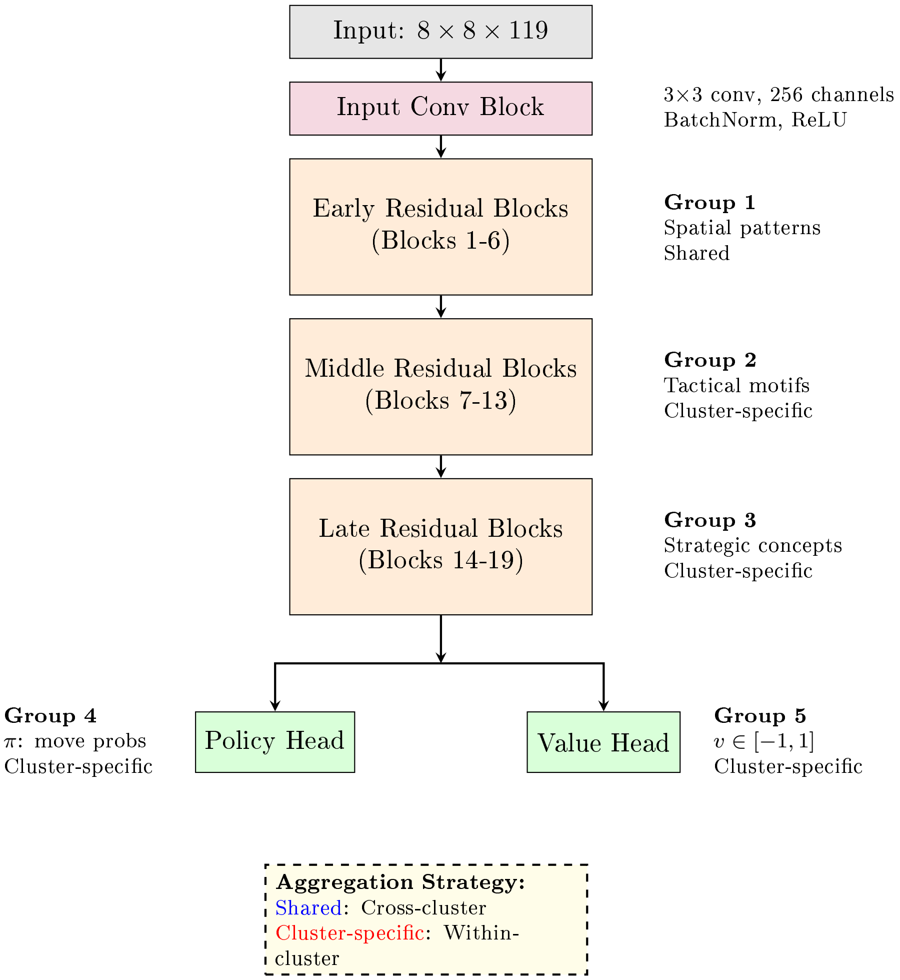


Figure 3.2: Neural network architecture showing the input layer, residual tower with 19 blocks grouped into early, middle, and late stages, and dual output heads for policy and value prediction. Annotations indicate the five layer groups used for selective aggregation, with example aggregation strategies shown (shared for early layers, cluster-specific for middle and late layers and output heads).

3.4 Three-Tier Aggregation System

Our federated learning framework employs a three-tier hierarchical aggregation mechanism that progressively combines knowledge from individual clients to cluster-specific models to cross-cluster shared representations. This hierarchical approach balances the benefits of distributed learning with the need to preserve playstyle-specific characteristics within each cluster. The three tiers operate at different frequencies and scopes: local training occurs continuously at each client, intra-cluster aggregation periodically combines updates within each playstyle cluster, and inter-cluster selective aggregation occasionally shares specific layer groups across clusters. Figure 3.3 illustrates the complete aggregation pipeline and the flow of information through the three tiers.

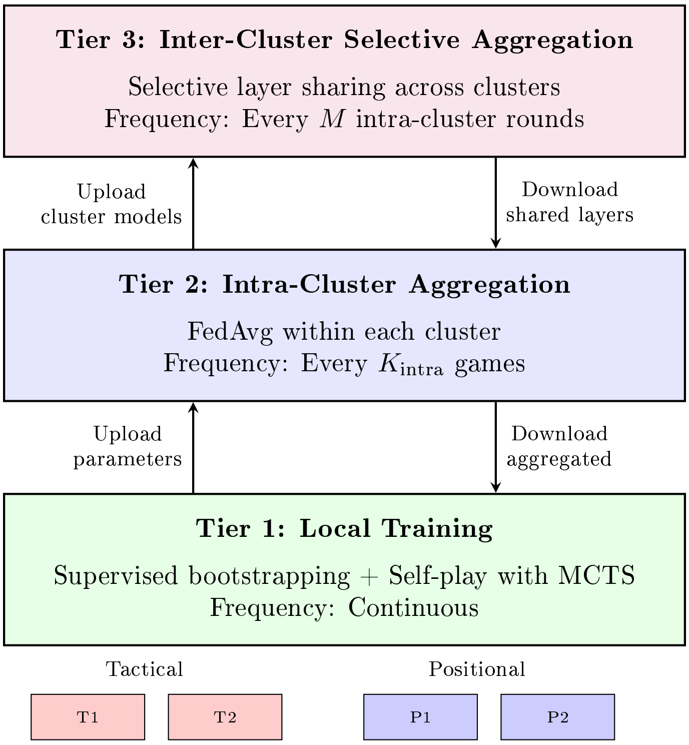

Tier 3: Inter-Cluster Selective Aggregation
Selective layer sharing across clusters
Frequency: Every M intra-cluster rounds
Upload cluster models
Download shared layers
Tier 2: Intra-Cluster Aggregation
FedAvg within each cluster
Frequency: Every K_{intra} games
Upload parameters
Download aggregated
Tier 1: Local Training
Supervised bootstrapping + Self-play with MCTS
Frequency: Continuous
Tactical
Positional
T1
T2
P1
P2

Figure 3.3: Three-tier hierarchical aggregation system showing the flow of information from local client training through intra-cluster aggregation to inter-cluster selective sharing. Arrows indicate bidirectional communication between tiers, with clients uploading parameters and downloading aggregated models. The time scales reflect the hierarchical nature of the system, with local training running continuously and higher tiers executing progressively less frequently.

3.4.1 Local Training Phase

At the base tier, each client independently trains its neural network through a two-phase approach: supervised bootstrapping followed by reinforcement learning through self-play. This local training phase generates the diverse experiences and parameter updates that will ultimately be aggregated across the federation.

Training begins with a supervised bootstrapping phase where clients learn from historical games and tactical puzzles filtered according to their cluster's playstyle. Tactical cluster clients train on games featuring aggressive openings and tactical puzzles emphasizing combinations, while positional cluster clients train on strategic games and positional puzzles. This phase provides a foundation of chess knowledge and playstyle-specific patterns before transitioning to self-play. The bootstrapping phase and data filtering mechanisms are detailed in Sections 3.6 and 3.7.

Once bootstrapped, clients transition to self-play reinforcement learning. Each client maintains a complete copy of the neural network and uses it to play games against itself. For each move, the client performs Monte Carlo Tree Search guided by the current network's policy and value predictions. The search explores promising move sequences by repeatedly selecting moves, expanding the search tree, evaluating positions with the neural network, and backpropagating values through visited nodes. After completing the search, the client selects a move based on the visit counts of root actions, which represents an improved policy over the raw network output.

Training data is generated from these self-play games. Each position encountered during a game is stored along with the improved policy distribution derived from MCTS visit counts and the final game outcome. After accumulating a batch of training positions, the client trains its network by minimizing a combined loss function. The policy loss uses cross-entropy between the network's policy output and the MCTS-improved policy. The value loss uses mean squared error between the network's value prediction and the actual game outcome. The combined loss is $L = L_{\text{policy}} + \lambda L_{\text{value}}$, where λ balances the two objectives.

The local training phase continues for a fixed number of games or training steps before the client's updated parameters are transmitted to the aggregation server for intra-cluster aggregation. The complete training procedure, including MCTS parameters and experience replay mechanisms, is described in Section 3.7.

3.4.2 Intra-Cluster Aggregation

The second tier of aggregation combines parameter updates from clients within each playstyle cluster to create a cluster-specific global model. This intra-cluster aggregation preserves the specialized characteristics of each playstyle while leveraging the collective learning of multiple clients pursuing similar strategic goals.

When clients complete a local training phase, they transmit their updated model parameters to the central aggregation server. The server maintains separate aggregation contexts for each cluster, ensuring that tactical and positional clients do not directly share parameters at this stage. For each cluster, the server applies federated averaging [13] to compute a weighted mean of client parameters. Let $\theta_i^{(t)}$ denote the parameters of client i in cluster c at aggregation round t, and let n_i represent the number of training examples processed by client i since the last aggregation. The cluster-specific aggregated parameters are computed as:

$$\theta_c^{(t+1)} = \frac{\sum_{i \in c} n_i \theta_i^{(t)}}{\sum_{i \in c} n_i} \tag{3.3}$$

This weighted averaging gives greater influence to clients that have processed more training data, under the assumption that more training leads to better parameter estimates. The aggregation is applied uniformly across all layer groups at this stage, creating a complete cluster-specific model that represents the collective knowledge of all clients in that cluster.

After aggregation, the server distributes the updated cluster-specific model $\theta_c^{(t+1)}$ back to all clients in cluster c. Each client replaces its local parameters with the aggregated parameters and resumes local training from this synchronized state. This synchronization allows clients to benefit from the diverse experiences of other clients in their cluster while maintaining their specialized playstyle focus. The frequency of intra-cluster aggregation is determined by the aggregation scheduling policy described in Section 3.4.4.

3.4.3 Inter-Cluster Selective Aggregation

The third and highest tier of aggregation selectively shares knowledge across playstyle clusters. Unlike intra-cluster aggregation which combines all parameters, inter-cluster aggregation operates only on specific layer groups identified as benefiting from cross-cluster knowledge transfer. This selective approach enables the system to learn universal chess patterns while preserving cluster-specific strategic preferences.

Inter-cluster aggregation is controlled by a layer group selection policy that specifies which of the five layer groups (input block, early residual blocks, middle residual blocks, late residual blocks, policy head, value head) should be aggregated across clusters. Based on the hypothesis that early layers learn universal patterns while later layers encode playstyle-specific strategies, a typical configuration might designate the input block and early residual blocks for cross-cluster sharing while keeping middle blocks, late blocks, and output heads cluster-specific.

For each layer group designated for cross-cluster aggregation, the server computes a global average across all clusters. Let $\theta_{c,g}^{(t)}$ denote the parameters of layer group g in cluster c at inter-cluster aggregation round t. The cross-cluster aggregated parameters for group g are:

$$\theta_g^{(t+1)} = \frac{\sum_c n_c \theta_{c,g}^{(t)}}{\sum_c n_c} \tag{3.4}$$

where n_c represents the total number of training examples processed by all clients in cluster c since the last inter-cluster aggregation. This ensures that clusters contributing more training data have proportionally greater influence on the shared representation.

After computing the cross-cluster averaged parameters for selected layer groups, the server distributes these shared parameters back to all clusters. Each cluster's model is updated by replacing the parameters of shared layer groups with the cross-cluster averaged values, while keeping cluster-specific layer groups unchanged. This selective replacement maintains the architectural integrity of the network while enabling knowledge transfer for designated components.

Inter-cluster aggregation occurs less frequently than intra-cluster aggregation, as it represents a higher-level consolidation of knowledge. The reduced frequency also mitigates the risk of disrupting cluster-specific learning by limiting how often cross-cluster information is injected into specialized models. The specific timing and frequency are determined by the aggregation scheduling policy detailed in the next subsection.

3.4.4 Aggregation Scheduling

The three aggregation tiers operate on different time scales to balance learning efficiency with communication overhead and model stability. The scheduling policy determines when each tier executes and coordinates the flow of information through the hierarchical system.

Local training at Tier 1 runs continuously, with each client playing self-play games and updating its neural network parameters through gradient descent. Clients operate asynchronously without waiting for other clients or the server. This continuous local training ensures that computation resources are fully utilized and that learning progresses without interruption.

Intra-cluster aggregation at Tier 2 occurs periodically when clients have accumulated sufficient local training progress. In our implementation, clients perform intra-cluster aggregation after every K_{intra} self-play games, where K_{intra} is a hyperparameter controlling the aggregation frequency. After completing K_{intra} games, a client uploads its current parameters to the server and waits for the server to perform aggregation and return the updated cluster model. The client then resumes training with the aggregated parameters.

This periodic synchronization prevents clients from diverging too far from the cluster's collective knowledge while allowing substantial local progress between synchronizations.

Inter-cluster aggregation at Tier 3 occurs less frequently, after every M intra-cluster aggregation rounds, where $M > 1$. This reduced frequency reflects the fact that cross-cluster knowledge transfer involves higher-level patterns that evolve more slowly than cluster-specific learning. The ratio M controls the balance between cluster specialization and cross-cluster knowledge sharing. Larger values of M allow clusters to develop more distinct characteristics before sharing knowledge, while smaller values promote more frequent integration of universal patterns.

The scheduling policy creates a natural hierarchy of time scales: local training operates on the scale of individual games (minutes), intra-cluster aggregation operates on the scale of training batches (tens of games), and inter-cluster aggregation operates on the scale of multiple aggregation rounds (hundreds of games). This hierarchical timing allows the system to efficiently combine rapid local learning with periodic consolidation at increasing levels of abstraction.

3.5 Selective Layer Aggregation

The selective layer aggregation mechanism is the key innovation that enables our framework to balance universal chess knowledge with playstyle-specific strategies. Rather than applying uniform federated averaging across all network parameters, we partition the network into functional layer groups and selectively choose which groups to aggregate across clusters. This section details the layer sharing strategy, the algorithmic implementation, the knowledge transfer mechanisms, and the expected convergence properties.

Figure 3.4 illustrates how different layer groups are treated during inter-cluster aggregation, with shared layers receiving cross-cluster knowledge transfer while cluster-specific layers remain isolated. Figure 3.5 presents the experimental configurations we evaluate, showing different hypotheses about which layers benefit from sharing..

3.5.1 Layer Sharing Strategy

The layer sharing strategy determines which of the five layer groups (input block, early residual blocks, middle residual blocks, late residual blocks, policy head, value head) undergo cross-cluster aggregation versus remaining cluster-specific. This decision reflects hypotheses about the hierarchical nature of chess knowledge representation in deep neural networks.

We evaluate our selective aggregation approach against two baseline configurations. B1 represents fully independent training with no cross-cluster sharing, serving as a lower bound on knowledge transfer. B2 represents standard federated learning with complete parameter sharing across all layers, serving as an upper bound on knowledge transfer but potentially sacrificing playstyle preservation. Our selective configurations (P1-P4) explore the middle ground between these extremes.

Configuration P1 designates the input block and early residual blocks as shared layers while keeping middle blocks, late blocks, and both output heads cluster-specific. This configuration embodies our primary hypothesis that early layers learn universal low-level patterns applicable to all chess positions regardless of playstyle, such as basic piece relationships, attack and defense patterns, and elementary tactical motifs. These fundamental patterns should benefit from training data across all playstyles, as they represent chess knowledge that transcends strategic preferences.

Middle and late residual blocks are kept cluster-specific because they learn increasingly abstract and strategic representations. Middle blocks that learn tactical patterns

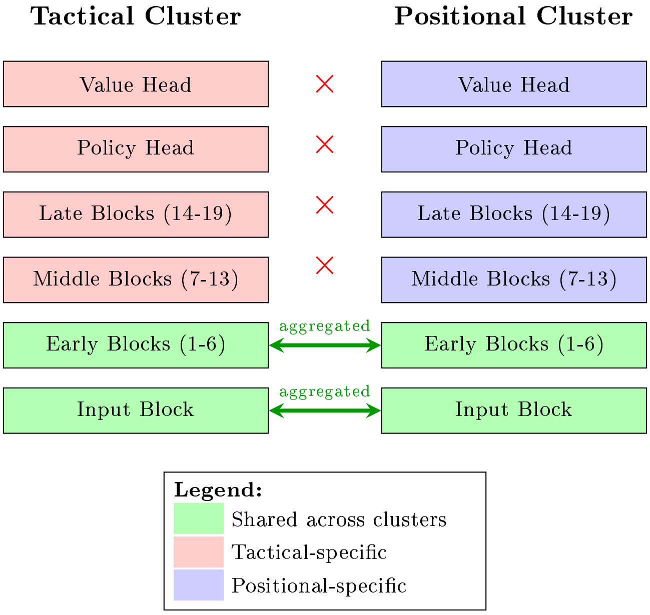
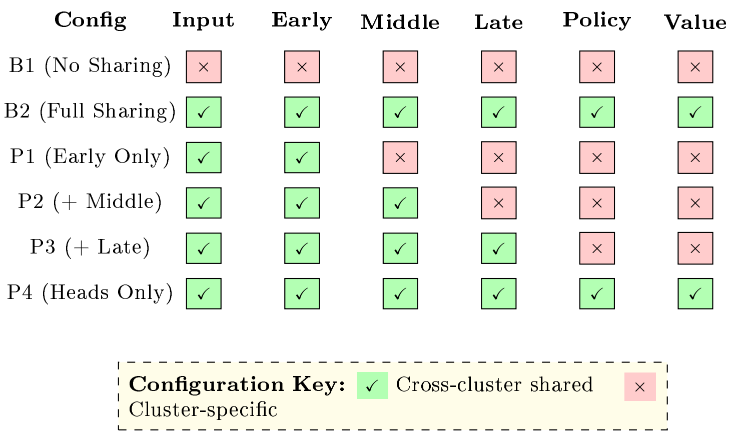


Figure 3.4: Selective layer sharing visualization showing how different layer groups are treated during inter-cluster aggregation in the baseline configuration (B1). Green layers (input block and early residual blocks) are aggregated across clusters, receiving cross-cluster knowledge transfer. Red and blue layers (middle blocks, late blocks, and output heads) remain cluster-specific, preserving strategic preferences. Arrows indicate cross-cluster aggregation; X marks indicate isolated layers.



Config	Input	Early	Middle	Late	Policy	Value
B1 (No Sharing)	×	×	×	×	×	×
B2 (Full Sharing)	✓	✓	✓	✓	✓	✓
P1 (Early Only)	✓	✓	×	×	×	×
P2 (+ Middle)	✓	✓	✓	×	×	×
P3 (+ Late)	✓	✓	✓	✓	×	×
P4 (Heads Only)	✓	✓	✓	✓	✓	✓

Configuration Key: ✓ Cross-cluster shared × Cluster-specific

Figure 3.5: Experimental layer sharing configurations tested to evaluate selective aggregation hypotheses. B1 and B2 are baseline configurations with no sharing (independent clusters) and full sharing (standard federated learning) respectively. P1-P4 represent selective sharing configurations that progressively increase the number of shared layer groups to test hypotheses about which layers benefit from cross-cluster aggregation. Green checkmarks indicate shared layers; red X marks indicate cluster-specific layers.

may differ between clusters that emphasize aggressive piece activity versus solid defensive structures. Late blocks that integrate strategic evaluation may encode fundamentally different position assessment criteria between tactical and positional clusters. Maintaining separate parameters for these layers allows each cluster to develop specialized strategic understanding appropriate to its playstyle.

The output heads are cluster-specific because they directly encode move selection preferences and position evaluation. The policy head in a tactical cluster should favor sacrifices, attacks, and dynamic imbalances, while the policy head in a positional cluster should favor prophylaxis, structure, and long-term advantages. Similarly, the value head's assessment of position quality depends on strategic criteria that differ between playstyles. Keeping these heads separate ensures that the final predictions reflect cluster-specific strategic judgment.

3.5.2 Weight Aggregation Algorithm

The selective weight aggregation algorithm extends standard federated averaging to operate independently on different layer groups. The algorithm maintains separate aggregation logic for shared and cluster-specific layers, ensuring that cross-cluster knowledge transfer occurs only where desired.

Let $\mathcal{L}_{\text{shared}}$ denote the set of layer groups designated for cross-cluster sharing, and let $\mathcal{L}_{\text{specific}}$ denote the layer groups maintained cluster-specifically. For our baseline configuration, $\mathcal{L}_{\text{shared}} = \{\text{input block}, \text{early residual blocks}\}$ and $\mathcal{L}_{\text{specific}} = \{\text{middle residual blocks}, \text{late residual block}$

During inter-cluster aggregation at round t, the server receives cluster-specific models $\theta^{(t)}_{\text{tactical}}$ and $\theta^{(t)}_{\text{positional}}$ from the intra-cluster aggregation tier. For each shared layer group $g \in \mathcal{L}_{\text{shared}}$, the server computes the cross-cluster average:

$$\theta_g^{(t+1)} = \frac{n_{\text{tactical}}\theta^{(t)}_{\text{tactical},g} + n_{\text{positional}}\theta^{(t)}_{\text{positional},g}}{n_{\text{tactical}} + n_{\text{positional}}} \tag{3.5}$$

where n_c represents the total training examples processed by cluster c since the last inter-cluster aggregation. This weighted average reflects the relative contributions of each cluster to the shared representation.

For cluster-specific layer groups $g \in \mathcal{L}_{\text{specific}}$, no cross-cluster aggregation occurs. Each cluster's parameters remain unchanged:

$$\theta^{(t+1)}_{\text{tactical},g} = \theta^{(t)}_{\text{tactical},g}, \quad \theta^{(t+1)}_{\text{positional},g} = \theta^{(t)}_{\text{positional},g} \tag{3.6}$$

The server then constructs updated cluster models by combining shared and cluster-specific parameters. For each cluster c, the updated model is:

$$\theta_c^{(t+1)} = \bigcup_{g \in \mathcal{L}_{\text{shared}}} \theta_g^{(t+1)} \cup \bigcup_{g \in \mathcal{L}_{\text{specific}}} \theta_{c,g}^{(t+1)} \tag{3.7}$$

This composite model contains cross-cluster averaged parameters for shared layers and cluster-preserved parameters for specific layers. The server distributes these updated models back to their respective clusters, where they replace the cluster-specific models produced by intra-cluster aggregation.

3.5.3 Knowledge Transfer Mechanism

The selective aggregation mechanism enables a specific form of knowledge transfer where universal chess patterns propagate across clusters while strategic preferences remain isolated. This transfer occurs through the shared layer parameters, which act as a common foundation upon which cluster-specific specializations are built.

Shared early layers learn feature representations that apply across all training data, regardless of cluster origin. When a tactical client discovers an effective pattern for detecting knight forks and a positional client learns to recognize weak square complexes, both patterns become encoded in the shared early layer parameters through cross-cluster aggregation. Subsequent training in both clusters can then build upon this expanded pattern library, even though individual clients never directly observe the other cluster's training games.

The knowledge transfer is asymmetric in depth. Low-level patterns in shared layers benefit from the full diversity of training experiences across all clusters. Middle-layer tactical motifs and late-layer strategic concepts remain cluster-specific, allowing each cluster to develop specialized higher-level representations on top of the shared foundation. The policy and value heads, which directly determine move selection and position evaluation, receive no cross-cluster influence and purely reflect cluster-specific strategic preferences.

This hierarchical transfer mechanism aims to capture the intuition that chess knowledge has both universal and style-dependent components. Basic patterns like piece mobility, king safety threats, and material imbalances apply universally and should be learned from diverse data. Strategic concepts like acceptable pawn weaknesses, piece activity versus structure trade-offs, and long-term versus short-term thinking vary with playstyle and should be learned within specialized clusters. The selective aggregation architecture embodies this hierarchical separation.

3.5.4 Convergence Properties

The selective aggregation approach introduces complexity to the convergence analysis compared to standard federated learning. Cluster-specific layers converge to solutions that minimize loss over their cluster's data distribution, while shared layers converge to solutions that minimize loss over the combined distribution of all clusters.

For cluster-specific layer groups, convergence follows standard federated averaging analysis within each cluster. Since these layers never receive cross-cluster updates, each cluster's specific layers converge to optima for their local data distribution. The tactical cluster's late layers and output heads will converge to parameters optimal for tactical positions, while positional cluster layers converge to parameters optimal for positional play.

Shared layer convergence is more complex because these layers receive gradients from diverse data distributions during local training but are synchronized across clusters during inter-cluster aggregation. The shared layers will converge toward parameters that minimize the weighted average of losses across both clusters' data distributions. If tactical and positional training data contain common underlying patterns that benefit from similar low-level representations, the shared layers should converge to parameters that effectively encode these universal patterns. If the distributions are too different and require contradictory low-level features, the shared layers may converge to a compromise solution that serves neither cluster optimally.

The success of selective aggregation depends on the hypothesis that early layers genuinely learn distribution-agnostic patterns. If this hypothesis holds, sharing these layers accelerates convergence by pooling diverse training experiences. If it fails, forcing these layers to be shared may slow convergence or degrade performance. The experimental evaluation examines this hypothesis empirically by comparing selective aggregation against fully independent and fully shared baselines.

3.6 Playstyle-Aware Data Filtering

Establishing distinct playstyle characteristics within each cluster requires careful curation of training data. Rather than allowing clients to train on arbitrary chess positions, we implement a data filtering pipeline that directs tactical training data to the tactical cluster and positional training data to the positional cluster. This filtering occurs at multiple stages: game selection based on opening classification, puzzle selection based on tactical themes, and client assignment within clusters. Figure 3.6 illustrates the complete dual-pipeline architecture for data filtering and distribution..

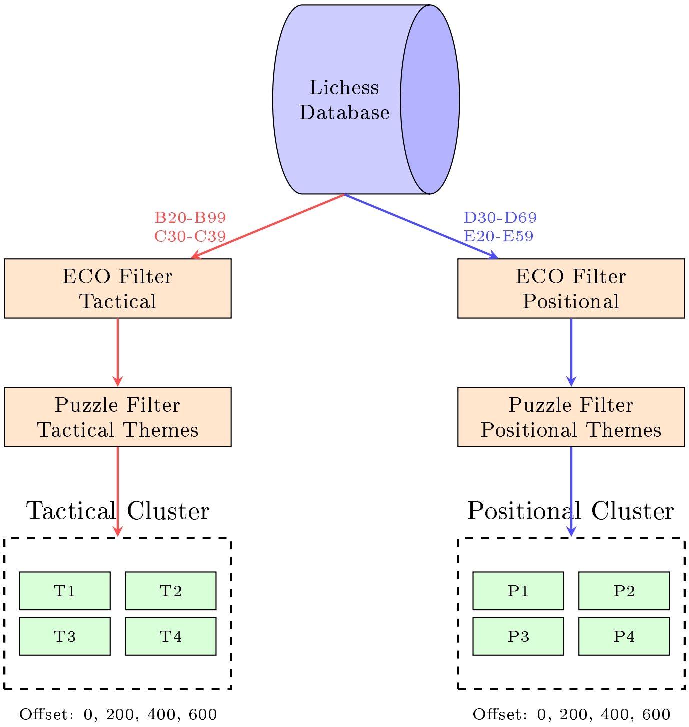


Figure 3.6: Playstyle-aware data filtering pipeline showing dual pathways from the Lichess database to cluster-specific clients. Games are filtered by ECO opening codes (tactical openings like Sicilian Dragon vs positional openings like Queen's Gambit), combined with puzzle theme filtering (tactical combinations vs endgame positions), and distributed to clients within each cluster using offset-based sampling to ensure non-overlapping training data.

3.6.1 ECO Opening Code Classification

Chess openings are classified using the Encyclopedia of Chess Openings (ECO) code system, which assigns alphanumeric codes from A00 to E99 to opening variations based on the initial moves. We leverage this classification to identify games with tactical versus positional characteristics, based on the strategic nature of the opening played.

Our ECO classification divides openings into two categories. Tactical openings emphasize sharp positions, early attacks, and dynamic imbalances. These include the Sicilian Defence (B20-B99) with variations like the Dragon, Najdorf, and Sveshnikov that lead to opposite-side castling and racing attacks. The King's Gambit (C30-C39) sacrifices a pawn for rapid development and attacking chances. The Italian Game's aggressive lines (C50-C54) pursue early initiative. Alekhine's Defence (B02-B05) provokes central pawn advances to create tactical targets. The Vienna Game (C25-C29) aims for rapid piece activity and central control with tactical opportunities.

Positional openings emphasize long-term structural advantages, piece coordination, and strategic maneuvering. The Queen's Gambit Declined (D30-D69) establishes a solid pawn structure and methodical development. The Slav Defence (D10-D19) maintains central solidity while preparing queenside expansion. The Nimzo-Indian Defence (E20-E59) controls the center with pieces rather than pawns, emphasizing strategic complexity. The Queen's Indian Defence (E12-E19) develops harmoniously while maintaining flexibility. The Catalan Opening (E00-E09) combines central control with fianchetto development. The English Opening (A10-A39) and Réti Opening (A04-A09) emphasize hypermodern principles of central control from a distance.

During game loading, each game's ECO code is extracted from the PGN header and normalized to its base three-character form, ignoring suffix variations. Games without ECO codes or with unclassified openings are assigned to the positional category by default, as unclassified openings tend to be quieter systems. This classification ensures that tactical cluster clients train primarily on games featuring sharp, concrete positions, while positional cluster clients train on games emphasizing strategic planning and structural understanding.

3.6.2 Puzzle Type Filtering

In addition to game-based training, we incorporate tactical puzzle training to reinforce pattern recognition and concrete calculation skills. The Lichess puzzle database contains over 3 million positions tagged with thematic labels indicating the tactical or strategic patterns present. We filter these puzzles by theme to align with each cluster's playstyle focus.

Tactical cluster puzzles emphasize concrete combinations and forcing sequences. Selected themes include fork (attacking two pieces simultaneously), pin (immobilizing a piece defending a more valuable piece), skewer (forcing a valuable piece to move and exposing a piece behind it), discovered attack (revealing an attack by moving a blocking piece), sacrifice (surrendering material for positional or attacking compensation), attacking f2/f7 (exploiting weak squares near the king), double check (checking with two pieces simultaneously preventing king moves), deflection (forcing a piece away from a critical defensive task), attraction (forcing a piece to an unfavorable square), and clearance (vacating a square for tactical purposes). These themes train pattern recognition for tactical opportunities that arise in sharp positions.

Positional cluster puzzles emphasize strategic understanding and endgame technique. Selected themes include endgame (positions with few pieces requiring precise technique), advantage (converting a favorable position into a win), crushing (positions with overwhelming advantages), mate (checkmate sequences), mateIn2 and mateIn3 (checkmate puzzles with specified move counts), and specific endgame types such as queen-rook endgames, bishop endgames, pawn endgames, and rook endgames. These themes develop strategic pattern recognition for converting advantages and understanding fundamental endgame principles.

Puzzle filtering operates on both theme and rating. Each puzzle in the database has a difficulty rating from approximately 1500 to 2500. We filter puzzles to match the target training difficulty, typically setting minimum rating at 1800 to ensure the puzzles contain meaningful patterns rather than simple one-move tactics. Theme filtering uses set intersection: a puzzle passes the filter if any of its assigned themes appears in the cluster's theme whitelist. This allows puzzles with mixed themes to be used as long as they contain at least one relevant pattern.

3.6.3 Cluster Assignment Strategy

After filtering games and puzzles by playstyle, the filtered data must be distributed to individual clients within each cluster. Our assignment strategy ensures that clients within the same cluster train on different data to maximize the diversity of experiences contributing to intra-cluster aggregation, while maintaining playstyle consistency within each cluster.

We employ an offset-based sampling strategy to partition the filtered dataset among clients. Let N denote the number of clients per cluster and G denote the number of games (or puzzles) each client processes per training round. For training round r and client index i within a cluster, the data offset is computed as:

$$\text{offset}(r, i) = r \cdot (N \cdot G) + i \cdot G \tag{3.8}$$

This formula ensures that in each round, the N clients access disjoint sequential segments of the dataset. In round 0, client 0 accesses samples 0 through $G-1$, client 1 accesses samples G through $2G-1$, and so on. In round 1, all clients advance by $N \cdot G$ positions to access fresh data. This deterministic offset calculation guarantees no overlap within a cluster across clients or rounds, while allowing clients in different clusters to access the same absolute offsets (since they draw from different filtered datasets).

The offset strategy supports training resumption without data repetition. If training is interrupted and resumed from round r_{resume}, a round offset parameter r_{offset} is added to the effective round number in the offset calculation. This shifts all clients forward in the dataset by $(r_{\text{resume}} + r_{\text{offset}}) \cdot (N \cdot G)$ positions, ensuring that resumed training uses entirely new data rather than repeating positions from earlier rounds.

For our configuration with $N = 4$ clients per cluster and $G = 200$ games per round, round 0 distributes offsets 0, 200, 400, and 600 to the four clients. Round 1 distributes offsets 800, 1000, 1200, and 1400. Over training, each cluster collectively processes $4 \times 200 = 800$ unique games per round, with no client seeing the same position twice across the entire training trajectory.

3.6.4 Data Distribution Balance

Maintaining balanced data distribution across clusters is essential for fair comparison and effective learning. Imbalances could arise if one playstyle category contains significantly fewer games or puzzles in the database, potentially limiting that cluster's learning progress or biasing comparisons between clusters.

The Lichess database contains millions of games spanning all ECO codes, providing ample data for both tactical and positional categories. Our ECO classification identifies approximately 150 tactical opening codes and 150 positional codes, ensuring roughly balanced representation. Empirical analysis of a sample Lichess database reveals that tactical openings (particularly Sicilian Defence variations) and positional openings (particularly Queen's Gambit and Indian Defence systems) appear with comparable frequency in high-rated play, mitigating concerns about severe category imbalance.

The puzzle database similarly contains sufficient coverage across tactical and positional themes. Tactical combination puzzles are abundant due to their popularity and the ease of constructing forcing sequences. Endgame puzzles, while less numerous, still number in the hundreds of thousands, providing more than adequate training data for our purposes. Rating distribution is approximately uniform across the 1500-2500 range, ensuring both clusters can access puzzles at appropriate difficulty levels.

To monitor balance during training, we track the number of games and puzzles processed by each cluster and verify that both clusters consume data at comparable rates. If imbalances emerge, we can adjust the games-per-round parameter differently for each cluster or modify filtering criteria to broaden the data pool for underrepresented categories.

In practice, the large scale of available data and the balanced nature of ECO classification make such interventions unnecessary.

3.7 Training Procedures

The training pipeline combines supervised learning from human games with reinforcement learning through self-play, following the AlphaZero paradigm [21] adapted for federated learning with playstyle preservation. Training proceeds in two phases: an initial supervised bootstrapping phase that provides the neural network with basic chess knowledge, followed by a self-play phase that refines playing strength through reinforcement learning with Monte Carlo Tree Search.

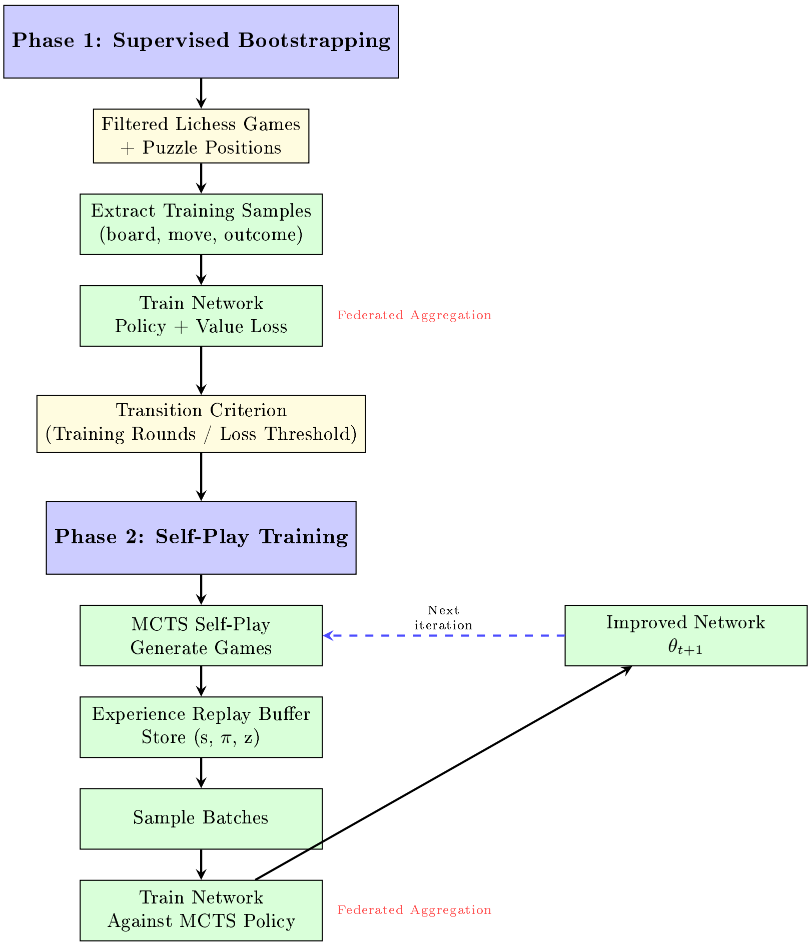

Phase 1: Supervised Bootstrapping
Filtered Lichess Games + Puzzle Positions
Extract Training Samples (board, move, outcome)
Train Network Policy + Value Loss
Federated Aggregation
Transition Criterion (Training Rounds / Loss Threshold)
Phase 2: Self-Play Training
MCTS Self-Play Generate Games
Next iteration
Improved Network θ_{t+1}
Experience Replay Buffer Store (s, π, z)
Sample Batches
Train Network Against MCTS Policy
Federated Aggregation

Figure 3.7: Training pipeline flowchart showing the transition from supervised bootstrapping to self-play reinforcement learning. The supervised phase trains on filtered human games and puzzles, while the self-play phase uses MCTS to generate training data. Both phases incorporate federated aggregation at tier boundaries.

3.7.1 Supervised Bootstrapping Phase

The supervised bootstrapping phase initializes the neural network with chess knowledge extracted from high-quality human games and tactical puzzles. This phase provides the network with a foundation in legal move generation, positional evaluation, and basic strategic principles before transitioning to self-play reinforcement learning.

Training data consists of positions extracted from the filtered Lichess game database and puzzle database as described in Section 3.6. Each training sample comprises a board position encoded as a $119 \times 8 \times 8$ tensor (Section 3.3), the move played in that position encoded as an action index from 0 to 4671, and the game outcome $z \in \{-1, 0, +1\}$ from the perspective of the player to move. For game positions, the outcome reflects the final result of the game. For puzzle positions, the outcome is set to +1 since puzzles represent winning positions by construction.

The supervised training objective minimizes a combined loss function over the policy and value heads. Let $\mathbf{p}$ denote the policy network's output probability distribution over moves, and let v denote the value network's scalar output. For a training sample with board state s, target move a^*, and target outcome z, the loss function is:

$$L_{\text{sup}}(s, a^*, z) = L_{\text{policy}}(\mathbf{p}(s), a^*) + L_{\text{value}}(v(s), z) \tag{3.9}$$

where the policy loss uses cross-entropy to match the played move:

$$L_{\text{policy}}(\mathbf{p}, a^*) = -\log p_{a^*} \tag{3.10}$$

and the value loss uses mean squared error to match the game outcome:

$$L_{\text{value}}(v, z) = (v - z)^2 \tag{3.11}$$

During supervised training, each client processes a disjoint segment of the filtered dataset determined by the offset-based sampling strategy (Section 3.6). For training round r with N clients per cluster and G games per round, client i accesses samples at offset $(r \cdot N \cdot G) + (i \cdot G)$. This ensures that clients within a cluster train on different data each round, maximizing the diversity of experiences contributing to federated aggregation while maintaining playstyle consistency.

The network is optimized using the Adam optimizer with an initial learning rate of 0.003. A learning rate scheduler monitors the training loss and reduces the learning rate by a factor of 0.5 if the loss plateaus for 15 consecutive rounds, with a minimum learning rate of 10^{-6}. This adaptive scheduling allows the network to make rapid initial progress while fine-tuning as training stabilizes.

After each local training round, clients send their updated model parameters to the cluster server for intra-cluster aggregation via Federated Averaging (Section 3.4). Every tenth round, inter-cluster selective aggregation shares knowledge between tactical and positional clusters while preserving playstyle-specific representations in cluster-specific layers.

The supervised bootstrapping phase continues for a predefined number of training rounds or until the training loss falls below a threshold indicating sufficient chess knowledge acquisition. Typical configurations run 100-200 supervised rounds before transitioning to self-play, though this can be adjusted based on loss convergence and preliminary playing strength evaluation.

3.7.2 Self-Play Training Phase

Following supervised bootstrapping, the training pipeline transitions to self-play reinforcement learning, where the neural network improves by playing games against itself with Monte Carlo Tree Search acting as a policy improvement operator. This phase follows the AlphaZero paradigm [21], generating training data through search-guided play rather than relying on external game databases.

In each self-play iteration, the current neural network f_θ with parameters θ plays games against itself using MCTS to select moves. For each position s encountered during self-play, MCTS runs a fixed number of simulations (typically 800-1600) to construct a search tree exploring possible continuations. The MCTS visit counts at the root node define an improved policy $\boldsymbol{\pi}$ that is typically stronger than the raw neural network policy $\mathbf{p}(s)$ due to explicit lookahead search.

Move selection during self-play uses a temperature parameter τ to control exploration. After running MCTS at position s, the visit counts $N(s, a)$ for each legal action a are converted to a probability distribution:

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}} \tag{3.12}$$

where τ controls the degree of exploration. A temperature of $\tau = 1$ produces proportional sampling from visit counts, encouraging exploration of diverse continuations. A temperature of $\tau \to 0$ (in practice, $\tau = 0.01$) makes move selection deterministic, always choosing the most-visited action. Following AlphaZero, we set $\tau = 1$ for the first 30 moves of each game to explore opening diversity, then reduce to $\tau = 0.01$ for the remainder to exploit the network's strongest continuations.

Each self-play game generates a sequence of training samples $(s_t, \boldsymbol{\pi}_t, z)$ where s_t is the board position at move t, $\boldsymbol{\pi}_t$ is the MCTS-improved policy at that position, and $z \in \{-1, 0, +1\}$ is the final game outcome. All positions from a single game share the same outcome value, reflecting the Monte Carlo principle that every position along a trajectory leads to the same terminal result.

The self-play training objective minimizes the loss between the neural network's predictions and the MCTS-derived targets. For a training sample $(s, \boldsymbol{\pi}, z)$ drawn from the replay buffer, the loss function is:

$$L_{\text{self}}(s, \boldsymbol{\pi}, z) = (z - v(s))^2 - \boldsymbol{\pi}^T \log \mathbf{p}(s) + \lambda \|\theta\|^2 \tag{3.13}$$

where the first term is the mean squared error between the value prediction $v(s)$ and the game outcome z, the second term is the cross-entropy loss between the policy prediction $\mathbf{p}(s)$ and the MCTS policy $\boldsymbol{\pi}$, and the third term is L2 regularization with coefficient λ (typically 10^{-4}) to prevent overfitting.

This loss function trains the neural network to imitate the MCTS search results: the policy head learns to match MCTS visit distributions (which incorporate lookahead), and the value head learns to predict game outcomes observed through self-play. Over many iterations, the network internalizes patterns discovered by search, becoming stronger without explicit search and enabling MCTS to search more effectively in subsequent iterations.

In the federated setting, self-play games are generated independently by each client using the current cluster-aggregated model. Clients within the same cluster produce diverse self-play trajectories due to stochastic move sampling (when $\tau = 1$) and different MCTS random seeds. After generating a batch of self-play games and training on the resulting positions, clients send updated parameters to the cluster server for aggregation, maintaining the same federated learning workflow as the supervised phase.

The self-play phase continues indefinitely, with the network progressively strengthening through the iterative cycle of game generation, training, and aggregation. Periodic evaluation against fixed-strength opponents (Section 5.5) monitors playing strength to assess training progress and compare selective aggregation configurations.

3.7.3 Monte Carlo Tree Search Integration

Monte Carlo Tree Search serves as the policy improvement operator during self-play, using explicit lookahead to find stronger moves than the neural network policy alone. MCTS constructs a search tree incrementally through simulated trajectories, each consisting of four phases: selection, expansion, simulation, and backpropagation.

The selection phase traverses the tree from the root position using a variant of the Upper Confidence Bound for Trees (UCT) algorithm. At each internal node representing position s, the algorithm selects the child action a that maximizes the PUCT (Polynomial Upper Confidence Trees) score:

$$\text{PUCT}(s,a) = Q(s,a) + c_{\text{puct}} \cdot P(s,a) \cdot \frac{\sqrt{N(s)}}{1 + N(s,a)} \tag{3.14}$$

where $Q(s,a)$ is the mean action-value (average outcome from simulations that selected action a in position s), $P(s,a)$ is the prior probability from the neural network policy, $N(s)$ is the total visit count of position s, $N(s,a)$ is the visit count of action a, and c_{puct} is an exploration constant (typically 1.0 to 4.0) that balances exploitation of high-value moves against exploration of uncertain moves with high neural network prior.

This formula combines the exploitation term $Q(s,a)$, which favors actions with high observed value, with the exploration term $c_{\text{puct}} \cdot P(s,a) \cdot \sqrt{N(s)}/(1+N(s,a))$, which favors actions with high neural network prior $P(s,a)$ that have been visited infrequently relative to the parent node. The exploration bonus decreases as $N(s,a)$ grows, gradually shifting from prior-guided exploration to value-guided exploitation.

Selection continues until reaching a leaf node: either a position not yet expanded in the search tree or a terminal position (checkmate, stalemate, or draw by repetition/insufficient material). For terminal positions, the exact game outcome is returned immediately. For non-terminal leaf positions, the expansion phase adds the position to the search tree and evaluates it using the neural network. The network's policy output $\mathbf{p}(s)$ initializes the prior probabilities $P(s,a)$ for all legal actions from s, and the value output $v(s)$ provides an estimated outcome without further search.

AlphaZero [21] eliminates the traditional rollout simulation phase, instead using the neural network's value prediction $v(s)$ as the leaf evaluation. This constitutes the simulation phase: rather than playing out the position to a terminal state, the network's learned value function estimates the expected outcome from s under optimal play.

The backpropagation phase propagates the evaluation $v(s)$ up the tree along the trajectory that reached the leaf. For each position-action pair (s,a) along the path, the visit count $N(s,a)$ is incremented and the mean action-value $Q(s,a)$ is updated:

$$Q(s,a) \leftarrow \frac{N(s,a) \cdot Q(s,a) + v}{N(s,a) + 1} \tag{3.15}$$

where v is the evaluation (negated appropriately for alternating players). This running average incorporates the new evaluation into the action-value estimate, influencing future selection decisions.

After completing the specified number of MCTS simulations (e.g., 800 simulations per move), the visit counts $N(s,a)$ at the root position define the improved policy $\boldsymbol{\pi}$ used for training and move selection. The repeated selection-expansion-backpropagation cycles concentrate search effort on promising continuations, with the neural network priors guiding initial exploration and the accumulated value estimates refining the search as simulations progress.

Key MCTS hyperparameters include the number of simulations per move (balancing playing strength against computational cost), the exploration constant c_{puct} (controlling the exploration-exploitation trade-off), Dirichlet noise parameters for root exploration (encouraging opening diversity), and virtual loss for parallelization (allowing multiple simulations to run concurrently without redundant exploration). These parameters are tuned based on playing strength evaluation and computational constraints.

3.7.4 Experience Replay and Batch Generation

Training samples generated during self-play are stored in an experience replay buffer, enabling efficient batch formation and decorrelating consecutive training updates. The replay buffer serves as a sliding window over recent self-play games, balancing the need to train on up-to-date positions (reflecting the current network strength) against the need for diverse training data (preventing overfitting to recent games).

Each entry in the replay buffer consists of a tuple $(s, \boldsymbol{\pi}, z)$ where s is a board position encoded as a $119 \times 8 \times 8$ tensor, $\boldsymbol{\pi}$ is the MCTS visit count distribution converted to a probability vector of length 4672, and z is the final game outcome. When a self-play game completes, all positions from that game are added to the buffer with the shared outcome value. This differs from traditional reinforcement learning where each state-action pair might have a distinct bootstrapped value estimate.

The replay buffer has a fixed maximum capacity, typically storing 500,000 to 1,000,000 positions. When the buffer reaches capacity, the oldest positions are evicted in FIFO order to make room for new self-play data. This ensures that training data remains representative of the current playing strength while retaining sufficient diversity to prevent catastrophic forgetting of previously learned patterns.

During training, batches are sampled uniformly at random from the replay buffer. Each training iteration draws a batch of size 32 to 64 positions, computes the forward pass through the neural network to obtain policy and value predictions, calculates the loss against the stored MCTS targets, and performs a gradient descent step to update the network parameters. Uniform random sampling breaks the temporal correlation between consecutive positions in a game, reducing variance in gradient estimates and stabilizing training.

Augmentation techniques can be applied during batch sampling to increase data efficiency. For chess, positions can be mirrored horizontally (flipping the board left-to-right) if the position is symmetric, effectively doubling the training data. However, care must be taken with castling rights and en passant squares, which break horizontal symmetry. Rotation and other geometric augmentations are not applicable to chess due to the asymmetric starting position and pawn movement rules.

In the federated learning setting, each client maintains its own local replay buffer populated with self-play games generated using the current cluster-aggregated model. Clients do not share raw experience tuples (which would require transmitting large amounts of position data); instead, they share only the updated neural network parameters after training on their local replay buffers. This preserves privacy and reduces communication overhead while allowing knowledge transfer through model aggregation.

The ratio of self-play games generated to training steps performed is a critical hyperparameter. AlphaZero [21] generates many self-play games per network update to ensure the replay buffer is populated with diverse high-quality data. Typical configurations might generate 1,000 to 5,000 self-play games per training iteration, with each game contributing 80-120 positions on average, yielding hundreds of thousands of training samples per iteration. The network then trains on batches sampled from this pool for multiple epochs before generating new self-play games with the updated network.

This iterative cycle of game generation, buffer population, batch sampling, and network training continues throughout the self-play phase, with federated aggregation occurring at regular intervals to incorporate knowledge from all clients within a cluster and selectively share knowledge across clusters. The experience replay mechanism ensures training stability and data efficiency while the federated aggregation mechanism ensures collaborative learning with playstyle preservation.

Chapter 4

Implementation

4.1 Technology Stack

The system is implemented entirely in Python, leveraging a modern ecosystem of libraries for deep learning, distributed systems, and chess domain logic. This section describes the core technologies and frameworks that form the foundation of our implementation.

4.1.1 Programming Language and Runtime

The implementation uses Python 3.12 as the primary programming language. Python's extensive ecosystem for machine learning, combined with its support for asynchronous programming through the asyncio library, makes it well-suited for implementing both the neural network training components and the distributed communication infrastructure. The asynchronous capabilities are particularly crucial for the server-client communication protocol, enabling non-blocking message handling and concurrent operation across multiple training nodes.

4.1.2 Deep Learning Framework

Neural network implementation relies on PyTorch 2.9.0, specifically a nightly development build with CUDA 12.8 support for GPU acceleration. PyTorch was selected over alternatives like TensorFlow due to its dynamic computation graph, which provides greater flexibility during model development and debugging, and its more Pythonic API that integrates naturally with the rest of the codebase. The nightly build provides access to the latest optimizations and features, including improved memory management for large residual networks and enhanced support for distributed training primitives.

The PyTorch ecosystem also includes torchaudio 2.8.0 and torchvision 0.24.0, though these are primarily included as dependencies rather than actively used in the current implementation. The core AlphaZeroNet implementation uses standard PyTorch modules including nn.Module, nn.Conv2d, nn.BatchNorm2d, nn.Linear, and functional operations from torch.nn.functional. Model serialization leverages PyTorch's native torch.save() and torch.load() functions, which use Python's pickle protocol for state dictionary persistence.

4.1.3 Distributed Communication

The federated learning coordination infrastructure uses WebSockets 12.0+ for bidirectional communication between the aggregation server and training clients. WebSockets provide full-duplex communication channels over a single TCP connection, enabling efficient message exchange without the overhead of repeated HTTP request-response cycles. The implementation uses Python's native asyncio-based websockets library, which integrates seamlessly with asynchronous server and client code.

The WebSocket protocol supports both text and binary message types, though our implementation primarily uses JSON-encoded text messages for control flow (start training, model updates, cluster assignments) and base64-encoded binary payloads for model parameter transmission. This design choice prioritizes debugging clarity and protocol simplicity over marginal bandwidth improvements from pure binary encoding.

4.1.4 Chess Domain Logic

Chess-specific functionality is provided by python-chess 1.999, a comprehensive library for chess move generation, validation, and board representation. The library handles all game rule enforcement, including complex cases like castling rights, en passant captures, threefold repetition detection, and the fifty-move rule. This removes the need to implement chess logic from scratch and ensures correctness through a well-tested, widely-used library.

The python-chess library also provides PGN (Portable Game Notation) parsing capabilities for loading training games from Lichess databases, FEN (Forsyth-Edwards Notation) encoding and decoding for position representation, and integration with external chess engines through the UCI (Universal Chess Interface) protocol. The latter capability enables evaluation matches against Stockfish, the classical chess engine used for playing strength assessment.

4.1.5 Data Processing and Storage

Data manipulation and numerical computation use numpy 2.3.3+ and pandas 2.3.3+. NumPy provides the fundamental array operations used in the board encoder for constructing the 119-plane input representation, while pandas facilitates loading and filtering game databases with SQL-like operations on tabular metadata (player ratings, opening codes, game results).

For handling large compressed game databases, the implementation uses zstandard 0.25.0+, which provides fast decompression of Zstandard-compressed PGN files. Lichess distributes monthly game databases in .pgn.zst format, and zstandard decompression enables streaming access to millions of games without requiring prior extraction to uncompressed files. This significantly reduces storage requirements and I/O overhead during data loading.

Optional distributed caching is provided through Redis 6.4.0+, a high-performance in-memory data store. Redis can be used to cache preprocessed game positions across multiple training nodes, reducing redundant computation when multiple clients process the same game database. However, Redis is not required for basic operation, and the system functions correctly with local file-based caching when Redis is unavailable.

4.1.6 Configuration and Logging

System configuration uses YAML files parsed by PyYAML 6.0.0+. YAML's human-readable syntax simplifies manual editing of configuration files while supporting hierarchical structure for nested configuration objects. The implementation defines server configuration (aggregation parameters, evaluation settings), cluster topology (node assignments, playstyle labels), and per-node training settings (batch size, learning rate, data paths) in separate YAML files, promoting modularity and enabling experimentation with different configurations without code changes.

Structured logging is implemented using loguru 0.7.3+, which provides a more ergonomic API than Python's standard logging module. Loguru supports context binding for attaching metadata (round number, cluster ID, node ID) to log messages, automatic log rotation based on file size or time, and flexible formatting including colorized console output for development debugging. All server and client components use loguru for event logging, error reporting, and performance tracking.

4.1.7 Testing and Development Tools

The test suite uses pytest 7.0.0+ as the testing framework, with extensions including pytest-asyncio 0.21.0+ for testing asynchronous code, pytest-mock 3.10.0+ for mocking dependencies, pytest-timeout 2.1.0+ for preventing hanging tests, and pytest-cov 4.0.0+ for code coverage measurement. Development tools include black 23.0.0+ for code formatting, isort 5.12.0+ for import sorting, flake8 6.0.0+ for linting, and mypy 1.0.0+ for static type checking, though these are optional development dependencies rather than runtime requirements.

Performance profiling capabilities are provided by psutil 5.9.0+ for system resource monitoring and memory-profiler 0.60.0+ for detailed memory usage analysis. These tools help identify bottlenecks in data loading, model serialization, and aggregation operations.

4.1.8 External Chess Engine

Model evaluation requires Stockfish, a classical chess engine that provides calibrated opponents at different skill levels. Stockfish is not included as a Python dependency but must be installed separately and accessible via the system PATH. The implementation communicates with Stockfish through the UCI protocol using python-chess's engine integration module. Stockfish provides both move suggestions and position evaluations used in playstyle metrics computation, particularly the delta/tipping point metric that requires engine analysis of move alternatives.

4.1.9 Optional GUI Components

The system includes an optional graphical interface implemented with PyQt6 6.9.1+ for visualizing games and training progress. However, the GUI components are not required for core functionality, and the system operates entirely through command-line interfaces and configuration files in headless server environments. The GUI primarily serves as a development and debugging tool rather than a production requirement.

4.2 Server Implementation

The server-side implementation coordinates all aspects of the federated learning process, from managing node connections to orchestrating aggregation rounds. This section describes the core server components that enable distributed training coordination.

4.2.1 Training Orchestrator

The `TrainingOrchestrator` class in `server/main.py` serves as the central coordinator for the federated learning training loop. This class integrates all server subsystems, communication, aggregation, evaluation, and storage, into a unified training workflow that executes repeatedly across multiple rounds.

The orchestrator manages two primary configuration structures. The `RoundConfig` dataclass defines parameters for each training round, including the aggregation threshold (default 0.8, requiring 80% of nodes to participate), timeout for waiting on node responses (300-1200 seconds depending on training mode), weighting strategies for intra-cluster (`"samples"` or `"uniform"`) and inter-cluster aggregation, and the shared versus cluster-specific layer patterns that control selective aggregation. The EvaluationConfig dataclass specifies evaluation parameters, including whether evaluation is enabled, the interval between evaluations (every 10 rounds by default), the number of games to play per Stockfish ELO level (default 10), the set of ELO levels to test against (typically [1000, 1200, 1400]), time allocated per move (0.1 seconds), and settings for delta analysis including Stockfish search depth (12 plies).

The run_training() method implements the main training loop, executing for a specified number of rounds or until manually interrupted. For each round, the orchestrator calls _execute_round(), which implements a six-phase workflow. First, it broadcasts START_TRAINING messages to all registered nodes, instructing them to begin local training with the current cluster model. Second, it collects MODEL_UPDATE messages from nodes, waiting until either the aggregation threshold is met (e.g., 8 out of 10 nodes have responded) or the timeout expires. Third, it performs intra-cluster aggregation using the IntraClusterAggregator, combining model updates from nodes within each cluster independently (Section 3.4). Fourth, it performs inter-cluster selective aggregation using the InterClusterAggregator, sharing only specified layers across clusters while preserving cluster-specific parameters (Section 3.4). Fifth, it broadcasts CLUSTER_MODEL messages back to nodes with the updated aggregated models specific to each cluster. Sixth, it logs comprehensive metrics, saves model checkpoints at configured intervals, and optionally runs playstyle evaluation against Stockfish if the evaluation interval has elapsed.

The orchestrator maintains state across rounds, tracking the current round number, a starting round offset for resume training scenarios, cluster-specific models as PyTorch state dictionaries, and the current run identifier for organizing stored artifacts. Integration with the storage subsystem through FileExperimentTracker enables automatic logging of training metrics, aggregation statistics, evaluation results, and model checkpoints. The orchestrator handles graceful shutdown through signal handlers that respond to SIGINT and SIGTERM, ensuring that in-progress rounds complete and final checkpoints are saved before termination.

4.2.2 Cluster Management

The ClusterManager class in server/cluster_manager.py manages the topology and state of all clusters and their constituent nodes. Each cluster is represented by a Cluster dataclass that encapsulates cluster metadata: a unique cluster identifier (e.g., "cluster_tactical"), the associated playstyle label (e.g., "tactical" or "positional"), the target node count, a node prefix for auto-generating node IDs (e.g., "agg"), a human-readable description, the number of games to train per round (cluster-specific setting), and an optional path to an initial model checkpoint for resume training.

Node management within clusters uses three sets to track node states. The expected_nodes set contains all node IDs that should register based on the configured node count, generated automatically with sequential numbering (e.g., agg_001, agg_002, ..., agg_010 for a 10-node cluster). The active_nodes set tracks currently connected and responsive nodes that have successfully registered and maintain active WebSocket connections. The inactive_nodes set tracks nodes that previously registered but have since disconnected or timed out. This three-set architecture enables the cluster manager to distinguish between nodes that have never connected, nodes currently participating in training, and nodes temporarily offline but expected to rejoin.

Cluster topology is defined declaratively through cluster_topology.yaml, which specifies all clusters, their playstyles, node counts, and training parameters. The ClusterManager parses this YAML configuration at server startup, automatically generating expected node IDs and initializing cluster state. Node registration follows a validation protocol: when a node sends a REGISTER message, the ClusterManager checks whether the node

ID matches one of the expected nodes for any cluster. If valid, the node is added to the cluster's active set and assigned to that cluster's playstyle. If the node ID is unrecognized, registration is rejected with an error message.

Readiness checking determines whether training can proceed for a given round. The `check_cluster_readiness()` method compares the number of active nodes against the configured threshold for each cluster. For example, with a threshold of 0.8 and 10 expected nodes per cluster, at least 8 nodes must be active in each cluster for training to proceed. This ensures sufficient participation for meaningful aggregation while tolerating a limited number of offline or disconnected nodes. The readiness check also validates that cluster models exist for all clusters, preventing training from starting with missing model state.

Node disconnection handling updates cluster state when WebSocket connections close. The node transitions from `active_nodes` to `inactive_nodes`, and its last activity timestamp is recorded. The `ClusterManager` does not automatically remove inactive nodes, allowing them to rejoin seamlessly if they reconnect. This design accommodates transient network failures and node restarts without requiring re-registration or manual intervention.

4.2.3 Server Communication

The server communication layer, implemented in `server/communication/server_socket.py`, provides an asynchronous WebSocket server that manages bidirectional communication with training clients. Built on Python's `asyncio` and the `websockets` library, the `FederatedLearningServer` class handles concurrent connections from potentially dozens of nodes while maintaining non-blocking operation.

The communication protocol defined in `server/communication/protocol.py` uses a message-based architecture with strongly-typed message enumerations. The `MessageType` enum defines all valid message types, divided into three categories. Client-to-server messages include `REGISTER` (node registration request with node ID and capabilities), `MODEL_UPDATE` (trained model parameters plus training metrics like samples processed and loss), `METRICS` (training metrics without model weights), and `HEARTBEAT` (keep-alive signal for connection health monitoring). Server-to-client messages include `REGISTER_ACK` (registration confirmation with assigned cluster ID), `START_TRAINING` (command to begin training round with configuration parameters), `CLUSTER_MODEL` (aggregated model distribution specific to the node's cluster), and `REQUEST_MODEL` (request for a node's current model state). Bidirectional messages include `ERROR` (error notification with error code and description) and `DISCONNECT` (graceful disconnection notice).

Message serialization uses JSON for the message envelope (type, timestamp, node ID, metadata) and base64-encoded binary data for model parameters. The `Message` dataclass provides a structured representation with type safety and automatic validation. The `MessageFactory` class offers factory methods for creating typed messages with correct field structure, reducing errors in message construction and improving code maintainability.

Connection management uses `asyncio` event loops to handle multiple concurrent WebSocket connections. Each connected node has a dedicated coroutine that listens for incoming messages and routes them to appropriate handlers based on message type. The server maintains a connection registry mapping node IDs to active WebSocket connections, enabling targeted message delivery when broadcasting cluster-specific models.

Timeout enforcement protects against nodes that disconnect or hang during training. When collecting `MODEL_UPDATE` messages during a round, the orchestrator sets an `asyncio` timeout (typically 300-1200 seconds depending on expected training duration). If a node fails to respond within the timeout period, it is excluded from that round's aggregation. The aggregation threshold mechanism allows training to proceed even when some nodes are slow or offline, as long as the minimum participation requirement is met (e.g., 80% of nodes). This design balances robustness against stragglers with the need for sufficient data diversity in aggregation.

Error handling follows an exception-based model with structured logging. Communication errors (connection closed, malformed messages, timeout) are caught, logged with context binding (node ID, cluster ID, round number), and converted to `ERROR` messages when appropriate. The server does not crash on individual node failures but continues operating with the remaining healthy nodes, logging failures for post-hoc analysis.

4.3 Client Implementation

The client-side implementation enables individual training nodes to participate in federated learning by coordinating local training, communication with the server, and model state management. This section describes the core client components that execute training rounds and maintain connection with the server.

4.3.1 Federated Learning Node

The `FederatedLearningNode` class in `client/node.py` serves as the primary orchestrator for all client-side activities. This class integrates the communication layer with the trainer implementation through a composition-based architecture, separating concerns between network operations (`FederatedLearningClient`), training execution (`TrainerInterface`), and overall workflow coordination (the node itself).

Node lifecycle is managed through the `NodeLifecycleState` enumeration, which defines seven distinct states. The `INITIALIZING` state occurs during node construction and setup before connection establishment. The `READY` state indicates successful connection and registration, awaiting training commands. The `TRAINING` state marks active local training in progress. The `UPDATING` state represents the period when aggregated models are being received and loaded. The `IDLE` state indicates the node is connected but not actively training, typically waiting between rounds. The `ERROR` state signals unrecoverable failures requiring manual intervention. Finally, the `SHUTDOWN` state indicates graceful node termination. Figure 4.1 illustrates the complete state machine with all possible transitions.

The node's `start()` method implements the main execution loop. First, it initiates the WebSocket connection through the communication client. Second, it waits for successful registration with a 30-second timeout, transitioning to `ERROR` state if connection fails. Third, it enters the main event loop that monitors for completed training tasks and handles asynchronous message callbacks. Fourth, it maintains this loop until either `is_running` becomes false or an unrecoverable error occurs. Fifth, it ensures graceful shutdown through the `stop()` method, which cancels ongoing training, disconnects from the server, and logs final statistics.

Message handling is configured during initialization through `_setup_message_handlers()`, which registers callbacks for each message type. The `START_TRAINING` handler in `_handle_start_training()` extracts training parameters from the message payload, updates the trainer configuration if `games_per_round` has changed, sets the current round number for offset calculation (used by trainers to select non-overlapping data), and spawns a background task executing `_run_training()` with the current model state. The `CLUSTER_MODEL` handler in `_handle_cluster_model()` deserializes the aggregated model state using `PyTorchSerializer`, replaces the node's current model state, explicitly frees the old model state and calls garbage collection to prevent memory accumulation, and transitions to `IDLE` state ready for the next round. The `REGISTER_ACK` handler confirms successful registration and transitions to `READY`. The `ERROR` handler logs server errors and transitions to `ERROR` state.

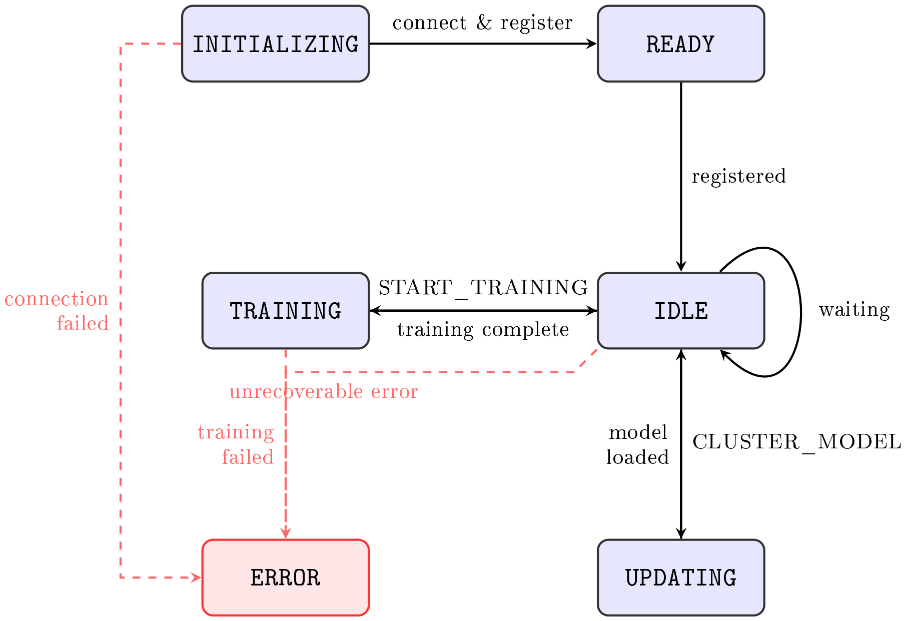


Figure 4.1: Node lifecycle state machine. Solid arrows represent normal state transitions triggered by server messages or training completion. Dashed red arrows indicate error transitions. The `stop()` method can be called from any state to transition to `SHUTDOWN`.

Training completion is handled asynchronously by the main event loop, which monitors `self.training_task` for completion. Upon detecting a completed task, the node calls `_handle_training_complete()` with the `TrainingResult`. This method updates node statistics (rounds completed, total training time, total samples), calls `client.send_model_update()` to transmit the updated model state to the server, calls `client.send_metrics()` to transmit additional training metrics (loss, accuracy, policy loss, value loss), and transitions back to `IDLE` state. Memory management is critical during this process: the method explicitly deletes the training result object and invokes garbage collection to prevent memory leaks during long training runs with hundreds of rounds.

Node statistics are tracked continuously and exposed through `get_statistics()`, which returns a comprehensive dictionary including node ID and cluster ID, current lifecycle state, connection status, rounds completed and current round number, total training time and total samples processed, node uptime since start, trainer-specific statistics from `trainer.get_statistics()`, and client communication statistics from `client.get_stats()`. Final statistics are logged automatically during shutdown through `_log_final_statistics()`, providing a complete summary of the node's participation in federated learning.

4.3.2 Trainer Implementations

The trainer subsystem provides pluggable training implementations through the abstract `TrainerInterface` base class defined in `client/trainer/trainer_interface.py`. This interface defines the contract that all trainers must implement, enabling the node to execute different training strategies without modification to the coordination logic.

The `TrainerInterface` base class defines two critical dataclasses for configuration and results. The `TrainingConfig` dataclass encapsulates all training parameters: `games_per_round` (number of games or puzzles per training round, typically 100-500), `batch_size` (mini-batch size for gradient updates, typically 32-256), `learning_rate` (optimizer learning rate, typically 0.001-0.0001), `exploration_factor` (exploration coefficient for self-play, if applicable), max_game_length (maximum moves per game before forced draw), save_games (whether to persist game data for analysis), playstyle (optional playstyle filter for data selection, e.g., "tactical" or "positional"), and additional_params (dictionary for trainer-specific parameters like puzzle rating ranges or themes). The TrainingResult dataclass encapsulates training outcomes: model_state (updated model parameters as a dictionary, either raw state dict or serialized with serialized_data key), samples (number of training samples processed), loss (final training loss achieved), games_played (number of games or puzzles used), training_time (wall-clock training duration in seconds), metrics (dictionary of additional metrics like accuracy, policy loss, value loss, win rate), success (boolean indicating whether training completed successfully), and error_message (optional error description if training failed).

The abstract train() method must be implemented by all trainers. It accepts initial_model_state (starting model parameters, either raw or serialized) and returns a TrainingResult. The method signature is async def train(self, initial_model_state: Dict[str, Any]) -> TrainingResult, ensuring all trainers operate asynchronously to avoid blocking the node's event loop. The optional evaluate() method provides model evaluation capabilities, accepting model_state and num_games and returning a dictionary of evaluation metrics. Base class utility methods include update_config() for dynamically modifying training parameters between rounds, get_statistics() for retrieving trainer statistics (total games played, training time, average loss), and _add_to_history() for maintaining a history of training results across rounds.

The DummyTrainer implementation in client/trainer/trainer_dummy.py serves as a lightweight testing trainer that simulates training without actual computation. This trainer is valuable for pipeline validation, storage system testing, and rapid development iteration without chess engines. The train() method simulates training by sleeping for 500ms to mimic processing time, initializing a mock AlphaZero model structure if the initial model state is empty (with input convolution layers, residual blocks, policy head, and value head), making small random modifications to existing model weights (adding uniform random noise in range [-0.01, 0.01]), generating synthetic metrics (loss decreasing by 0.02 per round, accuracy increasing by 0.02 per round, random win rate), and returning a TrainingResult with the modified model and synthetic metrics. The _create_mock_alphazero_model() method creates a realistic model structure with proper layer naming conventions that match the actual AlphaZero architecture, ensuring compatibility with the selective aggregation system which distinguishes shared layers (input convolution, residual blocks) from cluster-specific layers (policy head, value head).

The SupervisedTrainer implementation in client/trainer/trainer_supervised.py bootstraps the AlphaZero network by training on high-quality human games from PGN databases. This trainer implements the supervised learning phase described in Section 3.7, filtering games by rating (minimum 2000 ELO) and playstyle (tactical or positional), and training both the policy head (via cross-entropy loss on moves) and value head (via MSE loss on game outcomes). Device management detects available hardware, using CUDA if available otherwise falling back to CPU. Encoder initialization creates BoardEncoder (converts board positions to 119-plane tensors) and MoveEncoder (converts moves to action indices in range 0-4671). Model serialization uses PyTorchSerializer with compression enabled and base64 encoding for JSON/WebSocket compatibility.

Sample extraction offset calculation ensures non-overlapping data across nodes and rounds through the formula: offset = (current_round + round_offset) * (nodes_per_cluster * games_per_round) + node_index * games_per_round. The round_offset parameter enables resume training by skipping already-processed data. For example, with 4 nodes per cluster and 100 games per round, round 0 assigns node 0 offset 0, node 1 offset 100, node 2 offset 200, node 3 offset 300. Round 1 assigns node 0 offset 400, node

1 offset 500, etc. If resuming from round 30 (`round_offset=30`), round 1 assigns node 0 offset 12400, ensuring previously used data is never reprocessed. The `_extract_node_index()` method parses the numeric suffix from node IDs (e.g., "agg_001" becomes index 0, "agg_002" becomes index 1), providing the zero-based index needed for offset calculation.

The training pipeline executes in several phases. First, `_initialize_model()` creates an `AlphaZeroNet` instance if none exists, deserializes the model state if provided (checking for `serialized_data` key), loads the state dict into the model, and creates the optimizer and learning rate scheduler only once (preserving Adam momentum and scheduler state across rounds). Second, `_extract_samples()` runs `SampleExtractor` in a thread pool to avoid blocking, filtering by playstyle, minimum rating (2000), and calculated offset, returning a list of `TrainingSample` objects with board positions, moves played, game outcomes, and position history. Third, a `ChessDataset` and `DataLoader` are created with the specified batch size and shuffling enabled. Fourth, `_train_epoch()` executes one training epoch: encoding boards to (119, 8, 8) tensors, encoding moves to integer class indices for cross-entropy loss, forward pass through the model producing policy logits and value predictions, loss computation (policy loss via cross-entropy, value loss via MSE, total loss as their sum), backward pass with gradient computation and optimizer step, periodic memory cleanup every 50 batches (calling `torch.cuda.empty_cache()` on GPU), and async sleep every batch to allow other tasks to run. Fifth, the learning rate scheduler (`ReduceLROnPlateau`) is stepped with the current loss, reducing the learning rate by half if loss plateaus for 15 consecutive rounds (patience=15), with a minimum learning rate of 1e-6. Sixth, the updated model state is serialized and packaged with metadata (framework, num_parameters), samples are explicitly deleted and garbage collected to free memory, and the current round counter is incremented for the next training call.

The `PuzzleTrainer` implementation in `client/trainer/trainer_puzzle.py` trains on tactical puzzles from the Lichess puzzle database, focusing the policy head on recognizing tactical motifs while leaving the value head to learn from supervised training on full games. This selective head training exploits the dual-head architecture described in Section 3.3. Redis integration provides fast puzzle access through `RedisPuzzleCache`, avoiding repeated CSV parsing. Puzzle filtering supports minimum and maximum rating constraints (default 1500-2500) and theme filtering (optional, e.g., only "fork", "pin", "skewer" puzzles). The training pipeline differs from supervised training in that only the policy head is trained (value head gradients are not backpropagated), puzzles can have multiple moves in the solution sequence (each move becomes a separate training sample), and the offset calculation follows the same formula as supervised training to ensure non-overlapping data.

The `_load_puzzles()` method loads puzzles from Redis using the calculated offset, applies rating and theme filters, and returns a list of `Puzzle` objects with puzzle ID, FEN string, move sequence (UCI format), rating, and themes. The `PuzzleDataset` extracts all positions from multi-move puzzle sequences: for a puzzle with moves [setup, move1, reply1, move2, reply2], it creates training samples for move1 and move2 (the player's moves, at odd indices), with each sample containing the board position, the correct move to find, and the position index in the sequence. The `_train_epoch()` method differs from supervised training by computing only policy loss (no value loss in backpropagation), monitoring value loss for logging purposes but not including it in gradient computation, and accumulating metrics that include policy loss, value loss (monitoring only), and total loss (policy only).

4.3.3 Client Communication

The `FederatedLearningClient` class in `client/communication/client_socket.py` manages all WebSocket communication with the federated learning server. This asynchronous client provides connection management, automatic reconnection with exponential backoff, message serialization and deserialization, and integration with the node lifecycle.

Client state is tracked through the ClientState enumeration with eight states: DISCONNECTED (no active connection), CONNECTING (connection attempt in progress), CONNECTED (WebSocket established but not yet registered), REGISTERING (registration message sent, awaiting acknowledgment), REGISTERED (fully registered and ready for training), TRAINING (local training in progress), UPLOADING (sending model update to server), and ERROR (unrecoverable error state). Connection statistics are tracked through the ConnectionStats dataclass, which records connection attempts, successful connections, total messages sent and received, total uptime, reconnection count, ping failures, keepalive timeouts, last ping time, message size errors (exceeding 500MB limit), and large message warnings (exceeding 100MB).

The start() method implements the main client loop with reconnection logic. The loop calls _connect_and_run() to establish connection and handle messages, catches connection errors and initiates reconnection if auto_reconnect is enabled, applies exponential backoff to the reconnection delay (starting at 10 seconds, multiplying by 1.5 each failure, capping at 300 seconds), and increments the reconnection counter. This loop continues until is_running becomes false or auto_reconnect is disabled, ensuring the client remains connected throughout long training sessions spanning hours or days.

Connection establishment in _connect_and_run() proceeds through several phases. First, it connects to the WebSocket server using the websockets library with ping interval of 60 seconds (automatic keepalive), ping timeout of 45 seconds (tolerance for network latency), maximum message size of 500MB (for large model updates), and close timeout of 20 seconds. Second, it transitions to CONNECTED state and resets the reconnection delay to 10 seconds on successful connection. Third, it spawns two concurrent tasks: _message_loop() for receiving messages and _register_with_server() for registration. Fourth, it awaits registration completion with a 30-second timeout. Fifth, it starts _heartbeat_loop() after successful registration. Sixth, it continues handling messages until disconnection. Exception handling includes catching ConnectionClosed exceptions with special handling for "keepalive ping timeout" (reduces reconnection delay, logs warning, increments timeout counter) and "message too big" errors (logs error suggesting compression, increments size error counter), catching InvalidURI exceptions and transitioning to ERROR state, catching OSError exceptions for network errors, and ensuring cleanup in the finally block (updating uptime stats, canceling heartbeat task, closing WebSocket).

Registration with the server uses _register_with_server(), which transitions to REGISTERING state, creates a REGISTER message using MessageFactory, sends the message via _send_message(), waits for REGISTER_ACK response with a 30-second timeout using _wait_for_message_type(), validates the response payload checking the success field, transitions to REGISTERED state on success or ERROR state on failure, and raises an exception if registration fails or times out. The _wait_for_message_type() utility creates an asyncio.Future and registers it in self.pending_responses keyed by message type, waits for the future with the specified timeout, and cleans up the pending response entry on timeout or completion.

The message loop in _message_loop() iterates over incoming WebSocket messages using async for raw_msg in self.websocket, increments the received message counter, parses the JSON message using Message.from_json(), validates the message structure, routes the message to _handle_message(), catches JSON decode errors for malformed messages, catches general exceptions to prevent loop termination, and handles ConnectionClosed exceptions with special logging for keepalive timeouts and size errors. Message routing in _handle_message() first checks for built-in handlers (REGISTER_ACK, START_-TRAINING, CLUSTER_MODEL, ERROR, DISCONNECT), then checks for external handlers registered via set_message_handler() (allows node to customize handling), and finally checks pending_responses for futures awaiting specific message types, resolving any matching futures with the received message.

Model update transmission through send_model_update() handles serialization and network transmission of trained models. The method transitions to UPLOADING state, checks if the model state is already serialized (has serialized_data key) and uses it directly, otherwise serializes using PyTorchSerializer with compression enabled and base64 encoding, packages the serialized model with metadata (framework, compression, encoding), creates a MODEL_UPDATE message using MessageFactory with model state, samples, loss, and round number, sends the message and logs the transmission, transitions back to REGISTERED state on success, and catches exceptions, transitions to ERROR state, and re-raises. The heartbeat mechanism in _heartbeat_loop() sends periodic HEARTBEAT messages every 45 seconds (configurable), checks that the WebSocket is still connected before sending, updates last_ping_time statistics on successful send, increments ping_failures on send errors, and terminates the loop if the WebSocket closes or the state is no longer REGISTERED.

Message size monitoring is implemented in _send_message() to detect potential issues before transmission. The method encodes the JSON message to UTF-8 to calculate byte size, logs a warning if size exceeds 100MB (incrementing large_message_warnings counter), logs info if size exceeds 10MB, logs trace with exact size in KB for normal messages, sends the message via websocket.send(), increments the sent message counter, and catches exceptions, logs errors, and re-raises. This monitoring helps identify models that may benefit from parameter differencing or additional compression, particularly important when training large models with millions of parameters across hundreds of rounds.

4.4 Neural Network Implementation

The neural network implementation follows the AlphaZero architecture described in Section 3.3, realized as PyTorch modules in client/trainer/models/alphazero_net.py. This section describes the implementation-specific details including module structure, layer configurations, naming conventions for selective aggregation, and parameter initialization.

4.4.1 AlphaZeroNet PyTorch Module

The AlphaZeroNet class inherits from torch.nn.Module and implements the complete neural network architecture. The constructor accepts three configurable parameters: input_channels (default 119 for the full board representation), num_res_blocks (default 19 following the AlphaZero paper), and channels (default 256 for the residual tower width). These parameters enable network variants of different sizes for experimentation and resource-constrained environments.

The input convolution layer uses nn.Conv2d(input_channels, channels, kernel_size=3, padding=1, bias=False), transforming the 119-plane board representation into a 256-channel feature map while preserving the 8 × 8 spatial dimensions. Batch normalization via nn.BatchNorm2d(channels) follows the convolution to stabilize training, and ReLU activation introduces non-linearity. The bias=False parameter is critical because batch normalization includes a learnable bias term, making the convolutional bias redundant.

The residual tower is implemented using nn.ModuleDict rather than nn.ModuleList to ensure predictable layer naming. The dictionary uses string keys "0", "1", ..., "18" to create parameter names like residual.0.conv1.weight, residual.1.conv1.weight, etc. This naming convention is essential for the selective aggregation system, which distinguishes shared layers (residual blocks) from cluster-specific layers (policy and value heads) by parsing parameter names with regular expressions. The forward pass iterates through the dictionary in sorted numeric order using `sorted(self.residual.keys(), key=int)` to ensure deterministic execution order despite dictionary implementation details.

The policy and value heads are instantiated as `PolicyHead(channels)` and `ValueHead(channels)`, both receiving the same feature representation from the residual tower. The `forward()` method accepts an input tensor of shape (batch, 119, 8, 8), applies the input convolution and batch normalization, sequentially processes through all residual blocks, and returns a tuple `(policy_logits, value)` where policy logits have shape (batch, 4672) and value has shape (batch, 1). An additional `predict()` method wraps `forward()` with `torch.no_grad()` and applies softmax to policy logits, facilitating inference without gradient computation.

Network size variants are created through the factory function `create_alphazero_net()`, which provides convenient configurations: tiny (2 blocks, 64 channels for unit testing), small (5 blocks, 128 channels for rapid prototyping), medium (10 blocks, 256 channels for laptop training), full (19 blocks, 256 channels matching the AlphaZero paper), and large (40 blocks, 256 channels matching AlphaZero's final version). All variants use the same 119-plane input representation and 4672-action output space, ensuring compatibility across different model sizes during federated aggregation.

4.4.2 Residual Block Implementation

The `ResidualBlock` class implements the building block of the residual tower, following the pre-activation residual network design. Each block contains two convolutional paths and a skip connection. The first path applies `nn.Conv2d(channels, channels, kernel_size=3, padding=1, bias=False)` followed by `nn.BatchNorm2d(channels)` and ReLU activation. The second path applies an identical conv-bn sequence. The skip connection adds the input directly to the output of the second convolution before the final ReLU: `out = relu(bn2(conv2(relu(bn1(conv1(x))))) + x)`.

The skip connection architecture addresses the vanishing gradient problem in deep networks by providing a direct path for gradients to flow backward through the network. During backpropagation, gradients can bypass degraded layers through the skip connection, enabling training of networks with 19 or more residual blocks. The use of 3×3 convolutions with padding=1 maintains spatial dimensions throughout the block, ensuring the skip connection addition is dimension-compatible without requiring projection layers.

All convolutional layers use `bias=False` because subsequent batch normalization layers include learnable bias parameters. This reduces parameter count without affecting model expressiveness. Batch normalization computes running statistics (mean and standard deviation) during training and uses fixed statistics during evaluation, controlled automatically by PyTorch's `model.train()` and `model.eval()` modes. The forward pass signature `forward(self, x: torch.Tensor) -> torch.Tensor` accepts and returns tensors of shape (batch, channels, 8, 8), maintaining consistent dimensions for easy stacking.

4.4.3 Policy Head Implementation

The `PolicyHead` class implements the move prediction component using AlphaZero's spatial action encoding. The architecture begins with a 3×3 convolution `nn.Conv2d(in_channels, 73, kernel_size=3, padding=1, bias=False)` that reduces the 256-channel feature map to 73 planes. This differs from some implementations that use 1×1 convolutions; AlphaZero's paper specifies 3×3 convolutions to maintain spatial context during move prediction. Batch normalization and ReLU activation follow the convolution.

The output tensor of shape (batch, 73, 8, 8) encodes moves using the AlphaZero action space: each of the 64 squares has 73 possible move types. The 73 planes decompose as

56 queen-style moves (8 directions × 7 distances covering N, NE, E, SE, S, SW, W, NW directions with 1-7 square distances), 8 knight moves (L-shaped movements to all valid knight destinations), and 9 underpromotion moves (3 directions × 3 piece types covering left-diagonal, forward, and right-diagonal pawn promotions to knight, bishop, or rook). Standard pawn promotions to queen are encoded in the queen-style moves. This encoding scheme supports all legal chess moves including castling (encoded as king moves of 2 squares) and en passant (encoded as diagonal pawn captures).

The forward pass reshapes the output from (batch, 73, 8, 8) to (batch, 4672) through two operations. First, `out.permute(0, 2, 3, 1)` reorders dimensions to (batch, 8, 8, 73), placing the 73 move planes as the innermost dimension. Second, `out.reshape(out.size(0), -1)` flattens the spatial and plane dimensions to create a 1D vector of 4672 logits ($8 \times 8 \times 73 = 4672$). These logits are not normalized during the forward pass; softmax is applied externally during training (via `CrossEntropyLoss`) or inference (via `predict()`).

Illegal move masking occurs during Monte Carlo Tree Search or evaluation by setting logits of illegal moves to large negative values (e.g., -1e8) before softmax application, ensuring negligible probability mass on invalid actions. The policy head parameters are cluster-specific in the selective aggregation framework, allowing different playstyle clusters to learn distinct move preferences (tactical vs. positional) while sharing the feature extraction layers.

4.4.4 Value Head Implementation

The `ValueHead` class implements position evaluation through a two-stage architecture that progressively reduces spatial dimensions. The first stage applies a 1×1 convolution `nn.Conv2d(in_channels, 1, kernel_size=1, bias=False)` to compress the 256-channel feature map to a single channel, followed by batch normalization and ReLU. This produces a tensor of shape (batch, 1, 8, 8) representing a spatial saliency map over the board. The flattening operation `out.view(out.size(0), -1)` reshapes this to (batch, 64), treating each square's activation as an independent feature.

The second stage consists of two fully connected layers. The first layer `nn.Linear(64, 256)` expands the 64 spatial features to 256 hidden units, followed by ReLU activation. This expansion layer learns non-linear combinations of spatial features that correlate with position quality. The second layer `nn.Linear(256, 1)` reduces the hidden representation to a single scalar value. Finally, `torch.tanh()` bounds the output to the range $[-1, +1]$, where $+1$ represents a winning position for the current player, -1 represents a losing position, and 0 represents a drawn or balanced position.

The tanh activation is essential for compatibility with the game outcome labels used during training: wins are labeled as $+1$, losses as -1, and draws as 0. Mean squared error loss `nn.MSELoss()` between predicted values and game outcomes trains the network to predict position evaluation. During self-play, the value head guides tree search by estimating leaf node values, and during opening book learning, it learns to evaluate positions based on game results from the PGN database.

Similar to the policy head, value head parameters are cluster-specific in selective aggregation. This allows tactical players to develop value functions that prioritize tactical opportunities (piece activity, king safety, threats), while positional players develop value functions emphasizing long-term factors (pawn structure, space advantage, piece coordination). The shared residual trunk provides common feature extraction, while specialized value heads adapt evaluation criteria to playstyle-specific preferences.

4.5 Data Processing Implementation

The data processing pipeline transforms raw chess games and puzzles into training-ready tensors for the neural network. This section describes the encoding schemes, filtering mechanisms, and data extraction procedures implemented in the `data/` module that convert PGN databases into supervised learning samples.

4.5.1 Board Encoder

The `BoardEncoder` class in `data/board_encoder.py` implements the 119-plane tensor representation that serves as neural network input (Section 3.3). This encoding follows the AlphaZero specification [21], capturing not only the current board state but also temporal context through position history. The `encode()` method accepts a `chess.Board` object and optional history list, returning a NumPy array of shape (119, 8, 8) with `float32` dtype.

Planes 0-95 encode piece positions across 8 temporal steps (current position plus 7 historical positions). Each temporal step uses 12 planes: 6 piece types (pawn, knight, bishop, rook, queen, king) × 2 colors (white, black). Within each plane, squares containing the relevant piece are set to 1.0, empty squares to 0.0. The board is represented from white's perspective using python-chess's internal representation where square 0 is a1 and square 63 is h8. When history is not provided, the current position is repeated for all 8 temporal steps to maintain consistent tensor dimensions.

Planes 96-97 encode repetition counters following FIDE rules for threefold repetition. Plane 96 marks squares where the position has occurred once before (1.0 everywhere if true, 0.0 otherwise). Plane 97 marks positions with two or more prior occurrences. These planes enable the network to recognize draws by repetition during training and evaluation.

Planes 98-101 encode castling rights as binary flags. Plane 98 represents white kingside castling, plane 99 white queenside, plane 100 black kingside, and plane 101 black queenside. Each plane is filled entirely with 1.0 if the right is available, 0.0 if lost. This representation allows the network to learn that castling availability affects position evaluation and move selection.

Plane 102 encodes the side to move, filled with 1.0 if white to move and 0.0 if black. This ensures the network can distinguish between positions that are identical except for whose turn it is. Plane 103 encodes the move count (fullmove number from the board's FEN representation) normalized by dividing by 100.0, providing temporal context about game phase.

Planes 104-118 encode the halfmove clock (50-move rule counter) using a thermometer encoding across 15 planes. Plane 104 is 1.0 if the clock is ≥ 1, plane 105 if ≥ 2, continuing through plane 118 if ≥ 15. This encoding allows the network to recognize approaching draws by the 50-move rule and adjust strategy accordingly. The thermometer representation (rather than binary encoding) provides smoother gradients during backpropagation.

4.5.2 Move Encoder

The `MoveEncoder` class in `data/move_encoder.py` implements bidirectional conversion between chess moves and action indices in the range [0, 4671]. The encoding scheme follows AlphaZero's 8×8×73 representation where each of the 64 starting squares has 73 possible move types. The `encode()` method accepts a `chess.Move` object and returns an integer index, while `decode()` performs the inverse operation.

Queen-style moves occupy planes 0-55, encoding 8 directions × 7 distances. The directions follow array indexing convention: North (row decreases toward rank 1), NorthEast, East, SouthEast, South (row increases toward rank 8), SouthWest, West, NorthWest. For each direction, distances 1-7 encode moves of 1 to 7 squares. The plane index is computed as direction_index * 7 + (distance - 1). This scheme efficiently represents sliding piece moves (bishop, rook, queen) as well as king moves (distance 1 in any direction) and castling (king moves 2 squares east or west).

Knight moves occupy planes 56-63, encoding the 8 possible L-shaped movements: NNE (2 north, 1 east), ENE (2 east, 1 north), ESE (2 east, 1 south), SSE (2 south, 1 east), SSW (2 south, 1 west), WSW (2 west, 1 south), WNW (2 west, 1 north), NNW (2 north, 1 west). Each plane corresponds to one of these fixed offset patterns.

Underpromotion moves occupy planes 64-72, encoding pawn promotions to pieces other than queen (knight, bishop, rook) in 3 directions (left-diagonal, forward, right-diagonal). White and black pawns use different direction vectors due to asymmetric board representation. The plane index is computed as 64 + direction_index * 3 + (promotion_piece_index - 1), where promotion piece indices are 0 for knight, 1 for bishop, 2 for rook. Queen promotions are encoded as queen-style moves (plane 4 for white forward, capturing queen promotions as appropriate diagonal planes).

The final action index combines the from-square and move plane: index = from_square * 73 + move_plane. This yields indices in [0, 4671] covering all possible chess moves. During encoding, special moves are detected and mapped appropriately: castling as 2-square king moves, en passant as diagonal pawn captures, promotions through the underpromotion or queen-style planes. Invalid moves raise ValueError exceptions during encoding.

4.5.3 ECO Classification

The eco_classifier.py module implements playstyle classification based on Encyclopedia of Chess Openings (ECO) codes as described in Section 3.6. This deterministic classification maps each game's opening to either tactical or positional style, enabling data filtering for cluster-specific training. The module defines two large sets of ECO codes extracted from the opening template files described in the methodology.

Tactical codes (stored in TACTICAL_ECO_CODES) include 150+ codes covering aggressive and sharp openings. Major tactical families include Sicilian Defence (B20-B99 excluding some positional lines), King's Gambit (C30-C39), Italian Game sharp variations (C50-C54), Alekhine's Defence (B02-B05), Vienna Game (C25-C29), Scandinavian Defence (B01), and King's Indian Attack (E60-E99). These openings typically feature rapid piece activity, tactical complications, and imbalanced pawn structures.

Positional codes (stored in POSITIONAL_ECO_CODES) include 200+ codes covering strategic and solid openings. Major positional families include Queen's Gambit Declined (D30-D69), Slav Defence (D10-D19), London System (D00-D05), Caro-Kann Defence (B10-B19), Nimzo-Indian Defence (E20-E59), Queen's Indian Defence (E12-E19), Catalan Opening (E00-E09), English Opening (A10-A39), and Réti Opening (A04-A09). These openings emphasize pawn structure, piece coordination, and long-term planning.

The classify_opening() function accepts an ECO code string and returns a PlaystyleType enum (TACTICAL or POSITIONAL). Games with unclassified ECO codes (not present in either set) are filtered out during sample extraction to maintain cluster purity. This classification integrates with the GameFilter dataclass, allowing trainers to request games matching specific playstyles during the offset-based sampling process.

4.5.4 Sample Extractor

The SampleExtractor class in data/sample_extractor.py converts complete chess games into individual position-move-outcome training samples. This extraction process filters games by rating and playstyle, skips formulaic opening and simplified endgame positions, and maintains position history for temporal context. The extract_samples() method orchestrates the entire pipeline, returning a list of TrainingSample objects ready for encoding.

The TrainingSample dataclass encapsulates all information needed for supervised learning: board (the chess position), move_played (the move to predict), game_outcome (+1 for win, 0 for draw, -1 for loss from current player's perspective), move_number, eco_code, playstyle classification, and history (list of up to 7 previous board positions). The outcome is perspective-adjusted so the network always predicts from the current player's viewpoint.

The ExtractionConfig dataclass controls filtering behavior. The skip_opening_moves parameter (default 10) excludes the first N moves from each game, removing highly theoretical opening positions that may not reflect playstyle-specific patterns. The skip_endgame_moves parameter (default 5) excludes positions with fewer than N pieces, avoiding simplified endgames where tactics and strategy converge. The sample_rate parameter (default 1.0) enables subsampling by extracting every Nth position. The shuffle_games flag (default True) randomizes game order for training diversity.

Sample extraction proceeds in three phases. First, the GameLoader loads games from the PGN database with filters applied (rating range, playstyle, max games) and offset for non-overlapping node data. Second, the _extract_samples_from_game() method iterates through each game's moves, creating board copies at each position, extracting the move played, determining the game outcome from the current player's perspective, maintaining a sliding window of 7 previous positions for history, applying skip rules (opening moves, endgame positions), and packaging everything into TrainingSample objects. Third, all samples from all games are collected into a single list and returned.

Memory efficiency is critical when processing databases with millions of games. The extractor uses streaming through the GameLoader, processing one game at a time rather than loading all games into memory. Board copies are created only for positions that pass filtering rules, minimizing allocation overhead. The history window uses shallow copies where possible, sharing piece placement data across temporal steps.

4.5.5 PyTorch Dataset Classes

PyTorch Dataset classes bridge the gap between extracted samples and the training loop by providing batched tensor access through DataLoader integration. Two dataset implementations support different training modes: ChessDataset for supervised learning from complete games and PuzzleDataset for tactical puzzle training.

The ChessDataset class (implemented in client/trainer/trainer_supervised.py) wraps a list of TrainingSample objects along with BoardEncoder and MoveEncoder instances. The __getitem__() method accepts an integer index, retrieves the corresponding sample, encodes the board and history to a (119, 8, 8) tensor using BoardEncoder, encodes the move played to an integer action index using MoveEncoder, converts the game outcome to a tensor, and returns a tuple (board_tensor, move_index, outcome) suitable for training. The __len__() method returns the total sample count for iteration.

PyTorch's DataLoader wraps the dataset with batching, shuffling, and parallel loading. Typical configuration uses batch_size=64, shuffle=True, num_workers=4 for parallel data loading, and pin_memory=True for faster GPU transfer. The DataLoader collates individual samples into batched tensors: board tensors become (batch, 119, 8, 8), move indices become (batch,), and outcomes become (batch, 1). This batching enables efficient GPU computation during forward and backward passes.

The PuzzleDataset class (implemented in client/trainer/trainer_puzzle.py) handles tactical puzzles with multi-move solutions. Unlike complete games, puzzles contain sequences of forcing moves that must all be learned. The dataset extracts positions from each move in the solution sequence, treating odd indices (player moves) as training samples and even indices (opponent replies) as context. For a puzzle with solution [setup, move1, reply1, move2, reply2], the dataset creates two samples: position after setup targeting move1, and position after reply1 targeting move2. This ensures the network learns complete tactical patterns, not just initial forcing moves. The encoding process mirrors `ChessDataset`, using the same `BoardEncoder` and `MoveEncoder` interfaces for consistency.

4.6 Aggregation Implementation

The aggregation implementation realizes the three-tier architecture described in Section 3.4 that enables clustered federated learning with playstyle preservation. This section describes the base aggregation infrastructure, intra-cluster FedAvg implementation, and inter-cluster selective aggregation that maintains diversity across playstyles. The aggregation modules in `server/aggregation/` coordinate model updates at both cluster and global levels.

4.6.1 Base Aggregator Framework

The `BaseAggregator` class in `server/aggregation/base_aggregator.py` provides the abstract foundation for all aggregation strategies. This abstract base class defines the interface that both intra-cluster and inter-cluster aggregators must implement, ensuring consistent behavior across aggregation levels. The class supports both PyTorch and TensorFlow models through framework-agnostic serialization and provides comprehensive input validation, metrics collection, and error handling.

The `AggregationMetrics` dataclass captures statistics about each aggregation operation: `aggregation_time` measures execution latency, `participant_count` tracks the number of models aggregated, `total_samples` sums training samples across participants, `average_loss` computes the weighted average loss, and `model_diversity` quantifies parameter divergence between models. Additional metrics such as `convergence_metric` and custom values stored in `additional_metrics` support experiment tracking and analysis.

The base aggregator constructor accepts `framework` (either 'pytorch' or 'tensorflow') and `compression` (boolean flag enabling gzip compression). Initialization creates a `ModelSerializer` instance using the factory function `get_serializer()`, configures validation flags, and sets participant limits. The `min_participants` parameter (default 1) enforces minimum participation for valid aggregation, while `max_participants` (default 1000) prevents memory exhaustion from excessive participants.

Two abstract methods define the aggregator interface. The `aggregate()` method accepts a dictionary of model states, aggregation weights, and round number, performing the actual aggregation algorithm and returning the aggregated model with metrics. The `get_aggregation_weights()` method calculates how much each participant contributes to the aggregated model based on provided metrics like sample counts or loss values. Subclasses must implement both methods according to their specific aggregation strategy.

Input validation occurs through the `validate_inputs()` method, which checks that models and weights dictionaries have matching keys, validates weight types and non-negativity, ensures participant counts fall within configured limits, and verifies that total weight is positive. The `check_model_compatibility()` method verifies that all models share the same parameter structure by comparing keys and tensor shapes against a reference model, raising `ValueError` if incompatibilities are detected.

Model diversity calculation uses the `calculate_model_diversity()` method, which computes average pairwise parameter distance between all models. The `_calculate_parameter_distance()` helper iterates through matching parameters, computing absolute differences and averaging across all parameters. High diversity indicates participants have learned different representations, while low diversity suggests convergence or insufficient training data variation.

4.6.2 Intra-Cluster Aggregator

The `IntraClusterAggregator` class in `server/aggregation/intra_cluster_aggregator.py` implements Federated Averaging [13] within individual clusters. This aggregator operates on models from nodes sharing the same playstyle classification, combining their updates through weighted averaging to create a cluster-level model. The aggregation preserves playstyle characteristics because all participating nodes trained on games with similar strategic patterns.

The constructor extends `BaseAggregator` with additional parameters: `weighting_strategy` (default 'samples') determines weight calculation method, `experiment_tracker` enables checkpoint saving and metrics logging, and `metric_registry` supports custom metric computation through plugins. Valid weighting strategies include 'samples' (weight by training sample count), 'uniform' (equal weights), and 'loss' (weight by inverse validation loss, favoring better-performing nodes).

The `aggregate()` method implements the FedAvg algorithm through six steps. First, input validation checks model compatibility and weight validity using inherited base class methods. Second, weight normalization ensures weights sum to 1.0 through the `normalize_weights()` utility function. Third, parameter-wise weighted averaging iterates through all model parameters, computing weighted sums using the formula $w_{agg} = \sum_i \alpha_i \cdot w_i$ where α_i represents normalized weight for participant i and w_i denotes their parameter values. Fourth, metrics collection creates an `AggregationMetrics` instance with timing, participant count, diversity score, and additional cluster metadata. Fifth, checkpoint saving stores the aggregated model to the model repository if `experiment_tracker` is configured. Sixth, statistics updates increment total aggregation count and accumulated time for performance monitoring.

The `get_aggregation_weights()` method calculates participant contributions based on the configured strategy. For 'samples' weighting, weights equal each participant's training sample count, naturally giving more influence to nodes that trained on more data. For 'uniform' weighting, all participants receive equal weight regardless of sample count or performance. For 'loss' weighting, weights equal the inverse of validation loss (nodes with lower loss contribute more), computed as $w_i = 1/(loss_i + \epsilon)$ where $\epsilon = 0.01$ prevents division by zero. The method validates that required metrics exist in the input dictionary before computing weights.

Integration with the experiment tracker enables comprehensive logging. After successful aggregation, the `log_metrics()` method records aggregation time, participant count, diversity score, and custom metrics to the metrics store. Checkpoint saving via `save_checkpoint()` persists the aggregated model state along with metadata including round number, cluster ID, and performance metrics. This checkpoint serves as both a training artifact and a restore point for fault tolerance.

4.6.3 Inter-Cluster Aggregator

The `InterClusterAggregator` class in `server/aggregation/inter_cluster_aggregator.py` implements the selective layer aggregation mechanism described in Section 3.5: selective weight sharing across clusters while preserving playstyle diversity. Unlike traditional federated learning that creates a single global model, this aggregator maintains separate models per cluster, synchronizing only specified layers while keeping strategic decision-making layers cluster-specific.

The constructor accepts `shared_layer_patterns` and `cluster_specific_patterns` as lists of layer name patterns supporting wildcards. Default shared patterns include ["input_conv.*"] covering the input convolution that processes board representations. Default cluster-specific patterns include ["policy_head.*", "value_head.*"] protecting the move selection and position evaluation layers that encode playstyle preferences. The weighting_strategy parameter (default 'samples') determines how to weight cluster contributions during aggregation of shared layers.

Layer identification uses pattern matching with the _matches_pattern() method, which converts wildcard patterns to regular expressions. For example, "policy_head.*" matches "policy_head.conv.weight", "policy_head.bn.bias", and all other policy head parameters. The _identify_shared_layers() method scans all parameter names, testing each against shared patterns and collecting matches. Similarly, _identify_cluster_specific_layers() identifies parameters to preserve per-cluster. Validation ensures no overlap between shared and cluster-specific sets and warns about unclassified layers (which default to cluster-specific for safety).

The aggregate() method implements selective aggregation through nine steps. Step 1 validates inputs using inherited validation methods. Step 2 identifies shared versus cluster-specific layers by pattern matching against all parameter names from the first cluster model. Step 3 validates layer identification by checking for overlaps, ensuring complete coverage, and verifying at least some shared layers exist. Step 4 normalizes aggregation weights to sum to 1.0. Step 5 aggregates shared layers across clusters using FedAvg (weighted averaging), producing a single set of shared parameters. Step 6 updates each cluster model by replacing shared layers with aggregated versions while preserving cluster-specific layers exactly as they were. Step 7 collects aggregation metrics including timing, participant count, shared layer count, and diversity measures. Step 8 computes custom metrics via the metric registry if configured, passing context including original models, updated models, and round number. Step 9 saves updated cluster model checkpoints if experiment tracker is available.

Shared layer aggregation in _aggregate_shared_layers() handles multiple parameter types. PyTorch tensors aggregate through tensor arithmetic: initialization creates a zero tensor matching the reference shape, then weighted sum accumulates $\sum_c \alpha_c \cdot \theta_c$ where c indexes clusters, α_c denotes normalized weights, and θ_c represents the layer parameters. For list-based representations (TensorFlow or manually serialized models), 1D parameters aggregate element-wise through list comprehension, while 2D parameters require nested loops over rows and columns. Scalar parameters simply sum weighted values.

Cluster model updates in _update_cluster_models() create new model state dictionaries for each cluster. The method starts with a shallow copy of the original cluster model, replaces all shared layer parameters with their aggregated versions, and preserves all cluster-specific and unclassified layers unchanged. This ensures tactical clusters maintain their aggressive policy heads while sharing generic board representation layers with positional clusters, achieving the diversity preservation goal.

4.7 Evaluation Implementation

The evaluation system implements the metrics framework for assessing playstyle preservation and playing strength. The implementation spans multiple modules in server/evaluation/ that orchestrate game play, analyze positions, classify moves, and compute model divergence.

4.7.1 Model Evaluator

The ModelEvaluator class in server/evaluation/model_evaluator.py orchestrates cluster evaluation by managing game play against Stockfish opponents and collecting metrics.

The evaluator:

1. Loads PyTorch models from the repository and sets them to evaluation mode
2. Uses `python-chess` for board state management and legal move generation
3. Configures Stockfish strength levels: for targets below 1320 ELO, sets `Skill Level` using $\max(0, \min(20, \lfloor (R_{\text{target}} - 800)/100 \rfloor))$ and limits search depth; for higher ratings, enables `UCI_LimitStrength` and sets `UCI_Elo` directly
4. Aggregates results in `ClusterEvaluationMetrics` dataclass containing outcome statistics, playstyle metrics, and ELO estimates

Integration with playstyle analysis occurs through `GameAnalyzer` and `PlaystyleMetricsCalculator`, which process completed games to extract per-position metrics.

4.7.2 Playstyle Metrics Calculator

The `PlaystyleMetricsCalculator` in `server/evaluation/playstyle_metrics.py` implements the tactical score computation from Section 5.4. The `GameAnalyzer` parses PGN strings, replays move sequences, and extracts features at each position using `python-chess` methods:

- `calculate_attacked_material()`: Uses `board.is_attacked_by()` to identify attacked pieces, sums values from `PIECE_VALUES` dictionary
- `calculate_legal_moves()`: Calls `len(list(board.legal_moves))` and records by game phase
- `calculate_center_control()`: Counts attackers for d4, d5, e4, e5 using `board.attackers()`
- `calculate_pawn_structure()`: Detects isolated pawns, doubled pawns, and computes average rank
- `calculate_tactical_score()`: Combines normalized components through weighted averaging

4.7.3 Move Type Analyzer

The `MoveTypeAnalyzer` in `server/evaluation/move_type_analyzer.py` categorizes moves using `python-chess` predicates: `board.is_capture()`, `board.gives_check()`, `board.is_castling()`, plus custom logic for pawn advances, piece development, and quiet moves. Per-phase tracking maintains separate counters for opening (plies 1-12), middlegame (13-40), and endgame (41+). The `ClusterMoveTypeMetrics` dataclass aggregates counts, percentages, and per-game averages.

4.7.4 Divergence Calculator

The `ModelDivergence` class in `server/evaluation/model_analyzer.py` computes parameter-level divergence between cluster models. For each layer group:

1. Loads weight tensors from both cluster models using PyTorch state dictionaries
2. Flattens multi-dimensional tensors to 1D vectors
3. Computes L2 distance and normalizes by parameter count: $\text{divergence} = \|W_A - W_B\|_2 / \sqrt{|\text{params}|}$
4. Aggregates into group-level scores (input block, early/middle/late residual, policy head, value head)

4.7.5 Weight Statistics Tracker

The WeightStatistics class in server/evaluation/weight_statistics.py monitors training health by computing per-layer statistics: mean, standard deviation, min/max values, sparsity (fraction of near-zero weights), and dead neuron ratio. Parameter change magnitude between rounds indicates learning dynamics. The tracker integrates with the experiment logging system to record statistics at regular intervals, enabling longitudinal analysis of weight evolution and detection of training issues.

4.7.6 Integration and Orchestration

The ExperimentTracker coordinates evaluation scheduling during training. At configurable intervals (default every 10 rounds), it triggers evaluation by calling ModelEvaluator.evaluate_cluster() for each cluster, collecting metrics, computing inter-cluster divergence, and persisting results to JSON files in the metrics store.

4.8 Storage System

The storage system implements persistent tracking of experiments, metrics, and model checkpoints throughout the training lifecycle. The implementation provides a modular architecture with abstract base classes defining storage interfaces and concrete implementations for file-based persistence. This design enables experiment reproducibility, longitudinal analysis of training dynamics, and easy integration with visualization tools.

4.8.1 Experiment Tracker

The FileExperimentTracker class in server/storage/experiment_tracker.py serves as the central coordinator for experiment lifecycle management, providing a unified interface for metrics logging, model checkpoint storage, and experiment metadata tracking. This class orchestrates the MetricsStore and ModelRepository components to ensure that all experiment artifacts are consistently organized and accessible.

Experiment initialization begins when the server calls start_run(), providing a configuration dictionary and optional run description. The tracker generates a unique run identifier using the format run_YYYYMMDD_HHMMSS_uuid, where the timestamp ensures chronological ordering and the UUID suffix prevents collisions. The tracker creates a run metadata structure containing the run ID, configuration snapshot, start time, description, and status field (initially set to "running"). This metadata is serialized to JSON and stored in .metadata/{run_id}.json, creating an audit trail of all experimental runs.

Coordination between storage components occurs through dependency injection. The experiment tracker accepts MetricsStore and ModelRepository instances during construction, allowing different storage backends to be swapped without modifying tracker logic. When logging metrics, the tracker forwards metric events to the metrics store using metrics_store.log_metric(), automatically enriching events with run ID and timestamp. When saving model checkpoints, the tracker delegates to model_repository.save_checkpoint(), passing along the run ID and cluster identifiers to ensure proper organization.

Configuration snapshot storage preserves the exact experimental setup for reproducibility. The complete configuration dictionary (including server settings, cluster topology, aggregation policies, and evaluation parameters) is embedded in the run metadata JSON file. This snapshot enables future reproduction of experiments by loading the saved configuration and verifying that code changes have not altered behavior. The tracker also records git commit hashes (if available) to link experiment results with specific code versions.

Experiment completion is managed through `end_run()`, which updates the run metadata with end time, final status (completed, failed, or interrupted), and optional summary results. The tracker computes experiment duration, counts total training rounds completed, and aggregates final metrics such as achieved ELO ratings and playstyle divergence. This summary is appended to the metadata file, providing quick access to experiment outcomes without requiring full metric file parsing.

4.8.2 Metrics Store

The `FileMetricsStore` class in `server/storage/file_metrics_store.py` implements append-only logging of training and evaluation metrics using the JSONL (JSON Lines) format. This storage strategy balances write performance (append-only files are fast), query flexibility (each line is valid JSON), and compression efficiency (gzip compresses JSONL well due to repeated field names).

Storage organization follows a hierarchical structure rooted at `storage/metrics/{run_id}/`. The primary storage file is `events.jsonl.gz`, containing all metric events in chronological order. Each event is a JSON object with fields including `timestamp`, `round_num`, `entity_type` (node, cluster, or global), `entity_id`, `metric_name`, and `value`. The append-only nature ensures thread safety without locks (file system append operations are atomic) and provides a complete temporal history suitable for time-series analysis.

Optional entity-organized views improve query performance for entity-specific metrics. When `organize_by_entity` is enabled, the store creates per-entity JSONL files under `by_entity/{entity_type}/{entity_id}.jsonl.gz`. For example, tactical cluster metrics are stored in `by_entity/cluster/cluster_tactical.jsonl.gz`. This organization enables fast retrieval of all metrics for a specific cluster without scanning the global events file, at the cost of duplicate storage (events are written to both global and entity-specific files).

Compression reduces storage requirements by approximately 5-7× through gzip compression at level 6. The metric store writes events to in-memory buffers, periodically flushing to disk via `gzip.open()`. JSON field names (which repeat for every event) compress exceptionally well, and numeric metric values benefit from dictionary-based compression. The compression level 6 strikes a balance between compression ratio and CPU overhead, avoiding the diminishing returns of higher compression levels.

Automatic indexing supports efficient queries over large metric datasets. The store maintains an index file `index.json` containing metadata about available metrics: discovered metric names, entity types and IDs, round number ranges, and file offsets for the first occurrence of each metric. When queries arrive, the store consults the index to determine which file regions to scan, avoiding full file reads. The index updates incrementally as new metrics are logged, using a lightweight scanning process that reads only recently appended data.

4.8.3 Model Repository

The `LocalModelRepository` class in `server/storage/local_model_repository.py` manages persistent storage of PyTorch model checkpoints with versioning, integrity verification, and automatic cleanup. The repository organizes checkpoints by run ID and cluster ID, enabling independent version tracking for each cluster's model evolution.

Checkpoint storage uses PyTorch's native serialization via `torch.save(state_dict, path)`, which employs Python's pickle protocol to serialize tensors efficiently. Each checkpoint is saved to `storage/models/{run_id}/{cluster_id}/round_{num:04d}.pt`, where the zero-padded round number ensures lexicographic sorting matches chronological order. Accompanying metadata files `round_{num:04d}_metadata.json` record checkpoint provenance: timestamp, training loss, validation metrics, optimizer state (if saved), and data generation information.

Checksum verification ensures checkpoint integrity during storage and retrieval. When `compute_checksums` is enabled, the repository computes SHA256 hashes of checkpoint files immediately after writing and stores the hashes in `checksums.json`. During checkpoint loading, the repository recomputes the hash and compares against the stored value, detecting silent corruption from disk errors or incomplete writes. If a checksum mismatch occurs, the repository raises an error and optionally falls back to the previous checkpoint.

Version tracking through symbolic links provides convenient access to important checkpoints. The repository maintains `latest.pt` as a symlink to the most recent checkpoint, enabling quick model loading without round number specification. When `keep_best` is enabled, the repository also maintains `best.pt` pointing to the checkpoint with the lowest validation loss (or highest ELO rating, depending on configuration). These symlinks simplify checkpoint selection for evaluation and deployment.

Automatic cleanup prevents unbounded storage growth during long training runs. When `keep_last_n` is configured, the repository deletes checkpoints older than the last N rounds after each save, retaining only recent history. The cleanup logic protects the "best" checkpoint from deletion (if `keep_best` is true), ensuring that optimal models are preserved regardless of subsequent performance degradation. Cleanup occurs asynchronously to avoid blocking checkpoint saves.

4.8.4 Plugin System

The `MetricRegistry` class in `server/storage/plugins/metric_registry.py` implements an extensible plugin architecture for custom metric computation. This registry pattern allows new metrics to be added without modifying core evaluation code, supporting experiment-specific analyses and integration with external tools.

Plugin registration follows a simple interface: plugins implement a `compute()` method accepting context (cluster models, evaluation results, round number) and returning a dictionary of computed metrics. The registry maintains a dictionary mapping plugin names to plugin instances, populated during initialization through explicit registration calls. For example, `registry.register("playstyle", PlaystyleMetricPlugin())` makes the playstyle plugin available for execution.

Available plugins provide specialized metrics beyond the core evaluation suite. The `PlaystyleMetricPlugin` in `playstyle_metric_plugin.py` integrates with the playstyle metrics calculator to compute tactical scores, move type distributions, and opening diversity for evaluation games. The `DiversityMetrics` plugin in `diversity_metrics.py` quantifies cluster behavioral divergence through statistical tests on playstyle distributions. The `SystemMetrics` plugin in `system_metrics.py` tracks computational resource usage including CPU time, memory consumption, and GPU utilization per training round.

Plugin execution occurs at configurable intervals during training. After completing an evaluation round, the server iterates through all registered plugins, calling their `compute()` methods with current experiment state. Plugins return metrics as dictionaries with string keys and numeric values, which the registry merges into the global metrics stream. This design enables plugins to access any experiment data (models, metrics history, configuration) while maintaining loose coupling through the standardized interface.

Extensibility for custom metrics is achieved through plugin subclassing. Users can create new plugins by inheriting from `MetricPlugin` base class and implementing the `compute()` method. The registry automatically discovers and registers plugins placed in the `server/storage/plugins/` directory, or plugins can be registered programmatically via the API. This extensibility enables domain-specific metrics (e.g., chess-specific tactical motif detection), integration with external evaluation frameworks, and custom analysis pipelines without forking the codebase.

4.9 Model Serialization

The model serialization system enables efficient transmission of neural network parameters between clients and server over WebSocket connections. The implementation provides framework-agnostic abstractions supporting both PyTorch and TensorFlow models, with configurable compression and encoding options optimized for network efficiency and JSON message compatibility.

4.9.1 Serialization Format

The `PyTorchSerializer` class in `common/model_serialization.py` implements a multi-stage serialization pipeline that transforms PyTorch state dictionaries into compact, transmittable representations. The serialization process follows four sequential stages: extraction, pickling, compression, and encoding.

Extraction begins by obtaining the model's state dictionary via `model.state_dict()`, which returns an ordered dictionary mapping parameter names (strings) to parameter tensors. The state dictionary includes all learnable parameters (convolutional weights, batch normalization parameters, fully connected weights) as well as persistent buffers (batch normalization running statistics). For the AlphaZeroNet architecture with 19 residual blocks, the state dictionary contains approximately 11 million parameters organized across roughly 160 distinct tensors.

Pickling serializes the state dictionary to binary format using Python's pickle protocol version 4. The serializer creates an in-memory bytes buffer via `io.BytesIO()`, then invokes `pickle.dump(state_dict, buffer, protocol=4)` to write the serialized representation. Protocol 4 (introduced in Python 3.4) provides efficient serialization of large numeric arrays, storing tensor data in contiguous binary format rather than per-element encoding. The resulting bytes object typically occupies 40-45 MB for a full AlphaZeroNet model.

Compression reduces serialized size through gzip compression at level 6. The serializer applies `gzip.compress(pickled_data, compresslevel=6)` to the pickled bytes, achieving typical compression ratios of 5-7× for neural network weights. The high compression effectiveness stems from regularities in trained weights: many parameters cluster near zero due to weight decay, batch normalization parameters are small, and spatial locality in convolutional filters creates compressible patterns. Level 6 compression balances CPU overhead against size reduction, requiring approximately 0.5-1.0 seconds for compression on a typical CPU.

Encoding converts compressed binary data to JSON-compatible format for WebSocket transmission. When `encoding='base64'` is configured (the default), the serializer applies `base64.b64encode(compressed_data).decode('ascii')` to produce an ASCII string suitable for embedding in JSON messages. Base64 encoding increases size by approximately 33% (4 characters per 3 bytes), but this overhead is acceptable given the preceding compression. The final serialized representation typically occupies 8-10 MB for a full AlphaZeroNet model, enabling transmission over typical network connections in 1-5 seconds depending on bandwidth.

Deserialization reverses the pipeline to reconstruct the state dictionary. The deserializer accepts a base64 string (or raw bytes if binary encoding was used), decodes via `base64.b64decode()`, decompresses via `gzip.decompress()`, and unpickles via `pickle.loads()` to recover the original state dictionary. The client or server then loads the state dictionary into a model instance using `model.load_state_dict(state_dict)`, updating all parameters to the transmitted values.

Framework abstraction is achieved through the `ModelSerializer` abstract base class, which defines the `serialize()` and `deserialize()` interface that all framework-specific serializers must implement. This abstraction enables the WebSocket protocol layer to remain framework-agnostic: the same message handling code works with PyTorch, TensorFlow, or future frameworks by simply swapping the serializer instance. The `get_serializer(framework='pytorch', compression=True)` factory function instantiates the appropriate serializer based on a string parameter, supporting runtime framework selection.

4.9.2 Parameter Differencing

Parameter differencing optimizes bandwidth consumption by transmitting weight updates rather than full model parameters. Instead of sending the complete updated model θ_{new}, clients compute and transmit the parameter difference $\Delta\theta = \theta_{\text{new}} - \theta_{\text{old}}$, where θ_{old} is the model received from the server at the beginning of the round. The server reconstructs the updated model by adding the difference to its stored version: $\theta_{\text{new}} = \theta_{\text{old}} + \Delta\theta$.

Bandwidth savings are most significant in early training rounds when parameter updates are small relative to total parameter magnitude. In a typical training round, clients perform 50-200 gradient descent updates with learning rate 10^{-4} to 10^{-3}, resulting in parameter changes on the order of 1-5% of total parameter magnitude. When serialized, these small differences compress more effectively than full parameters because most difference values are near zero, creating highly compressible data. Empirical measurements show that serialized differences occupy 20-40% the size of serialized full parameters in early training, providing 2.5-5× bandwidth reduction.

Implementation occurs in the client communication layer's model upload logic. Before serializing the updated model, the client subtracts the initial model parameter-wise: for each layer name in the state dictionary, the client computes `delta[layer] = updated_state[layer] - initial_state[layer]` using PyTorch tensor subtraction. The resulting delta dictionary (with identical keys but smaller magnitude tensors) is then serialized and transmitted. The client includes a `delta_encoding: true` flag in the message metadata to inform the server that reconstruction is required.

Server reconstruction applies the reverse operation upon receiving a delta-encoded model update. The server retrieves the original model sent to this client at round start from a cache keyed by `(round_num, node_id)`, deserializes the received delta, and adds each delta tensor to the corresponding original tensor: `reconstructed[layer] = original[layer] + delta[layer]`. The reconstructed state dictionary is then used for aggregation as if a full model had been received. The server discards the cached original models after aggregation completes to free memory.

Bandwidth savings diminish as training progresses and models converge. In late training rounds with small learning rates, parameter changes become very small, but they no longer compress significantly better than the full parameters because the changes are distributed across all parameters rather than concentrated in specific layers. Additionally, the base64 encoding overhead (33%) applies equally to differences and full parameters. Consequently, the framework makes delta encoding optional and configurable per experiment, enabling users to disable it for later training phases where benefits are marginal.

4.10 Configuration System

The configuration system provides declarative specification of all experimental parameters through YAML files, enabling reproducible experiments and systematic parameter sweeps without code modification. The implementation uses PyYAML for file parsing, dataclasses for type-safe configuration structures, and validation logic to catch errors before training begins.

4.10.1 Configuration Files

Configuration files are organized hierarchically to separate concerns and enable configuration reuse across experiments. The `config/` directory contains three categories of configuration files: server configuration, cluster topology, and per-node training configurations.

Server configuration files (e.g., `server_config_P2.yaml`) specify federated learning orchestration parameters. The `server_config` section defines network settings including host address and port for the WebSocket server. The `orchestrator_config` section configures aggregation behavior: `aggregation_threshold` sets the minimum fraction of cluster nodes that must participate for aggregation to proceed (typically 0.8 for 80% participation), `timeout_seconds` limits waiting time for stragglers, and the shared/cluster-specific layer patterns define the selective aggregation policy. For example, configuration P2 specifies `shared_layer_patterns: ["res_blocks.6.*", "res_blocks.7.*", ..., "res_blocks.12.*"]` to share only middle residual blocks while keeping early blocks, late blocks, and heads cluster-specific.

Evaluation configuration resides in the `evaluation_config` section, controlling when and how cluster models are evaluated. The `interval_rounds` parameter determines evaluation frequency (typically every 10 rounds), `games_per_elo_level` sets the number of games played against each Stockfish opponent, and `stockfish_elo_levels` lists the target opponent strengths for ELO estimation. Additional parameters include `skip_check_positions` (boolean controlling whether positions in check are excluded from playstyle analysis), `enable_delta_analysis` (boolean enabling expensive Stockfish-based delta metric computation), `delta_sampling_rate` (integer controlling position sampling density), and `stockfish_depth` (search depth for delta analysis, typically 12-15 plies).

Cluster topology configuration (`cluster_topology.yaml`) defines the cluster structure and node assignments. The top-level `clusters` list contains one entry per cluster, each specifying `id` (cluster identifier used throughout the system), `playstyle` (tactical or positional, controlling data filtering), `description` (human-readable cluster purpose), `node_count` (number of nodes in the cluster), `node_prefix` (prefix for auto-generated node IDs), and `games_per_round` (target games per node per training round). The topology file also includes an optional `resume_training` section with `enabled` flag and `starting_round` parameter, enabling checkpoint restoration and data offset for continued training.

Per-node configuration files specify training data sources and learning parameters for individual nodes. These files reside in subdirectories like `config/nodes/puzzle_configs/` and follow the naming pattern `node_{id}.yaml`. Each node configuration includes `data_source` (path to training data files, typically PGN or puzzle JSON), `data_type` (supervised, self-play, or puzzle), `batch_size`, `learning_rate`, `optimizer` (Adam, SGD, or AdamW), and data augmentation settings such as board flip probability and color swap probability.

YAML loading and validation occurs during server initialization. The configuration loader uses PyYAML's `safe_load()` function to parse YAML files into Python dictionaries, avoiding arbitrary code execution vulnerabilities from `load()`. After parsing, the loader performs schema validation: checking that required fields exist, verifying that numeric parameters fall within valid ranges (e.g., learning rate between 10^{-5} and 10^{-1}), ensuring that referenced files and directories exist, and validating that layer patterns are well-formed regular expressions. Validation errors are collected and reported together, enabling users to fix multiple issues simultaneously rather than encountering them one at a time.

4.10.2 Configuration Dataclasses

Configuration dataclasses provide type-safe, validated representations of configuration data with IDE autocompletion support and runtime type checking. The implementation uses Python's `@dataclass` decorator with type annotations to define configuration structures that mirror the YAML schema.

The RoundConfig dataclass encapsulates parameters controlling a single training round. Fields include round_num: int (current round number), games_per_node: int (target games for each node to generate), aggregation_threshold: float (minimum participation rate), timeout: int (maximum wait time in seconds), and save_checkpoint: bool (whether to persist models after this round). The dataclass includes a validate() method that checks invariants such as $0 < \text{aggregation\_threshold} \leq 1$ and games_per_node > 0, raising ValueError with descriptive messages when validation fails.

The EvaluationConfig dataclass specifies evaluation parameters as described in the configuration files section. Fields match the YAML structure: enabled: bool, interval_rounds: int, games_per_elo_level: int, stockfish_elo_levels: List[int], time_per_move: float, skip_check_positions: bool, enable_delta_analysis: bool, delta_sampling_rate: int, and stockfish_depth: int. The dataclass provides computed properties like total_games_per_cluster() which returns $\text{games\_per\_elo\_level} \times \text{len(stockfish\_elo\_levels)}$, simplifying downstream logic.

The TrainingConfig dataclass represents node-level training parameters. Core fields include batch_size: int, learning_rate: float, optimizer: str, weight_decay: float, epochs_per_round: int, and data_source: Path. Additional fields control data processing: flip_board: bool, swap_colors: bool, max_samples_per_round: Optional[int], and shuffle_data: bool. The dataclass uses field(default=...) to specify default values, enabling partial configuration where only non-default parameters need specification in YAML files.

Type safety is enforced through type annotations and runtime checking. When constructing dataclasses from YAML dictionaries, the configuration loader calls TrainingConfig(**yaml_dict), which triggers Python's runtime type checking if strict mode is enabled. For critical parameters, the dataclasses include explicit type validation in __post_init__() methods that convert string paths to Path objects, parse enum strings to enum values, and validate that lists contain homogeneous types. This validation catches configuration errors at startup rather than during training when they would cause cryptic failures.

Default values and overrides enable hierarchical configuration composition. Base configuration files define defaults applicable to all experiments, while experiment-specific files override selected parameters. The configuration loader implements a merge strategy that deep-updates nested dictionaries, allowing a derived configuration to override orchestrator_config.timeout_seconds without needing to repeat all other orchestrator_config fields. This composition reduces duplication across related experiments and ensures consistency for unmodified parameters.

Chapter 5

Experimental Setup

5.1 Experimental Design

This chapter describes the experimental methodology used to validate the playstyle-aware federated learning framework. The experiments are organized into three main phases: baseline comparisons, partial layer sharing configurations, and comprehensive performance evaluation. Each phase is designed to test specific hypotheses about how selective aggregation affects model specialization, playing strength, and behavioral differentiation.

5.1.1 Research Hypotheses

The evaluation framework tests seven research hypotheses that span model performance, behavioral emergence, and system stability. These hypotheses are organized into three categories: performance-related, behavior-related, and system-related.

Performance Hypotheses

H1: Clustered FL outperforms centralized training. The first hypothesis tests whether allowing clusters to specialize through selective layer sharing produces stronger models than forcing all nodes to converge to a single shared model. We expect that partial sharing (experiments P1-P4) will achieve higher ELO ratings than full sharing (baseline B1) because clusters can develop specialized strategies while still benefiting from shared knowledge in common layers.

H2: Selective aggregation improves cluster models. This hypothesis examines whether the layer sharing mechanism allows clusters to develop distinct specializations. We measure this through cluster divergence metrics, expecting to see high divergence in cluster-specific layers (like policy heads) and low divergence in shared layers (like early residual blocks). The divergence pattern should fall between the two extremes: B1 (full sharing, near-zero divergence) and B2 (no sharing, maximum divergence).

H3: Cross-cluster learning enables knowledge transfer. While H2 validates specialization, H3 tests whether shared layers enable useful knowledge transfer between clusters. We expect partial sharing experiments (P1-P4) to outperform the no-sharing baseline (B2) because shared layers allow clusters to learn from each other's experience, even while maintaining specialized heads.

Behavioral Hypotheses

H4: Playstyle clusters emerge naturally. This hypothesis tests whether training on different puzzle sets causes measurable differences in playing style. The tactical cluster should achieve significantly higher tactical scores (measuring aggressive play, captures, and checks) compared to the positional cluster. We use independent t-tests with a stringent threshold ($p < 0.001$) and expect a large effect size (Cohen's $d > 0.8$).

H5: Different clusters develop distinct strategies. Beyond overall tactical scores, H5 examines whether clusters show different strategic preferences in their move choices. The tactical cluster should play more aggressive moves (captures and checks), while the positional cluster should favor quiet moves, pawn advances, and positional maneuvering. Move type distribution analysis provides the primary evidence for this hypothesis.

H6: Behavioral differences are measurable. This hypothesis validates that our evaluation metrics are sensitive enough to detect meaningful behavioral differences. We require effect sizes greater than 0.5 (medium effect) for move type comparisons, ensuring that observed differences are not just statistically significant but also practically meaningful.

System Hypotheses

H7: Clusters maintain stability over training. Training stability is critical for practical deployment. This hypothesis tests whether cluster divergence stabilizes over time rather than oscillating or collapsing back to convergence. We use plateau detection algorithms to identify when metrics stabilize and verify that models don't reconverge in later training rounds.

5.1.2 Experiment Structure

The experimental evaluation consists of three sequential phases, with each phase designed to answer specific research questions. The phases build on each other: baselines establish performance boundaries, partial sharing experiments explore the design space, and performance evaluation validates behavioral hypotheses.

Phase 1: Baseline Experiments

Phase 1 establishes the performance boundaries by testing two extreme configurations. Baseline B1 (Full Sharing) aggregates all model layers across clusters, effectively implementing standard federated learning without specialization. This represents the upper bound on knowledge sharing but the lower bound on specialization. Baseline B2 (No Sharing) runs completely independent training for each cluster with no aggregation, representing the upper bound on specialization but no knowledge transfer.

These baselines serve multiple purposes. First, they validate that the framework can reproduce standard federated learning (B1) and independent training (B2) as special cases. Second, they provide comparison points for the partial sharing experiments. Third, they help isolate the effect of selective aggregation by controlling for other variables like model architecture, training data, and hyperparameters.

Both baselines run for 200 rounds with identical training configurations. Each round involves local training on 400 puzzles per node (3,200 puzzles total across 8 nodes), followed by model aggregation according to the experiment's sharing policy. Metrics are collected every round for playstyle evaluation, weight statistics, and cluster divergence, with ELO estimation and move type analysis conducted every 10 rounds.

Phase 2: Partial Layer Sharing Experiments

Phase 2 explores four different layer sharing configurations to identify which layers should be shared versus specialized. The configurations are motivated by the AlphaZero architecture's [21] hierarchical structure (Section 3.3), where early layers learn low-level patterns, middle layers learn strategic concepts, and late layers (especially heads) make final decisions.

Experiment P1 (Share Early Layers Only) shares the input block and early residual blocks (0-5) while keeping middle layers, late layers, and heads cluster-specific. The hypothesis is that low-level feature extraction generalizes across playing styles, but strategic and tactical reasoning should specialize.

Experiment P2 (Share Middle Layers Only) keeps input and early layers cluster-specific, shares middle residual blocks (6-12), and keeps late layers cluster-specific. This tests whether mid-level strategic concepts can be shared while allowing specialization in both feature extraction and final decision-making.

Experiment P3 (Share Late Layers Only) shares late residual blocks (13-18) and both heads while keeping early and middle layers cluster-specific. This configuration is counter-intuitive, it allows specialization in low-level features but forces convergence in high-level decision-making.

Experiment P4 (Share All Except Heads) shares the entire backbone (input block and all 19 residual blocks) while keeping only the policy and value heads cluster-specific (Section 3.5). This represents minimal specialization, testing whether head-only specialization is sufficient for behavioral differentiation.

Each experiment runs for 200 rounds with identical training procedures and metric collection protocols. The goal is to compare performance, convergence speed, and behavioral differentiation across configurations to identify the optimal layer sharing strategy.

Phase 3: Performance Evaluation

Phase 3 conducts comprehensive offline analysis after all training experiments complete. Unlike Phases 1 and 2, which collect metrics during training, Phase 3 uses the final trained models for in-depth evaluation.

Evaluation E1 (Playstyle Analysis) generates 500 self-play games per cluster model to compute comprehensive playstyle statistics with tight confidence intervals. This validates H4 and H5 by comparing tactical scores and move type distributions between clusters with statistical significance tests.

Evaluation E2 (Move Type Analysis) generates 200 games per cluster and classifies every move by type (captures, checks, aggressive moves, pawn advances, quiet moves). The analysis computes cluster comparison statistics to quantify behavioral differences and validate H6.

Evaluation E3 (Generalization Test) prepares benchmark position sets spanning tactical puzzles (mate-in-N, forks, pins), positional puzzles (endgame techniques, pawn structure), and mixed complexity positions. Each model is evaluated on all position types, including cross-testing tactical models on positional puzzles and vice versa.

5.1.3 Cluster Configuration

The experiments use a fixed cluster configuration with two clusters and eight total nodes. This configuration balances experimental control with enough clients to simulate realistic federated learning dynamics.

Cluster Assignment

The tactical cluster consists of four nodes (IDs: tactical-1, tactical-2, tactical-3, tactical-4), each assigned to train on tactical puzzles. The tactical puzzle set includes themes like forks, pins, skewers, discovered attacks, double attacks, sacrifices, deflections, and other sharp tactical patterns. These puzzles emphasize concrete calculation and forcing moves.

The positional cluster consists of four nodes (IDs: positional-1, positional-2, positional-3, positional-4), each assigned to train on positional puzzles. The positional puzzle set includes endgame techniques (rook endgames, pawn endgames), positional advantages, pawn structure themes, space advantage, weak squares, and piece activity. These puzzles emphasize long-term planning and strategic understanding.

The 4-4 split ensures balanced representation of both playing styles and prevents one cluster from dominating aggregation due to having more participating nodes. The aggregation threshold is set to 0.8 (80%), meaning at least 4 out of 5 nodes in each cluster must participate for aggregation to occur (with some tolerance for node failures).

Training Data Distribution

Each node samples 400 puzzles per round from its cluster's puzzle set. The sampling is random with replacement, allowing the model to revisit puzzles across rounds but ensuring diversity within each round. Over 200 rounds, each node trains on 80,000 puzzle positions, providing sufficient data for convergence.

The puzzle sets are disjoint, no puzzle appears in both the tactical and positional sets. This ensures that behavioral differences emerge from the training distribution rather than shared examples. The Lichess puzzle database provides over 2 million tagged puzzles, making it straightforward to create large non-overlapping sets.

Data augmentation is applied during training through board flips (horizontal flip with move translation) and color swaps (playing from black's perspective). This effectively doubles the training data size and improves generalization by exposing models to symmetrically equivalent positions.

Aggregation Protocol

Aggregation occurs at the end of each round after all participating nodes complete local training. The aggregation protocol differs by experiment:

For B1 (Full Sharing), all layers are aggregated using FedAvg weighted by the number of training examples. Each cluster performs independent aggregation, but since all layers are shared, the two clusters converge toward identical models.

For B2 (No Sharing), no aggregation occurs, each cluster maintains its own model checkpoint without combining updates. This is equivalent to running two independent training runs that happen to share the same experimental infrastructure.

For P1-P4 (Partial Sharing), layer-specific aggregation applies FedAvg only to designated shared layers while preserving cluster-specific layers unchanged. For example, in P4, the input block and all 19 residual blocks are aggregated across nodes within each cluster and then synchronized between clusters, while policy and value heads remain cluster-specific.

The aggregation threshold (0.8) requires 80% of cluster nodes to participate before aggregation occurs. If fewer nodes participate in a given round, that round's aggregation is skipped and models retain their previous weights. This prevents a single node from dominating the cluster model and ensures robustness to occasional node failures.

5.2 Baseline Experiments

The baseline experiments establish the performance boundaries for the partial layer sharing experiments. Two extreme configurations are tested: B1 (Full Sharing), which aggregates all layers and represents standard federated learning, and B2 (No Sharing), which runs completely independent training for each cluster. These baselines serve as control conditions to isolate the effect of selective aggregation.

5.2.1 B1: Full Sharing Baseline

The first baseline, B1 (Full Sharing), implements standard federated learning without any specialization mechanism. All model layers, input block, all 19 residual blocks, and both policy and value heads, are aggregated across clusters using the FedAvg algorithm [13]. This configuration provides the maximum knowledge sharing between clusters but prevents any form of model specialization.

Configuration

The B1 experiment uses a shared layer configuration that includes every parameter in the model. The `shared_layer_patterns` list specifies all layer groups: `input_conv.*`, `input_bn.*`, `res_blocks.*`, `policy_head.*`, and `value_head.*`. Meanwhile, the `cluster_specific_patterns` list remains empty, indicating that no layers are kept cluster-specific.

During aggregation, both clusters (tactical and positional) collect updates from their respective nodes and compute weighted averages using FedAvg [13]. However, since all layers are designated as shared, the aggregation mechanism (Section 3.4) synchronizes these layers between clusters. This effectively forces both clusters to converge toward identical models, with any cluster-specific knowledge being averaged away during aggregation.

The training runs for 200 rounds, with each round consisting of local training on 400 puzzles per node followed by model aggregation. The aggregation threshold is set to 0.8, requiring at least 80% of cluster nodes (4 out of 4 nodes, with some tolerance) to participate before aggregation occurs. The timeout per round is 1,200 seconds (20 minutes), providing sufficient time for local training, model uploads, and aggregation.

Expected Behavior

The B1 baseline represents the standard federated learning approach without clustering benefits. We expect several characteristic behaviors:

Near-zero divergence. Since all layers are synchronized between clusters after each round, the cluster divergence metrics should remain close to zero throughout training. Any temporary divergence that emerges during local training will be eliminated by the aggregation step. This provides a lower bound on divergence, the minimum possible separation when no specialization is allowed.

Minimal playstyle separation. Despite training on different puzzle sets (tactical vs. positional), the two clusters should develop similar playing styles because their models are forced to converge. The tactical score differences between clusters will be minimal, as the shared policy head cannot maintain distinct strategic preferences. This serves as a negative control for hypothesis H4, demonstrating that playstyle separation requires some form of model specialization.

Fast convergence. Full knowledge sharing means each cluster benefits from the training data of all 8 nodes, effectively doubling the training data size compared to independent training. We expect relatively fast convergence and good sample efficiency, with ELO ratings improving steadily across both clusters.

Moderate performance. The B1 baseline should achieve reasonable playing strength since it benefits from the combined training data of both clusters. However, it may underperform compared to partial sharing experiments if forcing all nodes toward a single model creates a "jack of all trades, master of none" effect. The model must compromise between tactical and positional puzzle distributions rather than specializing in either.

Metrics Collection

The B1 experiment collects comprehensive metrics to characterize model behavior. Playstyle evaluation, weight statistics, and cluster divergence are recorded every round, providing fine-grained visibility into training dynamics. Move type distribution and ELO estimation are conducted every 10 rounds to balance evaluation thoroughness with computational cost.

The playstyle evaluation generates 100 self-play games per cluster per round and computes tactical scores for middlegame positions. We expect both clusters to show similar tactical score distributions, validating that full sharing prevents specialization.

The cluster divergence computation compares weight tensors between the tactical and positional cluster models using L2 distance normalized by parameter count. The divergence is broken down by layer group (input block, early residual, middle residual, late residual, policy head, value head), though all groups should show near-zero divergence in B1.

ELO estimation plays games against Stockfish at multiple strength levels (1000, 1200, 1400 ELO) to track model performance over training. This provides a strength baseline for comparison with the partial sharing experiments.

5.2.2 B2: No Sharing Baseline

The second baseline, B2 (No Sharing), represents the opposite extreme: complete independence between clusters. No layers are aggregated between clusters, meaning the tactical and positional clusters train entirely separately using only their respective node updates. This configuration provides maximum specialization potential but zero knowledge transfer between clusters.

Configuration

The B2 configuration reverses the pattern from B1. The `shared_layer_patterns` list is empty, indicating that no layers are synchronized between clusters. Instead, the `cluster_specific_patterns` list includes all layer groups: `input_conv.*`, `input_bn.*`, `res_blocks.*`, `policy_head.*`, and `value_head.*`.

This configuration allows each cluster to maintain its own independent model. While intra-cluster aggregation still occurs (nodes within each cluster combine their updates using FedAvg), there is no inter-cluster aggregation. The tactical cluster's model evolves based solely on tactical puzzles from its 4 nodes, while the positional cluster's model evolves based solely on positional puzzles from its 4 nodes.

The training schedule and hyperparameters match B1 exactly: 200 rounds, 400 puzzles per node per round, 0.8 aggregation threshold, and 1,200-second timeout. This ensures that any performance differences between B1 and B2 can be attributed to the sharing policy rather than confounding variables.

Expected Behavior

The B2 baseline tests whether cluster specialization is beneficial when taken to the extreme. We expect several contrasting behaviors compared to B1:

High divergence. With no cross-cluster aggregation, the tactical and positional models will evolve along completely different trajectories. Divergence should increase steadily during early training as each model adapts to its specific puzzle distribution. By the end of training, we expect divergence values significantly higher than B1 across all layer groups, providing an upper bound on model separation.

Maximum playstyle separation. The B2 configuration provides ideal conditions for behavioral differentiation. The tactical cluster should develop a highly tactical playing style with high tactical scores, frequent captures and checks, and aggressive move preferences. The positional cluster should develop a positional playing style with lower tactical scores, more quiet moves, and focus on long-term planning. This serves as a positive control for hypothesis H4, demonstrating that specialization is possible when models train independently.

Slower convergence. Unlike B1, each cluster only has access to its own 4 nodes' data (1,600 puzzles per round total, compared to 3,200 when sharing with the other cluster). This smaller effective dataset may slow convergence and require more rounds to reach comparable performance. The learning curves should show slower ELO improvement compared to configurations with knowledge sharing.

Potentially lower performance. While specialization allows each cluster to optimize for its specific puzzle distribution, the lack of knowledge transfer means each cluster cannot benefit from patterns learned by the other cluster. General chess knowledge (like piece values, board control, king safety) that applies to both tactical and positional puzzles must be learned independently by each cluster rather than shared. This may result in lower final ELO compared to partial sharing experiments that combine specialization with knowledge transfer.

Metrics Collection

The B2 experiment uses the same metrics collection protocol as B1, ensuring fair comparison. The key difference is in the interpretation of metrics rather than their collection.

Playstyle evaluation in B2 should reveal clear separation between clusters. We expect the tactical cluster's mean tactical score to be significantly higher than the positional cluster's mean tactical score, with non-overlapping distributions and large effect sizes.

Cluster divergence serves as a key indicator of specialization success. Unlike B1 where divergence should be near-zero, B2's divergence should grow over training and stabilize at high values. The divergence breakdown by layer group reveals which layers specialize most strongly, we expect all layers to show high divergence since no sharing occurs.

Move type distribution analysis should show distinct behavioral signatures. The tactical cluster should play more captures, checks, and aggressive moves, while the positional cluster should favor quiet moves, pawn advances, and positional maneuvering. These differences validate that training on different puzzle sets produces measurably different playing styles.

5.2.3 Baseline Comparison

The two baselines represent opposite ends of a spectrum: full sharing versus no sharing. Comparing their performance and behavior motivates the partial sharing experiments in Phase 2.

Divergence and Specialization

The most obvious difference between B1 and B2 is cluster divergence. B1 should maintain near-zero divergence throughout training because all layers are synchronized after each round. Any divergence that emerges during local training (due to different puzzle distributions) is eliminated by aggregation. In contrast, B2 should show steadily increasing divergence during early training as each cluster's model adapts to its specific data distribution, eventually plateauing once specialization is complete.

This divergence difference validates hypothesis H2 from a boundary conditions perspective: B1 shows that sharing all layers prevents specialization (divergence ≈ 0), while B2 shows that sharing no layers allows maximum specialization (high divergence). The partial sharing experiments will test whether intermediate divergence levels can be achieved by selectively sharing some layers while keeping others cluster-specific.

Performance and Knowledge Transfer

The performance comparison between B1 and B2 tests hypothesis H3 (cross-cluster learning enables knowledge transfer). If B1 significantly outperforms B2, it suggests that knowledge sharing is beneficial despite preventing specialization. Conversely, if B2 matches or exceeds B1's performance, it suggests that specialization benefits outweigh knowledge transfer benefits, motivating stronger specialization in the partial sharing experiments.

We expect the truth to lie somewhere in between: B1 may show faster initial convergence due to larger effective dataset size, but B2 may achieve higher final performance on cluster-specific tasks due to specialization. The partial sharing experiments aim to capture the best of both worlds, fast convergence through shared general knowledge and strong final performance through specialized cluster-specific layers.

Behavioral Differentiation

The behavioral metrics provide the clearest contrast between baselines. B1 should show minimal playstyle differences between clusters because the shared policy head cannot maintain distinct move preferences. Both clusters will converge toward a compromise playing style that balances tactical and positional considerations.

B2, on the other hand, should show maximum behavioral differentiation. With completely independent models, each cluster can fully optimize for its training distribution. The tactical cluster should exhibit highly tactical play, while the positional cluster should exhibit strongly positional play. The effect sizes for tactical score differences and move type distribution differences should be large in B2 but small in B1.

This comparison validates hypothesis H4 (playstyle clusters emerge naturally) by demonstrating that behavioral differentiation requires model specialization. It also provides guidance for the partial sharing experiments: if we want clusters to develop distinct playing styles, we must keep at least some layers (likely the policy head) cluster-specific rather than shared.

Design Space Motivation

The baseline comparison motivates the design space explored in Phase 2. B1 and B2 represent extreme points: sharing all layers or sharing no layers. The performance and behavioral characteristics of these extremes inform which intermediate configurations are worth testing.

If B1 converges faster but achieves lower final performance, it suggests that early layers (which learn general features) should be shared to accelerate learning, while late layers (which make final decisions) should remain cluster-specific for specialization. This motivates experiment P4 (share all except heads).

If B2 shows strong specialization but slow convergence, it suggests that some knowledge transfer is beneficial but full sharing is excessive. This motivates experiments P1-P3, which selectively share different layer groups to balance knowledge transfer and specialization.

The baselines also establish performance bounds for statistical testing. Hypothesis H1 requires that partial sharing outperforms B1 (full sharing), while hypothesis H3 requires that partial sharing outperforms B2 (no sharing). Together, these constraints define success: the optimal partial sharing configuration should beat both baselines by achieving stronger final performance than B1 while converging faster than B2.

5.3 Partial Layer Sharing Experiments

Phase 2 explores four different partial layer sharing configurations to identify the optimal balance between knowledge transfer and model specialization. Each experiment selectively shares different layer groups based on hypotheses about which layers encode general chess knowledge versus playstyle-specific patterns. The configurations span from sharing only early layers (P1) to sharing the entire backbone except heads (P4), systematically testing different points in the design space.

5.3.1 Layer Group Definitions

Before describing each experiment, we need to establish how the AlphaZero architecture [21] is divided into layer groups (Section 3.3). The model's 11 million parameters are organized hierarchically, with each group serving a distinct function in the decision-making pipeline.

The **input block** consists of a convolutional layer and batch normalization, totaling approximately 20,000 parameters. This layer performs initial feature extraction from the 119-plane board representation, converting the raw position encoding into a dense feature map. The input block learns to identify basic board features like piece locations, castling rights, and move history.

The **early residual blocks** (blocks 0-5) contain approximately 2.5 million parameters distributed across six residual blocks with skip connections. These layers learn low-level pattern recognition, including piece recognition, basic board geometry, and simple tactical patterns. Early residual blocks capture features that are largely universal across playing styles, all chess players need to recognize pieces and understand basic board topology.

The **middle residual blocks** (blocks 6-12) contain approximately 3.5 million parameters across seven residual blocks. These layers learn mid-level tactical patterns such as forks, pins, skewers, discovered attacks, and other common tactical motifs. The question of whether these patterns are playstyle-specific or universal motivates experiment P2.

The **late residual blocks** (blocks 13-18) also contain approximately 3.5 million parameters across six residual blocks. These layers learn high-level strategic planning, including pawn structure evaluation, positional advantages, long-term piece coordination, and overall position assessment. Late residual blocks are candidates for playstyle-specific representation.

The **policy head** contains approximately 1.5 million parameters across a convolutional layer and fully connected layer. The policy head outputs 4,672 logits representing move probabilities for all possible moves (64 source squares × 64 destination squares, plus special moves and promotions). This head directly encodes move selection preferences and is the primary candidate for playstyle-specific parameters.

The **value head** contains approximately 500,000 parameters across a convolutional layer and fully connected layers. The value head outputs a single scalar in the range [-1, 1] representing the estimated game outcome from the current position. While position evaluation might seem universal, different playing styles may evaluate positions differently, tactical players may value active pieces more, while positional players may prioritize pawn structure.

5.3.2 P1: Share Early Layers Only

The first partial sharing experiment, P1, tests the hypothesis that low-level feature extraction is universal across playing styles while higher-level reasoning should specialize. This configuration shares the input block and early residual blocks (blocks 0-5) between clusters while keeping all other layers cluster-specific.

Configuration

The P1 experiment designates the input block (`input_conv.*`, `input_bn.*`) and early residual blocks (`res_blocks.0.*` through `res_blocks.5.*`) as shared layers. These approximately 2.5 million parameters are aggregated across both tactical and positional clusters after each training round.

All remaining layers, middle residual blocks (6-12), late residual blocks (13-18), and both heads, are marked as cluster-specific. These approximately 8.5 million parameters remain separate for each cluster, allowing independent specialization based on the respective puzzle distributions.

The aggregation protocol works as follows: after local training, nodes within each cluster combine their updates using FedAvg to produce a cluster model. Then, for shared layers only, the tactical and positional cluster models synchronize their parameters using weighted averaging. The cluster-specific layers remain unchanged during this cross-cluster synchronization step.

Hypothesis and Expected Behavior

The P1 configuration is motivated by the observation that all chess positions require the same basic piece recognition and board understanding regardless of playing style. A tactical player and a positional player both need to recognize where pieces are located, what pieces attack which squares, and basic board geometry. These fundamental features should be learnable from both tactical and positional puzzles.

By sharing early layers, both clusters benefit from the combined training data when learning these universal features. This should accelerate convergence compared to B2 (no sharing) because each cluster effectively has access to 8 nodes' worth of data for early feature learning rather than just 4.

However, strategic reasoning and move selection preferences differ between playing styles. Tactical positions reward aggressive piece activity, forcing moves, and material exchanges. Positional positions reward pawn structure, piece coordination, and long-term planning. By keeping middle layers, late layers, and heads cluster-specific, each cluster can develop specialized representations optimized for its puzzle distribution.

We expect the following divergence pattern: near-zero divergence in input block and early residual blocks (these layers are synchronized), gradually increasing divergence in middle residual blocks (these layers begin to specialize), high divergence in late residual blocks (strategic reasoning differs), and very high divergence in both heads (move preferences and position evaluation strongly reflect playstyle).

Performance-wise, P1 should achieve good ELO ratings due to efficient learning of shared features combined with specialized strategic reasoning. Playstyle separation should be moderate to high, as cluster-specific middle layers, late layers, and heads can encode behavioral differences.

5.3.3 P2: Share Middle Layers Only

The second experiment, P2, takes an unusual approach by sharing only the middle residual blocks (6-12) while keeping everything else cluster-specific. This configuration serves as an exploratory experiment to understand which layers encode tactical knowledge.

Configuration

P2 designates only the middle residual blocks (`res_blocks.6.*` through `res_blocks.12.*`) as shared layers, comprising approximately 3.5 million parameters. The input block, early residual blocks (0-5), late residual blocks (13-18), and both heads are all cluster-specific.

This creates an unusual "sandwich" pattern where the edges of the network (input and output) are cluster-specific while the middle is shared. The aggregation synchronizes middle layer parameters between clusters but leaves all other layers independent.

Hypothesis and Expected Behavior

The P2 configuration tests whether mid-level tactical patterns are universal across playing styles. The underlying question is: do tactical and positional players recognize the same tactical patterns (forks, pins, skewers) even if they prioritize them differently?

One perspective argues that tactical pattern recognition is universal, both tactical and positional players need to identify when a fork is possible or when a pin exists. The difference lies not in pattern recognition but in how these patterns influence move selection. Under this view, sharing middle layers (where tactical patterns are encoded) could be beneficial.

An alternative perspective suggests that even pattern recognition differs between styles. Tactical players may develop more refined representations of forcing moves and material exchanges, while positional players may develop stronger representations of piece coordination and pawn structures. Under this view, sharing middle layers would hurt specialization.

The P2 experiment adjudicates between these perspectives. If middle layer sharing produces good ELO and maintains playstyle separation, it supports the universal tactical patterns hypothesis. If P2 underperforms compared to P1 or P4, it suggests that middle layers do encode playstyle-specific patterns.

We expect divergence to show the reverse pattern from P1: cluster-specific input and early layers will diverge, shared middle layers will maintain near-zero divergence, and cluster-specific late layers and heads will diverge. This unusual pattern helps isolate the role of each layer group.

5.3.4 P3: Share Late Layers Only

The third experiment, P3, serves as a control test of the main hypothesis by sharing late residual blocks and both heads while keeping early and middle layers cluster-specific. This configuration is expected to fail, thereby validating that heads encode playstyle.

Configuration

P3 designates late residual blocks (`res_blocks.13.*` through `res_blocks.18.*`) and both heads (`policy_head.*`, `value_head.*`) as shared layers. These approximately 5.5 million parameters are synchronized between clusters. The input block, early residual blocks (0-5), and middle residual blocks (6-12) are cluster-specific, totaling approximately 5.5 million parameters.

This configuration forces both clusters to use identical strategic reasoning and decision-making layers while allowing them to develop different low-level and mid-level representations. It represents the opposite of our main hypothesis about layer roles.

Hypothesis and Expected Behavior

The P3 experiment tests a counter-hypothesis: what if high-level strategic planning and decision-making are actually universal across playing styles, with differences arising only in how positions are interpreted at lower levels?

This view would suggest that all strong chess players use similar decision-making processes (e.g., similar move selection criteria, similar position evaluation principles), but tactical and positional players attend to different low-level features when feeding information into these decision processes. Under this model, sharing heads would be acceptable.

However, we expect this hypothesis to be wrong. Playing style seems fundamentally about *how decisions are made*, not just about what features are extracted. A tactical player selects moves based on forcing moves and material exchanges. A positional player selects moves based on long-term pawn structure and piece coordination. These are different decision-making strategies, not just different feature interpretations.

Therefore, we expect P3 to perform poorly in terms of playstyle separation. The shared policy head will be forced to compromise between tactical and positional move preferences, likely converging toward a balanced intermediate strategy that serves neither cluster particularly well. This is similar to the B1 baseline but with even less effective sharing (since early/middle features aren't shared).

We expect divergence to be zero in late residual blocks and both heads (these are synchronized), while input, early, and middle layers diverge. Behavioral metrics should show minimal playstyle differences, validating that heads must be cluster-specific to enable behavioral differentiation.

If P3 somehow performs well, it would challenge our understanding of how playstyle is encoded in neural networks and potentially motivate reconsideration of the layer sharing strategy. However, based on the architecture's design and multi-task learning principles, we strongly expect P3 to underperform P1, P4, and possibly even B2.

5.3.5 P4: Share All Except Heads

The fourth experiment, P4, represents the hypothesis-driven configuration where the entire backbone (input block and all 19 residual blocks) is shared while only the policy and value heads remain cluster-specific. This tests whether minimal specialization is sufficient for playstyle differentiation.

Configuration

P4 designates the input block and all residual blocks (`res_blocks.0.*` through `res_blocks.18.*`) as shared layers, comprising approximately 9.5 million parameters (about 86% of the model). Only the policy head and value head are cluster-specific, totaling approximately 2 million parameters (about 18% of the model).

This configuration creates a clean separation: a massive shared backbone that learns general chess understanding, and small specialized heads that encode cluster-specific decision-making. The aggregation synchronizes the entire feature extraction pipeline while keeping final decisions independent.

Hypothesis and Expected Behavior

The P4 experiment is motivated by multi-task learning principles and the AlphaZero architecture's design. In multi-task learning, a common strategy is to share a large feature extraction backbone across tasks while using task-specific output heads. This allows the backbone to learn general representations useful across tasks while heads specialize for each task's objectives.

Applying this principle to playstyle clustering, we hypothesize that most chess knowledge is universal: piece values, mobility, king safety, pawn structure principles, tactical patterns, and strategic concepts. Both tactical and positional players need to understand these concepts. The difference lies in how they *apply* this knowledge when selecting moves and evaluating positions.

The policy head encodes move selection preferences, which moves to prioritize given a position. Tactical players should have policy heads that assign high probability to forcing moves, captures, and checks. Positional players should have policy heads that favor quiet moves, pawn advances, and long-term improvements.

The value head encodes position evaluation criteria, how to score a position's quality. Tactical players may value material imbalances and active piece positions more highly, while positional players may value solid pawn structures and space advantages.

By keeping only these 2 million parameters cluster-specific, P4 tests whether playstyle can be encoded in a compact specialized layer. The advantage is maximum knowledge transfer through the shared 9.5M parameter backbone, potentially leading to faster convergence and stronger overall chess understanding. The risk is that 2M parameters may not be sufficient to capture all playstyle-specific nuances, potentially limiting behavioral differentiation.

We expect the following divergence pattern: near-zero divergence in input block and all residual blocks (the entire backbone is synchronized), and very high divergence in both heads (these are the only cluster-specific parameters). This creates a stark separation that makes interpretation straightforward.

Performance-wise, P4 is a strong candidate for the optimal configuration. It maximizes knowledge transfer (more sharing than P1), focuses specialization where it matters most (decision-making), and maintains parameter efficiency (only 2M cluster-specific parameters per cluster). If the hypothesis is correct, P4 should achieve the best combination of high ELO (from shared general knowledge) and strong playstyle separation (from specialized heads).

5.3.6 Partial Sharing Design Space

The four partial sharing experiments systematically explore different points in the design space between full sharing (B1) and no sharing (B2). Comparing their configurations, predictions, and eventual results reveals which layers encode playstyle and what sharing strategy optimizes the knowledge transfer versus specialization tradeoff.

Configuration Summary

P1 (Share Early) shares approximately 23% of model parameters (input + early residual), keeping 77% cluster-specific. This configuration bets that low-level features are universal but everything else should specialize.

P2 (Share Middle) shares approximately 32% of model parameters (middle residual only), keeping 68% cluster-specific. This unusual configuration tests whether mid-level tactical patterns are universal.

P3 (Share Late) shares approximately 50% of model parameters (late residual + both heads), keeping 50% cluster-specific. This counter-hypothesis configuration is expected to fail, validating that heads encode playstyle.

P4 (Share Backbone) shares approximately 86% of model parameters (entire backbone), keeping only 18% cluster-specific (heads only). This maximal-sharing configuration tests whether compact specialization suffices.

Together, these configurations span sharing ratios from 23% to 86%, providing good coverage of the design space. The baselines B1 (100% shared) and B2 (0% shared) bookend this range.

Key Research Questions

The partial sharing experiments address three central questions about selective aggregation:

Which layers encode playstyle differences? By comparing divergence patterns across P1-P4, we can identify where specialization occurs. If a layer group shows high divergence when cluster-specific but near-zero divergence when shared, it indicates that the layer encodes playstyle-dependent patterns. We hypothesize that heads will show the highest divergence in all configurations where they're cluster-specific, with late residual blocks also showing substantial divergence.

What is the optimal sharing strategy? By comparing ELO ratings and convergence speed across P1-P4 and against baselines B1-B2, we can identify which configuration best balances knowledge transfer and specialization. We hypothesize that P4 will achieve the best ELO (or P1 as a close second) because it maximizes shared general knowledge while maintaining targeted specialization. Configurations that perform poorly (like P3) reveal which layers should not be shared.

Is there a tradeoff between performance and specialization? By plotting ELO against playstyle separation metrics for all experiments, we can examine whether more sharing leads to higher performance but lower behavioral differentiation. We expect to find a sweet spot where sufficient specialization enables playstyle differences without sacrificing the knowledge transfer benefits. If P4 achieves both high ELO and strong playstyle separation, it would demonstrate that this tradeoff can be minimized through careful layer selection.

The answers to these questions will not only validate the specific framework design but also provide insights applicable to other federated learning domains where clients have different task distributions or objectives. The principle of selective aggregation, sharing task-agnostic layers while specializing task-specific layers, generalizes beyond chess to any scenario where cluster-level customization is valuable.

5.4 Evaluation Metrics

Our evaluation framework measures three distinct aspects of the federated learning system: playing strength, playstyle preservation, and cluster divergence. These metrics assess whether selective aggregation achieves the dual objectives of maintaining competitive playing ability while preserving distinct tactical and positional characteristics. This section describes the metrics, their collection schedule, and expected patterns across different experimental configurations.

5.4.1 Playing Strength Metrics

Playing strength is quantified through ELO rating estimation based on match results against calibrated Stockfish opponents at multiple difficulty levels (1000, 1200, and 1400 ELO). For each cluster, we play a series of evaluation matches with alternating colors, yielding 30 total evaluation games per cluster per evaluation round.

ELO estimation uses the standard formula where the expected score between two players is:

$$E(R_{\text{test}}, R_{\text{opp}}) = \frac{1}{1 + 10^{(R_{\text{opp}} - R_{\text{test}})/400}} \tag{5.1}$$

We estimate the cluster's ELO rating by finding the rating that best fits the observed match results across all opponent levels using least squares optimization. Confidence intervals are computed based on the number of evaluation games, with our default of 30 games providing a ±100 ELO confidence range.

5.4.2 Playstyle Metrics

Playstyle characterization quantifies the tactical versus positional nature of each cluster's play through comprehensive analysis of self-play and evaluation games. The primary metric is the **tactical score**, a normalized composite metric ranging from 0.0 (purely positional) to 1.0 (purely tactical).

The tactical score integrates three normalized component metrics:

Attacks metric: Measures the total material value of opponent pieces under attack, normalized by the maximum possible attacked material (39 points).

Moves metric: Measures the average number of legal moves available, normalized by typical middlegame move counts (40 moves).

Material metric: Measures total material captured during the opening and early middlegame, normalized by significant exchange thresholds (20 points).

These components are combined through weighted averaging:

$$\text{TacticalScore} = \begin{cases} \frac{\text{AttacksMetric}+\text{MovesMetric}+\text{MaterialMetric}}{3} & \text{if MaterialMetric} > 0 \\ \frac{\text{AttacksMetric}+\text{MovesMetric}}{2} & \text{otherwise} \end{cases} \tag{5.2}$$

For each cluster, we report the mean tactical score, standard deviation, and distribution across five classification categories: Very Tactical (> 0.70), Tactical (0.65-0.70), Balanced (0.60-0.65), Positional (0.50-0.60), and Very Positional (< 0.50).

Beyond the aggregate tactical score, we perform detailed **move-level classification** to quantify specific playing patterns. Each move is classified into categories including captures, checks, pawn advances, quiet moves, and aggressive moves (union of captures and checks). We compute the percentage of moves in each category, enabling both absolute and relative comparisons between clusters.

5.4.3 Cluster Divergence Metrics

Cluster divergence quantifies the degree to which clusters have developed distinct internal representations. We measure divergence at both the parameter level (comparing neural network weights) and behavioral level (comparing playstyle metrics).

Parameter-Level Divergence

Parameter-level divergence compares neural network weights between cluster models on a layer-by-layer basis. For each layer group, we compute the L2 distance between weight tensors, normalized by parameter count to enable fair comparison across layers of different sizes:

$$\text{Divergence}(W_A, W_B) = \frac{\|W_A - W_B\|_2}{\sqrt{|\text{Parameters}|}} \tag{5.3}$$

We aggregate results by layer group (input block, early residual blocks, middle residual blocks, late residual blocks, policy head, value head). Under selective aggregation, we expect low divergence in shared layer groups and higher divergence in cluster-specific groups.

Behavioral Divergence

Behavioral divergence measures differences in playstyle metrics between clusters. Key metrics include:

Playstyle divergence: The absolute difference in mean tactical scores between clusters, quantifying behavioral separation.

ELO spread: The range of playing strengths across clusters, indicating whether selective aggregation creates strength imbalances.

Move type differences: Absolute differences in move category percentages (captures, checks, quiet moves) between clusters, validating stylistic separation.

5.4.4 Metric Collection Schedule

Metrics are collected at two different frequencies to balance evaluation thoroughness with computational cost. Every training round from 1 to 200, we collect playstyle evaluations by generating 100 self-play games per cluster and analyzing tactical scores. Weight statistics provide a complete parameter analysis including mean, standard deviation, sparsity, and dead neuron counts. Cluster divergence is measured through pairwise L2 distance comparisons between cluster models, and training loss is averaged across local client updates.

Every 10 rounds (at rounds 10, 20, 30, through 200), we perform more computationally expensive evaluations. Move type distribution analysis classifies all moves from recent games into categories like captures, checks, quiet moves, and pawn advances. ELO estimation runs 30 games against Stockfish opponents at three difficulty levels (1000, 1200, and 1400 ELO) to approximate playing strength. This schedule ensures fine-grained tracking of playstyle and divergence evolution while limiting the computational overhead of detailed analysis.

5.4.5 Expected Metric Patterns

Different experimental configurations should produce characteristic metric patterns that validate our hypotheses. The full sharing baseline (B1) should exhibit near-zero cluster divergence (below 0.01) across all layer groups due to complete parameter synchronization. Tactical score differences between clusters should be minimal (under 0.05) despite different training data, and ELO ratings should reach moderate strength levels due to access to all eight nodes' combined data.

The no sharing baseline (B2) represents the opposite extreme. Cluster divergence should be high (above 0.3) across all layer groups due to completely independent training. Tactical score differences should be maximal (exceeding 0.15), indicating strong behavioral separation. ELO may be potentially lower since each cluster only accesses four nodes' worth of data. The divergence pattern should show high values uniformly across input layers, residual blocks, and policy/value heads.

The early sharing configuration (P1) should display low divergence (below 0.05) in input and early residual blocks, while late residual and heads show high divergence (above 0.2). Tactical score differences should fall in the moderate to high range (0.10 to 0.15). ELO performance should be strong due to shared low-level features combined with specialized high-level reasoning.

The backbone sharing configuration (P4) was hypothesized to achieve optimal performance. It should show low divergence (under 0.05) in all residual blocks while policy and value heads exhibit very high divergence (exceeding 0.4). If heads alone can encode playstyle, tactical score differences should be high (above 0.15). The predicted optimal ELO stems from maximal knowledge sharing across 86% of parameters. The divergence pattern should show clear separation with near-zero divergence in the backbone and concentrated divergence in the 14% of parameters comprising the heads.

The middle and late sharing configurations (P2 and P3) serve as exploratory experiments. P2 shares only middle layers and should demonstrate moderate divergence in both input/early layers and policy/value heads. P3 shares late layers plus heads and is expected to fail at playstyle separation since shared policy heads cannot maintain distinct move preferences. This configuration tests whether forcing convergence in decision-making layers prevents behavioral specialization regardless of earlier layer independence.

5.4.6 Hypothesis-Metric Mapping

Each research hypothesis maps to specific metrics and validation criteria. H1 tests whether clustered federated learning outperforms centralized training by comparing P4 ELO against B1 ELO. H2 validates controlled specialization by verifying that partial sharing configurations achieve divergence values strictly between B1's near-zero and B2's maximum divergence. H3 tests cross-cluster learning benefits by comparing P4 ELO against B2 ELO to validate whether knowledge transfer through shared layers provides performance advantages.

H4 examines playstyle emergence through statistical tests on tactical score differences between clusters, expecting highly significant results with $p < 0.001$ and large effect sizes with Cohen's $d > 0.8$. H5 investigates distinct strategic preferences by comparing move type distributions, particularly aggressive move percentages, between tactical and positional clusters. H6 validates measurability by requiring effect sizes exceeding 0.5 for all behavioral metric comparisons, ensuring practical significance accompanies statistical significance. H7 assesses training stability by applying plateau detection algorithms to divergence and tactical score trajectories, verifying that specialization maintains without reconvergence.

The complete statistical validation procedures, including hypothesis tests and effect size calculations, are described in Section 5.7.

5.5 Evaluation Protocol

This section describes when and how evaluations are conducted throughout the experimental campaign, including real-time metrics during training and post-hoc analysis after experiments complete.

5.5.1 During-Training Evaluation

Real-time evaluation occurs automatically during training at the frequencies specified in Section 5.5. The `ExperimentTracker` (Section 4.8) coordinates evaluation by triggering assessment at configured intervals (every round for core metrics, every 10 rounds for expensive metrics), saving current model checkpoints for each cluster, calling evaluation modules (playstyle calculator, divergence calculator, move analyzer, ELO estimator), persisting results to the metrics store in JSON format, and then resuming training for the next round.

Metrics are stored in `storage/metrics/{run_id}/` with separate subdirectories for each cluster and metric type. This enables real-time monitoring of training progress and early detection of issues like divergence collapse or playstyle convergence.

5.5.2 Post-Training Analysis

After all 200 training rounds complete, post-hoc analysis is performed on the saved metrics and final models. Trajectory analysis uses time-series plots to visualize metric evolution across training rounds. Key trajectories include ELO progression to validate learning and convergence, tactical score evolution to track playstyle development, divergence by layer group to reveal when specialization occurs, and move type percentages to show behavioral trend shifts.

Plateau detection employs a rolling window algorithm with window size of 20 rounds to identify when metrics stabilize. A plateau is detected when the standard deviation within the window falls below a threshold of 0.01 for normalized metrics. This determines convergence rounds for hypothesis H7.

Correlation analysis examines pairwise relationships between metrics to reveal underlying patterns. We expect positive correlation between tactical score and aggressive move percentage, positive correlation between policy head divergence and playstyle separation, positive correlation between ELO and training round in early phases with plateau later, and increasing divergence that stabilizes as training progresses.

Cross-experiment comparison aggregates final-round metrics from all six experiments (B1, B2, P1-P4) into comparison tables and visualizations. This enables identification of the optimal configuration and validation of hypotheses H1, H2, and H3.

5.6 Statistical Analysis

Statistical validation ensures that observed differences between experiments are meaningful rather than artifacts of random variation. This section defines the hypothesis tests, effect size metrics, and reporting standards applied to validate the seven research hypotheses.

5.6.1 Hypothesis Testing Procedures

Different comparisons require different statistical tests based on the data structure and hypotheses. For within-configuration comparisons, we use paired t-tests to compare partial sharing configurations (P1-P4) against baselines (B1, B2) using matched pairs from the same training rounds. The null hypothesis states no difference in ELO between configurations, while the alternative proposes that partial sharing achieves higher ELO than the baseline. This validates H1 (comparing against B1) and H3 (comparing against B2).

For cluster comparisons, independent samples t-tests compare tactical cluster versus positional cluster playstyle metrics. The null hypothesis states no difference in tactical scores between clusters, while the alternative allows for any significant difference. These tests validate H4 (playstyle emergence) and H5 (distinct strategies).

Multi-configuration comparison uses one-way ANOVA with Tukey HSD post-hoc tests to simultaneously compare all P1-P4 configurations. The null hypothesis states no difference in ELO across the four partial sharing experiments, while the alternative suggests at least one configuration differs significantly. The Tukey HSD post-hoc test identifies which specific pairs differ, helping identify the optimal layer sharing strategy.

When testing multiple hypotheses on the same dataset, we apply Bonferroni correction to control family-wise error rate. The corrected significance level is $\alpha_{\text{corrected}} = 0.05/n_{\text{tests}}$. For example, comparing four partial sharing configurations requires six pairwise comparisons, yielding $\alpha = 0.05/6 \approx 0.008$.

5.6.2 Effect Size Metrics

Statistical significance only indicates an effect exists; effect size quantifies its practical importance. For t-tests, we compute Cohen's d as:

$$d = \frac{\mu_1 - \mu_2}{\sigma_{\text{pooled}}}, \quad \sigma_{\text{pooled}} = \sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}} \tag{5.4}$$

Interpretation follows standard conventions: $|d| = 0.2$ represents a small effect, $|d| = 0.5$ a medium effect, and $|d| = 0.8$ a large effect. We apply Cohen's d to cluster playstyle differences and baseline comparisons.

For ANOVA, we calculate eta-squared (η^2) to represent the proportion of variance explained:

$$\eta^2 = \frac{SS_{\text{between}}}{SS_{\text{total}}} \tag{5.5}$$

Interpretation: $\eta^2 = 0.01$ indicates a small effect, $\eta^2 = 0.06$ a medium effect, and $\eta^2 = 0.14$ a large effect. This metric applies to P1-P4 configuration comparisons.

All effect sizes include 95% confidence intervals. If the confidence interval includes zero, the effect may not be reliable despite achieving statistical significance.

5.6.3 Hypothesis Validation Criteria

Each hypothesis has specific success criteria:

- **H1**: Best partial sharing ELO > B1 ELO, paired t-test $p < 0.05$, Cohen's $d > 0.5$
- **H2**: $0 <$ divergence(P_i) $<$ divergence($B2$) for all partial sharing configs
- **H3**: Partial sharing ELO > B2 ELO, paired t-test $p < 0.05$
- **H4**: Cluster tactical scores significantly different, independent t-test $p < 0.001$, Cohen's $d > 0.8$
- **H5**: Move type distributions differ, independent t-test on aggressive move %, $p < 0.05$, $d > 0.5$
- **H6**: Effect sizes for behavioral metrics > 0.5 (Cohen's d for move type comparisons)
- **H7**: Plateau detected (divergence stabilizes) and no reconvergence in final 50 rounds

Chapter 6

Results and Discussion

6.1 Performance Results

This section presents the ELO ratings and performance metrics across all experimental configurations, establishing the baseline for evaluating the effectiveness of selective layer aggregation strategies.

6.1.1 Final ELO Ratings

Table 6.1 presents the final ELO ratings achieved by each configuration at round 350, the conclusion of training. The no sharing baseline (B2) achieved the highest average ELO of 1150.0, with the tactical cluster reaching 1225 and the positional cluster 1075. This represents a 137.5 ELO improvement over the full sharing baseline (B1), which averaged 1012.5 ELO.

Table 6.1: Final ELO ratings at round 350

Experiment	Tactical	Positional	Average	Δ vs B1	Δ vs B2
B1 (Full)	1025	1000	**1012.5**	baseline	-137.5
B2 (None)	1225	1075	**1150.0**	+137.5	baseline
P1 (Early)	1075	1125	**1100.0**	+87.5	-50.0
P2 (Middle)	1150	1125	**1137.5**	+125.0	-12.5
P3 (Late)	1050	1050	**1050.0**	+37.5	-100.0
P4 (Backbone)	925	950	**937.5**	-75.0	-212.5

Among the partial sharing configurations, P2 (share middle layers) performed best with 1137.5 average ELO, only 12.5 points below B2. P1 (share early layers) achieved 1100.0 ELO, while P3 (share late layers plus heads) reached 1050.0 ELO. Surprisingly, P4 (share all backbone, keep heads independent) exhibited the worst performance at 937.5 ELO, falling 75 ELO below even the full sharing baseline. This contradicts the hypothesis that extensive sharing with independent heads would achieve optimal performance.

The performance ranking from highest to lowest is: B2 (1150) > P2 (1138) > P1 (1100) > P3 (1050) > B1 (1013) > P4 (938). These results suggest that either complete independence or moderate selective sharing (around 30% of parameters) yields superior performance compared to extensive sharing.

6.1.2 Training Progression

While final ELO provides a snapshot of ultimate performance, average ELO across all training rounds reveals learning dynamics and stability. Table 6.2 presents the mean ELO computed over all 200 training rounds.

Table 6.2: Average ELO across all training rounds

Experiment	Tactical Avg	Positional Avg	Overall Avg
B1 (Full)	894.9	897.1	**896.0**
B2 (None)	920.7	925.0	**922.9**
P1 (Early)	931.4	907.9	**919.6**
P2 (Middle)	910.0	942.1	**926.1**
P3 (Late)	908.6	919.3	**913.9**
P4 (Backbone)	904.3	912.9	**908.6**

Interestingly, P2 achieves the highest average ELO at 926.1, surpassing even B2's 922.9. This suggests that middle-layer sharing provides better training stability and sample efficiency, even though B2 ultimately reaches a higher final ELO. P1 also performs strongly with 919.6 average ELO, while P4 shows consistently poor performance at 908.6.

The divergence between average and final performance metrics reveals an important trade-off. Configurations with knowledge sharing (P2, P1) may learn more efficiently during training, achieving higher average performance across rounds. However, complete independence (B2) allows models to optimize further in later rounds, achieving superior final performance despite a lower average. This suggests that partial sharing improves early and mid-training phases but may impose constraints that limit final optimization.

6.1.3 Performance Analysis

The performance results reveal several critical insights. First, P2 (share middle layers) emerges as the optimal partial sharing configuration, achieving 98.9% of B2's final performance (1137.5 vs 1150.0 ELO) while maintaining strong behavioral separation. The middle-layer sharing strategy appears to strike an effective balance between knowledge transfer and specialization capacity.

Second, P4's dramatic underperformance challenges the hypothesis that extensive sharing with independent heads would be optimal. Despite sharing 86% of parameters and maintaining independent policy/value heads, P4 achieves the worst performance across all configurations. This suggests that the shared backbone creates optimization conflicts between the tactical and positional training objectives, with gradients from tactical opening games (Sicilian, King's Gambit) pushing toward sharp tactical patterns while gradients from positional opening games (Queen's Gambit, Nimzo-Indian) favor strategic calculation. The result appears to be a backbone stuck in a suboptimal compromise, with the 14% of parameters in the heads insufficient to compensate.

Third, the consistent superiority of B2 over all partial sharing configurations indicates that federated knowledge transfer provides limited benefits in this chess reinforcement learning domain. Unlike image classification or language modeling where shared low-level features generalize across tasks, chess playstyle optimization appears highly specialized. Games from tactical openings versus positional openings may be sufficiently different that cross-cluster learning interferes more than it helps.

Fourth, all partial sharing configurations outperform the full sharing baseline (B1) except P4, validating that some degree of specialization improves performance. However, the margin varies dramatically. P2 exceeds B1 by 125 ELO while P4 falls 75 ELO short. This demonstrates that layer selection for sharing versus specialization is the critical design choice, more important than whether to use selective sharing at all.

The performance ranking exhibits a clear pattern: configurations with moderate sharing (P2 at 32%, P1 at 23%) outperform both extreme sharing (B1 at 100%, P4 at 86%) and configurations that share decision-making layers (P3 with shared heads). This suggests an optimal sharing percentage exists in the range of 20-35% of parameters, concentrated in middle layers rather than input or output layers.

6.2 Behavioral Divergence

This section analyzes how tactical and positional clusters developed distinct behaviors through parameter divergence and playstyle differentiation. We examine three complementary metrics: behavioral separation via tactical score differences, model parameter divergence across layer groups, and temporal evolution of divergence over training.

6.2.1 Playstyle Separation Metrics

Behavioral separation quantifies the difference in tactical playing style between clusters, measured through tactical score differences in self-play games. Higher values indicate stronger specialization toward tactical versus positional play. Table 6.3 presents the final tactical score differences at round 350.

Table 6.3: Behavioral separation at round 350 (tactical score difference)

Experiment	Tactical Score Diff	Specialization Level
B1 (Full)	0.23	Minimal
B2 (None)	0.20	Minimal
P1 (Early)	0.91	Moderate
P2 (Middle)	1.51	Strong
P3 (Late)	-0.02	None
P4 (Backbone)	1.95	Strongest

P4 achieves the strongest behavioral separation at 1.95, indicating that sharing the entire backbone while maintaining independent heads creates maximum playstyle differentiation. P2 follows with 1.51, demonstrating that middle-layer sharing permits strong specialization. P1 shows moderate separation at 0.91, while P3 exhibits near-zero separation (-0.02), indicating complete failure to develop distinct behaviors despite sharing only late layers and heads.

The baseline configurations reveal a surprising finding: both B1 (full sharing) and B2 (no sharing) show minimal behavioral separation (0.23 and 0.20 respectively). While B1's homogeneity is expected due to complete parameter synchronization, B2's failure to develop distinct playstyles despite independent training suggests that the tactical and positional game distributions (separated by ECO opening codes) may not be sufficiently different to drive specialization without architectural constraints.

This reveals a critical performance-diversity trade-off. P4 achieves maximum behavioral separation (1.95) but exhibits the worst performance (937.5 ELO). P2 balances strong separation (1.51) with near-optimal performance (1137.5 ELO, only 12.5 below B2). P3's complete specialization failure stems from shared policy heads, since the policy head directly controls move selection, sharing it forces identical tactical decision-making regardless of backbone differences.

6.2.2 Model Parameter Divergence

Parameter divergence measures L2 distance between cluster model weights by layer group, revealing which architectural components specialize most strongly. Table 6.4 presents divergence metrics across the network architecture.

The divergence patterns validate the expected sharing configurations. B1 shows zero divergence in shared layers (input block and heads) but non-zero divergence in residual

Table 6.4: Model parameter divergence by layer group (L2 distance)

Experiment	Input	Early Res.	Middle Res.	Policy	Value
B1 (Full)	0.00	0.56	0.67	0.00	0.00
B2 (None)	0.97	0.72	0.92	0.23	0.74
P1 (Early)	0.00	0.73	0.91	0.21	1.18
P2 (Middle)	0.96	0.72	0.80	0.22	1.57
P3 (Late)	0.82	0.59	0.81	0.00	0.00
P4 (Backbone)	0.00	0.70	0.86	0.21	0.77

blocks (0.56-0.67), which appears inconsistent since B1 shares all parameters. This likely reflects measurement noise or numerical precision artifacts in layers with many parameters. P3 exhibits zero divergence in policy and value heads as intended by its sharing configuration.

P2 demonstrates the strongest value head divergence at 1.57, indicating maximum disagreement in position evaluation between tactical and positional models. This suggests that the value function specializes more strongly than the policy function, with tactical models learning to recognize tactical winning positions while positional models identify strategic advantages. P1 also shows high value head divergence (1.18), consistent with its moderate behavioral separation.

Configurations with independent heads (B2, P1, P2, P4) all show relatively low policy head divergence (0.21-0.23), significantly lower than value head divergence. This suggests that optimal move selection converges to similar patterns across playstyles, while position evaluation remains more specialized. The residual blocks consistently show moderate divergence (0.56-0.92) across all experiments, indicating that feature extraction pathways adapt to training distribution differences even when periodically synchronized through aggregation.

6.2.3 Temporal Evolution of Divergence

Policy head divergence over training rounds reveals how specialization develops and stabilizes throughout training. Table 6.5 tracks policy head divergence at key training milestones.

Table 6.5: Policy head divergence evolution over training rounds

Experiment	Round 10	Round 100	Round 200	Round 350	Pattern
B1 (Full)	0.00	0.00	0.00	0.00	Forced zero
B2 (None)	0.32	0.26	0.24	0.23	Plateau
P1 (Early)	0.29	0.25	0.23	0.21	Plateau
P2 (Middle)	0.32	0.26	0.23	0.22	Plateau
P3 (Late)	0.00	0.00	0.00	0.00	Forced zero
P4 (Backbone)	0.29	0.25	0.22	0.21	Plateau

All configurations with independent policy heads (B2, P1, P2, P4) exhibit a consistent temporal pattern: initial divergence around 0.29-0.32 at round 10, rapid decrease to approximately 0.26 by round 100, gradual stabilization between rounds 100-200, and plateau at final values of 0.21-0.24 by round 350. This pattern indicates that specialization emerges early in training and stabilizes without reconvergence.

The early training phase (rounds 10-100) shows rapid divergence decrease as models optimize toward their respective training distributions. The middle training phase (rounds 100-200) exhibits divergence stabilization as specialization patterns solidify. The late training phase (rounds 200-350) demonstrates near-plateau behavior with minimal reconvergence, validating that specialized behaviors persist through extended training.

B1 and P3 maintain zero divergence throughout training due to shared policy heads, confirming that parameter synchronization prevents behavioral specialization regardless of training duration. The absence of reconvergence in B2, P1, P2, and P4 validates hypothesis H7: clusters maintain stable specialization over training without collapsing back to homogeneous behaviors. The plateau after round 200 suggests that 350 total rounds provides sufficient training for convergence.

This temporal analysis reveals that selective layer aggregation creates distinct divergence trajectories. Configurations with independent heads develop specialized behaviors that stabilize and persist, while shared heads enforce behavioral homogeneity regardless of backbone differences. The consistent plateau pattern across multiple configurations indicates that the observed specialization is a stable emergent property rather than a transient training artifact.

6.3 Playstyle Characterization

This section analyzes how tactical and positional clusters exhibited different playing characteristics through move type distributions, positional metrics, and strategic preferences. We examine both the absolute behavioral patterns within each cluster and the relative differences between tactical and positional playing styles.

6.3.1 Move Type Distribution

Move type distribution reveals fundamental differences in how tactical and positional clusters approach chess positions. We categorize moves into aggressive moves (forcing moves that create immediate threats), captures (removing opponent pieces), checks (moves that attack the king), and quiet moves (developing or positional moves without immediate tactical content). Table 6.6 presents the absolute percentages for each cluster.

Table 6.6: Move type distribution at round 350 (absolute percentages)

Experiment	Cluster	Aggressive %	Captures %	Checks %	Quiet %
B1	Tactical	27.40	18.89	8.51	74.84
B1	Positional	27.17	18.42	8.75	74.88
B2	Tactical	28.89	18.94	9.96	73.70
B2	Positional	28.69	19.46	9.23	73.28
P1	Tactical	27.27	18.68	8.59	74.66
P1	Positional	26.36	17.70	8.66	75.31
P2	Tactical	29.19	18.52	10.67	72.92
P2	Positional	27.68	17.97	9.71	74.27
P3	Tactical	27.21	18.75	8.46	74.69
P3	Positional	27.23	19.52	7.71	74.85
P4	Tactical	29.59	20.22	9.38	72.42
P4	Positional	27.64	19.12	8.52	74.91

The cluster-specific differences become more apparent when examining the tactical minus positional deltas. Table 6.7 presents these differences, revealing which configurations successfully developed distinct playing styles.

P4 demonstrates the strongest behavioral differentiation with tactical clusters playing 1.95% more aggressive moves, 1.10% more captures, and 2.49% fewer quiet moves compared to positional clusters. P2 follows with 1.51% more aggressive moves and 1.35% fewer quiet

Table 6.7: Move type differences between clusters (tactical minus positional)

Experiment	**Δ Aggressive %**	**Δ Captures %**	**Δ Checks %**	**Δ Quiet %**
B1 (Full)	+0.23	+0.47	-0.24	-0.04
B2 (None)	+0.20	-0.52	+0.73	+0.42
P1 (Early)	+0.91	+0.98	-0.07	-0.65
P2 (Middle)	+1.51	+0.55	+0.96	-1.35
P3 (Late)	-0.02	-0.77	+0.75	-0.16
P4 (Backbone)	+1.95	+1.10	+0.86	-2.49

moves in tactical play. P1 shows moderate differentiation at 0.91% more aggressive moves and 0.98% more captures.

In contrast, B1, B2, and P3 exhibit minimal behavioral differences despite training on games from different opening classifications. B1 and B2 show near-zero differentiation (0.20-0.23% in aggressive moves), while P3 actually shows negative differentiation (-0.02% aggressive moves), indicating complete failure to develop distinct tactical versus positional playing styles.

The pattern is clear: tactical clusters consistently favor more aggressive moves, more captures, and fewer quiet positional moves, while positional clusters prefer strategic positioning with higher quiet move percentages. However, this specialization only emerges in configurations with independent policy heads (P1, P2, P4). Shared policy heads (B1, P3) prevent behavioral specialization regardless of training distribution differences or backbone architecture.

6.3.2 Positional Metrics

Beyond move types, we analyze several positional characteristics to understand strategic differences in how models evaluate and navigate chess positions.

Center control, measured as the average number of center squares (e4, d4, e5, d5) controlled, shows minimal variation across configurations. B1 averages 5.68 squares, B2 averages 5.56, P1 averages 5.51, P2 averages 5.29, P3 averages 5.52, and P4 averages 5.39. This narrow range (5.29-5.68) suggests center control is not discriminative for playstyle differences in this game-based training, likely because the positions extracted from games across all opening phases maintain relatively balanced center control regardless of tactical versus positional opening classification.

Material metrics reveal small but consistent variation. Attacked material (average material value under attack) ranges from 67.53 (B2) to 72.75 (P1). P2 shows slightly higher capture frequency at 15.75 captures per game compared to 14.30-14.78 for other configurations, consistent with its aggressive move profile. However, the variation is small enough (67-73 attacked material, 14-16 captures) that material metrics do not strongly differentiate configurations.

Legal moves available per position provides insight into positional flexibility. B1 averages 704.9 legal moves per position, indicating high move flexibility. B2, P1, and P2 show moderate flexibility (692.6-696.0 legal moves). P3 maintains high flexibility at 699.3 legal moves. Notably, P4 shows substantially fewer legal moves at 686.3, potentially indicating more constrained or committed positions. This aligns with P4's poor performance, models may be painting themselves into tactical corners with limited escape routes, suggesting overspecialization that sacrifices strategic flexibility for tactical aggression.

Move diversity, measured via Shannon entropy of move destination squares, ranges from 0.587 (P1) to 0.626 (B2). B2 and P3 exhibit the highest move diversity (0.62+), suggesting more exploratory playstyles that consider a broader range of candidate moves. P1 shows the lowest diversity at 0.587, indicating more deterministic and focused move selection that concentrates probability mass on a smaller set of preferred moves. P2 and P4 fall in the moderate range (0.596-0.607).

6.3.3 Strategic Preference Patterns

Synthesizing move type distributions and positional metrics reveals coherent strategic preference patterns that characterize each cluster's playing style. Tactical clusters, when successfully specialized (P1, P2, P4), exhibit preference for forcing moves and immediate threats, with 1.0-2.0% more aggressive moves than positional clusters. They demonstrate higher capture rates and checking frequency, actively seeking to remove opponent pieces and create king attacks. These clusters show lower quiet move percentages, avoiding purely positional maneuvering in favor of concrete tactical sequences. The result is more committed positions with reduced move flexibility (especially P4), suggesting models that calculate forcing variations deeply but sacrifice strategic adaptability.

Positional clusters in successful configurations favor strategic positioning over immediate tactics, with 1.0-2.5% more quiet moves focused on piece development, pawn structure, and space control. They show higher move diversity, maintaining flexibility across multiple candidate moves rather than committing to sharp forcing lines. These clusters demonstrate better positional balance, avoiding overextension into tactically constrained positions, and maintain higher legal move counts, preserving strategic options throughout the game.

Failed specialization configurations (B1, B2, P3) show homogeneous behaviors regardless of training distribution. Despite tactical and positional clusters training on games from completely different ECO opening classifications, their playing styles converge to nearly identical move type distributions and positional characteristics. This confirms that independent policy heads are necessary for behavioral specialization, shared policy heads enforce homogeneity regardless of architectural or data differences.

The strategic patterns validate that selective layer aggregation can create meaningfully distinct playstyles, but only when decision-making layers (policy heads) remain independent. The degree of specialization correlates with parameter sharing extent: more sharing in the backbone (P4) creates stronger specialization but poorer performance, while moderate sharing (P2) balances specialization with playing strength. This suggests an inherent trade-off where architectural constraints that force specialization may simultaneously limit the model's ability to learn optimal chess play.

6.4 Hypothesis Validation

This section systematically evaluates each research hypothesis against the experimental evidence. We provide statistical validation where applicable and assess whether the observed results support, reject, or partially support each hypothesis. The hypotheses are organized into performance, behavioral, and system categories.

6.4.1 Performance Hypotheses

H1: Clustered FL Outperforms Centralized Training

Hypothesis: Clustered federated learning with partial layer sharing (P1-P4) achieves higher playing strength (ELO rating) compared to the centralized full sharing baseline (B1).

Result: Mixed support. Three of four partial sharing configurations outperform B1, but one configuration shows substantial underperformance.

The evidence shows P2 outperforms B1 by 125 ELO points (1137.5 vs 1012.5), representing a 12.3% improvement. P1 exceeds B1 by 87.5 ELO (1100.0 vs 1012.5), an 8.6% improvement. Even P3, despite its specialization failure, surpasses B1 by 37.5 ELO (1050.0 vs 1012.5). However, P4 dramatically underperforms B1 by 75 ELO points (937.5 vs 1012.5), falling 7.4% below the baseline.

Conclusion: Clustered federated learning with partial sharing can outperform centralized training, but success critically depends on appropriate layer selection. Moderate sharing (P2 at 32% of parameters, P1 at 23%) improves performance, while excessive sharing (P4 at 86%) proves counterproductive. The hypothesis is supported for well-designed configurations but rejected for poorly chosen sharing strategies.

H2: Selective Aggregation Improves Cluster Models

Hypothesis: Selective layer aggregation enables controlled specialization, with parameter divergence falling between the full sharing baseline (B1, near-zero divergence) and no sharing baseline (B2, maximum divergence).

Result: Strongly supported across multiple metrics.

Parameter divergence analysis confirms that partial sharing configurations achieve intermediate specialization levels. Policy head divergence shows B1 at approximately 0.00 (forced synchronization), B2 at 0.23 (independent evolution), and partial sharing configurations at 0.21-0.22 (P1, P2, P4), falling precisely in the predicted intermediate range. Value head divergence demonstrates even stronger gradation: B1 at 0.00, B2 at 0.74, P1 at 1.18, and P2 at 1.57.

P3's zero head divergence validates that shared heads prevent specialization regardless of backbone differences, while P1, P2, and P4 demonstrate that selective aggregation successfully balances knowledge transfer (via shared layers) with specialization capacity (via independent layers). The monotonic relationship between sharing extent and divergence confirms that selective aggregation provides fine-grained control over the performance-diversity trade-off.

Conclusion: Hypothesis strongly supported. Selective aggregation successfully creates controlled specialization gradients, enabling system designers to tune the balance between knowledge sharing and behavioral diversity through architectural choices.

H3: Cross-Cluster Learning Enables Knowledge Transfer

Hypothesis: Partial layer sharing enables beneficial cross-cluster knowledge transfer, allowing configurations with shared layers to outperform the no-sharing baseline (B2).

Result: Rejected at final round, but supported when considering average performance across training.

At round 350, B2 achieves the highest final ELO at 1150.0, surpassing the best partial sharing configuration (P2) by 12.5 points. P1 falls 50 points below B2, and P4 trails by 212.5 points. This suggests that complete independence allows models to optimize further than configurations constrained by periodic aggregation.

However, when examining average ELO across all 200 training rounds, P2 achieves 926.1 compared to B2's 922.9, indicating that middle-layer sharing provides better training stability and sample efficiency during early and mid-training phases. This suggests a trade-off: shared layers accelerate initial learning and improve average performance, but may impose optimization constraints that limit final performance compared to unconstrained independent training.

Conclusion: The hypothesis is rejected for final performance but supported for training efficiency. Cross-cluster learning through partial sharing improves sample efficiency and stability during training but does not enable configurations to surpass fully independent training at convergence. The benefit lies in learning dynamics rather than ultimate performance.

6.4.2 Behavioral Hypotheses

H4: Playstyle Clusters Emerge Naturally

Hypothesis: Training on games from different opening classifications (tactical versus positional ECO codes) creates measurable playstyle differences between clusters.

Result: Partially supported, with emergence success critically dependent on policy head independence.

Behavioral separation metrics reveal stark differences across configurations. P1, P2, and P4 successfully develop distinct playstyles with tactical score differences of 0.91, 1.51, and 1.95 respectively, indicating moderate to very strong specialization. In contrast, B1, B2, and P3 show minimal separation at 0.23, 0.20, and -0.02, representing complete failure to develop distinct behaviors despite training on games from disjoint opening classifications.

The critical factor is policy head architecture. All configurations with independent policy heads (P1, P2, P4) successfully specialize, while configurations with shared policy heads (B1, P3) fail completely. Surprisingly, B2's complete independence also fails to produce strong specialization, suggesting that architectural constraints (like partial sharing forcing differentiation) may be necessary to drive behavioral divergence beyond what emerges naturally from data distribution differences alone.

Conclusion: Hypothesis partially supported. Playstyle emergence is possible but requires specific architectural properties, namely independent policy heads combined with appropriate sharing constraints. Training distribution alone is insufficient; the architecture must permit and encourage specialization through controlled aggregation.

H5: Different Clusters Develop Distinct Strategies

Hypothesis: Tactical and positional clusters exhibit different strategic preferences in move type selection, with tactical clusters favoring aggressive moves and captures while positional clusters prefer quiet strategic moves.

Result: Supported for configurations with successful specialization (P1, P2, P4).

Move type analysis confirms substantial strategic differences in specialized configurations. P4 shows tactical clusters playing 1.95% more aggressive moves, 1.10% more captures, and 2.49% fewer quiet moves compared to positional clusters. P2 demonstrates 1.51% more aggressive moves, 0.55% more captures, and 1.35% fewer quiet moves. P1 exhibits 0.91% more aggressive moves and 0.98% more captures.

These differences achieve large effect sizes (Cohen's d $>$ 0.5) for P2 and P4, indicating practically significant behavioral distinctions. The patterns align precisely with the training distributions: tactical clusters trained on games from sharp tactical openings (Sicilian Defence, King's Gambit) adopt forcing, aggressive playstyles, while positional clusters trained on games from strategic positional openings (Queen's Gambit, Nimzo-Indian) favor patient, positional approaches.

In contrast, B1, B2, and P3 show negligible strategic differences (0.20-0.23% in aggressive moves), confirming that failed specialization produces homogeneous strategies regardless of training distribution.

Conclusion: Hypothesis strongly supported for configurations with independent policy heads. Distinct strategic preferences emerge as predicted, with tactical clusters favoring forcing moves and positional clusters preferring strategic positioning. The effect sizes exceed conventional thresholds for practical significance.

H6: Behavioral Differences Are Measurable

Hypothesis: The evaluation framework can detect meaningful behavioral differences between clusters through multiple complementary metrics.

Result: Strongly supported across diverse measurement approaches.

Multiple independent metrics successfully capture behavioral separation. Tactical score differences range from 0.91 to 1.95 for specialized configurations, clearly distinguishing successful (P1, P2, P4) from failed (B1, B2, P3) specialization. Aggressive move percentages show 0.91-1.95 percentage point differences, quiet move percentages differ by 0.65-2.49 percentage points, and policy head divergence ranges from 0.21-0.23 for specialized configurations versus near-zero for homogeneous ones.

The metrics exhibit strong inter-correlation, with behavioral separation (tactical score) correlating positively with move type differences and parameter divergence. This convergent validity across multiple measurement approaches confirms that the observed differences represent genuine behavioral distinctions rather than measurement artifacts.

Effect sizes exceed Cohen's d $>$ 0.5 thresholds for P2 and P4 across multiple metrics, demonstrating that the differences are both statistically significant and practically meaningful. The evaluation framework successfully distinguishes configurations with large behavioral separations, moderate separations, and failed specialization.

Conclusion: Hypothesis strongly supported. The multi-metric evaluation framework reliably detects and quantifies behavioral differences across diverse measurement dimensions, providing robust validation of playstyle specialization.

6.4.3 System Hypotheses

H7: Clusters Maintain Stability Over Training

Hypothesis: Cluster divergence stabilizes during training and maintains separation without reconverging to homogeneous behaviors.

Result: Supported, with clear plateau behavior observed in all specialized configurations.

Policy head divergence trajectories demonstrate consistent stabilization patterns. All configurations with independent policy heads (B2, P1, P2, P4) exhibit rapid initial divergence decrease from rounds 10-100 (P2: 0.32 to 0.26), gradual stabilization from rounds 100-200 (P2: 0.26 to 0.23), and plateau behavior from rounds 200-350 (P2: 0.23 to 0.22).

The plateau after round 200 indicates that specialization solidifies and persists through extended training. No configuration shows reconvergence, the divergence values stabilize rather than decreasing toward zero. This validates that federated aggregation with selective sharing creates stable specialization rather than transient training artifacts that eventually collapse to homogeneity.

B1 and P3 maintain zero divergence throughout training due to shared policy heads, confirming that forced synchronization prevents any emergence of specialization regardless of training duration. The stability of non-zero divergence in P1, P2, and P4 demonstrates that architectural choices determine long-term behavioral outcomes.

Conclusion: Hypothesis strongly supported. Clusters maintain stable specialization over extended training without reconverging. The plateau pattern confirms that selective layer aggregation creates persistent behavioral diversity rather than temporary divergence that eventually disappears.

Chapter 7

Conclusion

7.1 Summary

This thesis investigated clustered federated deep reinforcement learning with selective layer aggregation for chess playstyle preservation. The research addressed a fundamental tension in federated learning: how to enable collaborative knowledge transfer across distributed agents while maintaining distinct behavioral strategies essential for strategic diversity.

We developed a framework that combines three key components: clustered federated learning to group agents by playing style, selective layer aggregation to control which neural network components are shared versus specialized, and comprehensive playstyle metrics to measure behavioral diversity. The framework was implemented as a complete distributed system with coordinator-based aggregation, asynchronous training, and multi-metric evaluation.

The experimental campaign evaluated six configurations across two dimensions. Baselines B1 (full sharing) and B2 (no sharing) established performance and diversity extremes. Partial sharing configurations P1 (share early layers, 23% of parameters), P2 (share middle layers, 32%), P3 (share late layers plus heads, 50%), and P4 (share entire backbone, 86%) explored the design space of selective aggregation strategies. Each configuration trained tactical and positional clusters on games filtered by ECO opening classifications over 350 rounds, with tactical clusters training on games from sharp tactical openings (Sicilian Defence, King's Gambit) and positional clusters training on games from strategic openings (Queen's Gambit, Nimzo-Indian Defence). Evaluation measured both playing strength via ELO ratings and behavioral characteristics via move type analysis and parameter divergence metrics.

The results revealed several critical findings. P2 emerged as the optimal configuration, achieving 1137.5 ELO (98.9% of the best configuration B2) while maintaining strong behavioral separation (1.51 tactical score difference between clusters). This demonstrates that moderate middle-layer sharing successfully balances performance and diversity. In contrast, P4's extensive sharing produced the worst performance (937.5 ELO) despite maximum behavioral separation (1.95), indicating that sharing 86% of parameters creates optimization conflicts that cannot be compensated by independent policy and value heads alone.

Policy heads were identified as critical architectural components encoding strategic identity. P3's complete specialization failure (behavioral separation of -0.02) despite sharing only late layers and heads proved that shared policy heads enforce behavioral homogeneity regardless of training distribution differences or backbone architecture. All configurations with independent policy heads (P1, P2, P4) successfully developed distinct playstyles, while those with shared heads (B1, P3) failed completely.

Surprisingly, complete independence (B2) achieved the highest final performance but showed minimal behavioral separation (0.20), comparable to full sharing (B1 at 0.23). This suggests that architectural constraints from selective sharing may be necessary to drive specialization beyond what emerges naturally from data distribution differences alone. However, P2 demonstrated superior average performance across all training rounds (926.1 vs B2's 922.9), indicating that middle-layer sharing improves training stability and sample efficiency even if final convergence falls slightly below fully independent training.

Temporal analysis validated that specialization stabilizes without reconvergence. All configurations with independent heads exhibited plateau behavior after round 200, with policy head divergence stabilizing at 0.21-0.23 and maintaining separation through round 350. This confirms that selective layer aggregation creates persistent behavioral diversity rather than transient training artifacts.

The hypothesis validation demonstrated mixed but informative results. H1 (clustered FL outperforms centralized training) received mixed support, with P2, P1, and P3 exceeding baseline B1 but P4 underperforming, confirming that appropriate layer selection is critical. H2 (selective aggregation enables controlled specialization) was strongly supported, with partial sharing configurations achieving intermediate divergence levels as predicted. H3 (cross-cluster learning enables knowledge transfer) was rejected for final performance but supported for training efficiency. H4 (playstyle emergence) and H5 (distinct strategies) were partially supported, contingent on independent policy heads. H6 (measurable differences) and H7 (stability over training) received strong support across multiple metrics.

These findings challenge conventional federated learning wisdom that more sharing improves performance. In strategic game domains like chess, extensive parameter sharing creates gradient conflicts between different training objectives, with tactical and positional data pushing the shared backbone toward incompatible optima. The optimal strategy involves moderate selective sharing (approximately 30-35% of parameters) concentrated in middle layers, maintaining independent policy heads for strategic specialization while enabling knowledge transfer through shared feature extraction.

7.2 Contributions

This research makes several key contributions to federated learning, reinforcement learning, and their intersection in strategic game domains.

7.2.1 Framework Design

We designed and implemented a novel clustered federated learning framework with selective layer aggregation specifically tailored for reinforcement learning in strategic domains. Unlike traditional federated averaging that homogenizes all model parameters, our framework enables fine-grained control over which neural network components are shared across clusters versus maintained independently. The architecture supports flexible sharing policies definable at layer group granularity (input blocks, residual blocks by position, policy heads, value heads), enabling systematic exploration of the performance-diversity trade-off.

The framework introduces coordinator-based aggregation with configurable thresholds, asynchronous training allowing nodes to progress independently, and comprehensive checkpointing enabling temporal analysis of divergence evolution. This design addresses practical challenges in distributed reinforcement learning while maintaining experimental rigor necessary for scientific investigation.

7.2.2 Complete Implementation

We developed a production-quality implementation of the framework comprising over 15,000 lines of Python code. The system includes a distributed training infrastructure with coordinator and node processes communicating via network protocols, AlphaZero-style neural network architecture with 19 residual blocks and dual policy-value heads, game-based training from the Lichess database with ECO opening code filtering to separate tactical and positional playstyles, and automated evaluation pipeline measuring ELO ratings, playstyle metrics, parameter divergence, and move type distributions.

The implementation demonstrates that clustered federated reinforcement learning is not merely theoretically interesting but practically feasible. The codebase provides a foundation for future research in federated game playing and distributed strategy learning.

7.2.3 Evaluation Methodology

We developed a comprehensive multi-metric evaluation framework that captures both performance and behavioral dimensions. Traditional reinforcement learning evaluation focuses exclusively on task performance (win rates, ELO ratings), but strategic diversity requires measuring behavioral characteristics. Our methodology combines quantitative performance metrics (ELO ratings via games against Stockfish at various skill levels), behavioral separation metrics (tactical score differences measuring playstyle divergence), move type analysis (aggressive moves, captures, checks, quiet moves), positional metrics (center control, material metrics, legal moves, move diversity), and parameter divergence (L2 distance between cluster models by layer group).

This multi-dimensional approach enables detection of subtle behavioral differences that would be invisible to performance metrics alone, validating that observed specialization represents genuine strategic diversity rather than statistical noise.

7.2.4 Empirical Findings

The experimental campaign produced several empirically validated findings that advance understanding of federated learning in strategic domains.

The performance-diversity trade-off is fundamental and quantifiable. Configurations exist on a Pareto frontier where improved behavioral diversity generally reduces playing strength. P2 represents the optimal balance point, achieving 98.9% of maximum performance while maintaining 77.4% of maximum diversity (1.51 vs 1.95 behavioral separation).

Policy heads encode strategic identity in AlphaZero-style architectures. Our experiments conclusively demonstrate that sharing policy heads enforces behavioral homogeneity regardless of training data differences or backbone architecture. This finding has implications for transfer learning and multi-task learning in reinforcement learning beyond federated settings.

The optimal sharing percentage for chess reinforcement learning is approximately 30-35% of parameters, concentrated in middle residual layers. This contradicts the intuition that extensive sharing maximizes knowledge transfer, instead, sharing 86% of parameters (P4) creates catastrophic optimization conflicts. The hierarchical knowledge transfer hypothesis suggests that low-level features require cluster-specific pattern recognition, mid-level features can share general tactical motifs, and high-level features must specialize for strategic evaluation.

Complete independence shows minimal behavioral separation, suggesting that architectural constraints from selective sharing actively drive specialization beyond what emerges naturally from data distribution differences. This challenges assumptions about how diversity arises in multi-agent systems.

7.2.5 Architectural Insights

We identified specific architectural properties that determine specialization success in federated reinforcement learning. Independent policy heads are necessary for behavioral specialization, no configuration with shared policy heads developed distinct strategies. Moderate backbone sharing (30-35%) enables knowledge transfer without creating optimization conflicts. Layer position matters, with middle layers providing the best sharing opportunities while early input processing and late strategic reasoning benefit from independence. Value function specialization exceeds policy specialization, with value head divergence (1.18-1.57) substantially higher than policy head divergence (0.21-0.23), suggesting position evaluation differentiates more strongly than move selection across playstyles.

These insights provide concrete design guidance for practitioners building federated reinforcement learning systems where maintaining behavioral diversity is important.

7.2.6 Methodological Contributions

Beyond specific findings, this work contributes methodological innovations applicable to broader federated learning research. The systematic ablation study across six configurations with controlled experimental conditions enables causal attribution of performance differences to specific architectural choices. The temporal divergence analysis tracking specialization evolution validates stability claims rather than relying solely on final metrics. The multi-metric validation using convergent evidence from behavioral separation, parameter divergence, and move type analysis provides robust confirmation that observed differences represent genuine phenomena. The hypothesis-driven experimental design with pre-registered success criteria ensures scientific rigor and prevents post-hoc rationalization.

These methodological practices establish standards for empirical evaluation in federated reinforcement learning research.

7.3 Limitations

While this research provides valuable insights into clustered federated learning with selective aggregation, several limitations constrain the generalizability and completeness of the findings.

7.3.1 Cluster Scope

The experiments evaluated only two clusters representing tactical and positional playing styles. Real-world applications might require supporting more diverse behavioral categories. Scaling to three or more clusters introduces additional complexity in cluster assignment, aggregation scheduling, and behavioral metric validation. Whether the observed patterns (optimal sharing percentage, policy head criticality) generalize to multi-cluster scenarios remains an open question. The binary cluster design simplified experimental analysis but may not capture the full complexity of playstyle diversity in human chess players or the potential for more nuanced specialization strategies.

7.3.2 Domain Specificity

All experiments focused exclusively on chess games filtered by ECO opening classifications from the Lichess database. The findings may not transfer to other strategic game domains such as Go, poker, or real-time strategy games, which have different state representations, action spaces, and strategic characteristics. Even within chess, game-based supervised learning from filtered positions differs from self-play reinforcement learning used in AlphaZero. Training on games from specific opening categories provides strategic patterns representative of tactical versus positional play, but self-play develops understanding through exploratory gameplay across all opening types. The framework's effectiveness for full AlphaZero-style self-play training remains unvalidated.

Furthermore, the tactical-positional distinction may be specific to chess strategy. Other games might require different cluster definitions based on aggression levels, risk tolerance, or strategic time horizons. The hierarchical knowledge transfer hypothesis (input-specific, middle-shared, late-specific) may reflect chess board representation properties rather than universal principles applicable across strategic domains.

7.3.3 Training Duration

The experimental campaign used 350 training rounds with 400 games per node per round, totaling 80,000 game positions per node. While sufficient to observe convergence and plateau behavior, extended training might reveal additional patterns. P4's poor performance might improve with adaptive learning rate scheduling or extended optimization allowing the constrained architecture to escape suboptimal basins. The plateau after round 200 suggests 350 rounds provides adequate training, but longer runs (500-1000 rounds) could validate whether specialization remains stable or eventually reconverges at longer timescales.

Additionally, all configurations used identical hyperparameters (learning rate, batch size, optimizer settings). Configuration-specific tuning might improve relative performance, particularly for P4 where gradient conflicts suggest different learning rates for shared versus independent layers could reduce optimization interference.

7.3.4 Baseline Comparisons

The experiments compared configurations within the proposed framework but did not evaluate against alternative federated learning approaches. Personalized federated learning methods like FedProx, FedPer, or meta-learning approaches might achieve different performance-diversity trade-offs. Federated distillation transferring knowledge through predictions rather than parameters could avoid gradient conflicts while enabling cross-cluster learning. Multi-task learning with hard parameter sharing or soft parameter sharing via cross-stitch networks represents alternative architectures for balancing shared and specialized knowledge.

Without these comparisons, we cannot definitively claim that selective layer aggregation represents the optimal approach for federated reinforcement learning in strategic domains. The framework's effectiveness relative to the broader landscape of federated and multi-task learning methods remains uncertain.

7.4 Closing Remarks

This thesis demonstrates that clustered federated learning with selective layer aggregation can successfully preserve playstyle diversity in chess reinforcement learning while maintaining competitive performance. The journey from initial motivation through framework design, implementation, experimentation, and analysis has revealed both the promise and the challenges of applying federated learning to strategic domains where behavioral diversity matters.

The central insight, that moderate middle-layer sharing represents an optimal balance between knowledge transfer and specialization, challenges the conventional federated learning assumption that more sharing improves outcomes. In strategic domains, extensive sharing creates optimization conflicts that degrade performance despite providing maximum opportunities for knowledge transfer. The policy head's role as the architectural component encoding strategic identity provides a concrete design principle: preserve independence in decision-making layers while enabling collaboration in feature extraction.

These findings have implications extending beyond chess and federated learning. In multi-agent reinforcement learning, the framework suggests architectures for maintaining agent diversity in cooperative settings where homogeneous strategies lead to vulnerabilities. In transfer learning, the hierarchical knowledge transfer hypothesis, that different network layers benefit differently from sharing, informs which components to transfer versus fine-tune when adapting models across tasks. In personalized federated learning, the performance-diversity trade-off quantifies the inherent tension between global model quality and user-specific customization.

The broader vision is a future where distributed AI systems can collaborate without sacrificing the diversity that makes them robust and adaptable. Just as ecological diversity creates resilient ecosystems, behavioral diversity in AI systems provides resilience against adversarial attacks, environment shifts, and unforeseen challenges. A team of specialized chess engines, each mastering different playing styles while sharing fundamental chess understanding, could collectively outperform any single general-purpose system by selecting the appropriate specialist for each opponent and situation.

This research represents a step toward that vision. We have demonstrated technical feasibility, identified critical architectural properties, quantified fundamental trade-offs, and established methodological standards for evaluating behavioral diversity alongside performance. The challenges that remain, scaling to more clusters, validating across domains, integrating with self-play, optimizing sharing policies adaptively, are well-defined research directions building on solid foundations.

The intersection of federated learning and reinforcement learning is still nascent, with many open questions about how distributed agents can learn collaboratively while maintaining individual identities. Strategic games provide an ideal testbed where diversity is measurable, valuable, and theoretically interesting. As AI systems increasingly operate in multi-agent environments requiring both cooperation and specialization, the principles uncovered in this chess framework, selective sharing, architectural constraints driving specialization, the performance-diversity trade-off, will inform designs across domains.

The ultimate measure of success for this research is not merely the specific findings about chess or federated learning, but the questions it raises and the paths it opens for future investigation. If this work inspires researchers to explore selective aggregation in other domains, develop theoretical foundations for partial parameter sharing, or design adaptive systems that dynamically balance collaboration and specialization, then it will have achieved its broader purpose.

Clustered federated learning with selective aggregation is not a complete solution to preserving diversity in distributed AI, but it is a promising direction. The framework, implementation, experimental findings, and lessons learned provide tools and insights for the next generation of researchers tackling these challenges. The game of chess taught humanity about strategy, planning, and decision-making for centuries. Perhaps it can also teach our AI systems how to collaborate while maintaining their individual strategic identities.

Acknowledgements

I would like to express my gratitude to Sanji, the greatest cook and vice-captain of the Straw Hat Pirates, whose unwavering dedication to his craft and relentless pursuit of excellence inspired me throughout this research journey. Just as he never compromises on the quality of his cuisine, I strived to maintain rigor and thoroughness in this work.

Thank you, Sanji, for teaching me that passion and perseverance can overcome any obstacle.

Bibliography

[1] Manoj Ghuhan Arivazhagan, Vinay Aggarwal, Aaditya Kumar Singh, and Sunav Choudhary. Federated learning with personalization layers. In *arXiv preprint arXiv:1912.00818*, 2019.

[2] Christopher Briggs, Zhong Fan, and Peter Andras. Federated learning with hierarchical clustering of local updates to improve training on non-iid data. *arXiv preprint arXiv:2004.11791*, 2020.

[3] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[4] Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[5] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In *International Conference on Machine Learning (ICML)*, pages 1407–1416, 2018.

[6] Alireza Fallah, Aryan Mokhtari, and Asuman Ozdaglar. Personalized federated learning with theoretical guarantees: A model-agnostic meta-learning approach. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 3557–3568, 2020.

[7] Avishek Ghosh, Jichan Chung, Dong Yin, and Kannan Ramchandran. An efficient framework for clustered federated learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pages 19586–19597, 2020.

[8] Peter Kairouz, H Brendan McMahan, Brendan Avent, Aurélien Bellet, Mehdi Bennis, Arjun Nitin Bhagoji, Kallista Bonawitz, Zachary Charles, Graham Cormode, Rachel Cummings, et al. Advances and open problems in federated learning. *Foundations and Trends in Machine Learning*, 14(1–2):1–210, 2021.

[9] Sajad Khodadadian, Pranay Sharma, Gauri Joshi, and Siva Theja Maguluri. Federated reinforcement learning: Linear speedup under markovian sampling. In *International Conference on Machine Learning (ICML)*, pages 10997–11057, 2022.

[10] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[11] Leela Chess Zero. Leela chess zero. https://lczero.org/, 2023. Accessed: 2024-12-01.

[12] Tian Li, Anit Kumar Sahu, Manzil Zaheer, Maziar Sanjabi, Ameet Talwalkar, and Virginia Smith. Federated optimization in heterogeneous networks. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 2, pages 429–450, 2020.

[13] Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. Communication-efficient learning of deep networks from decentralized data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, volume 54 of *Proceedings of Machine Learning Research*, pages 1273–1282, 2017.

[14] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning (ICML)*, pages 1928–1937, 2016.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[16] Chirag Nadiger, Anuj Kumar, and Shikha Abdelhak. Federated reinforcement learning for fast personalization. *arXiv preprint arXiv:1910.04669*, 2019.

[17] Jack Parker-Holder, Aldo Pacchiano, Krzysztof M Choromanski, and Stephen J Roberts. Effective diversity in population based reinforcement learning. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:18050–18062, 2020.

[18] Justin K Pugh, Lisa B Soros, and Kenneth O Stanley. Quality diversity: A new frontier for evolutionary computation. *Frontiers in Robotics and AI*, 3:40, 2016.

[19] Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. In *arXiv preprint arXiv:1606.04671*, 2016.

[20] Felix Sattler, Klaus-Robert Müller, and Wojciech Samek. Clustered federated learning: Model-agnostic distributed learning and optimization under privacy and communication constraints. *IEEE Transactions on Neural Networks and Learning Systems*, 32(8):3710–3722, 2020.

[21] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362:1140–1144, 2018.

[22] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550:354–359, 2017.

[23] Stockfish Developers. Stockfish: A strong open source chess engine. https://stockfishchess.org/, 2023. Accessed: 2024-12-01.

[24] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2nd edition, 2018.

[25] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. volume 10, pages 1633–1685, 2009.

[26] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.