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.

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"

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.

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.

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

These AI-assisted searches were particularly useful for:

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. 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). 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) 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 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 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) 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. 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. 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. 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:

Non-IID Data: In typical federated settings, data across clients is not independently and identically distributed. 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. 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. 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, 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 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, 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.

2.4.2 Neural Network Chess Engines

The introduction of AlphaZero 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), 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) introduced asynchronous updates from multiple workers to a shared model. IMPALA (Importance Weighted Actor-Learner Architecture) separates experience collection from learning, using importance sampling to handle the resulting off-policy data.

Distributed AlphaZero: The original AlphaZero training 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. 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 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 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 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 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 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 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, 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 and experimental design that will validate our approach.

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), 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 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, en-

suring 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 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.

3.1.4 Performance Metrics

Evaluating our framework requires metrics that capture both playing strength and strategic diversity. We organize our metrics into three categories: performance metrics, model-level divergence metrics, and behavioral metrics.

For playing strength evaluation, we employ ELO rating estimation through systematic matches against the Stockfish chess engine at multiple difficulty levels. Each cluster model is evaluated independently, producing separate ELO estimates ELO_k for each cluster k, along with confidence intervals based on rating deviation. This allows us to assess whether selective layer sharing improves playing strength compared to fully independent training or complete parameter sharing, and whether both clusters maintain competitive performance despite their specialization.

Strategic diversity at the model level is quantified through cluster divergence metrics that examine the internal representations learned by different cluster models. For each layer ℓ in the network, we compute the cosine similarity between corresponding weight tensors θ_k^ℓ and θ_j^ℓ from clusters k and j, defined as $\cos(\theta_k^\ell, \theta_j^\ell) = \frac{\theta_k^\ell \cdot \theta_j^\ell}{\|\theta_k^\ell\|_2 \|\theta_j^\ell\|_2}$. We also compute the normalized L2 distance $d_\ell(\theta_k, \theta_j) = \frac{\|\theta_k^\ell - \theta_j^\ell\|_2}{\sqrt{\|\theta_k^\ell\|_2^2 + \|\theta_j^\ell\|_2^2}}$ to capture magnitude differences. These layer-wise metrics are aggregated into group-level divergence scores for the input block, early residual layers, middle residual layers, late residual layers, policy head, and value head. This hierarchical analysis reveals which network components remain shared across clusters and which become specialized.

Weight statistics complement divergence metrics by tracking the evolution of model parameters during training. For each cluster and layer group, we monitor the mean, standard deviation, minimum, and maximum weight values, as well as the proportion of weights near zero. This helps identify potential training issues such as vanishing gradients, dead neurons, or unstable optimization. We also track the magnitude of weight changes between consecutive rounds, quantifying the learning rate and convergence behavior at different network depths.

Behavioral diversity is assessed through playstyle evaluation and move type distribution analysis. The playstyle score $\psi(M)$ for a model M combines multiple chess-specific metrics derived from self-play games. Following the methodology of Novachess.ai, we analyze attacked material, legal move counts, material captured, and center control during the critical middle-game phase. These metrics are normalized and combined through a weighted average to produce a tactical score ranging from 0 (very positional) to 1 (very tactical), with intermediate values indicating balanced play. The classification provides both a continuous score and discrete cat-

egories ranging from very positional through balanced to very tactical.

Move type distribution metrics provide a complementary behavioral perspective by classifying each move in self-play games into categories: captures, checks, pawn advances, piece development, castling, quiet moves, and aggressive moves (captures plus checks). For each cluster, we compute the percentage of moves falling into each category, averaged across multiple games. Let $P_k(m)$ denote the empirical probability that cluster k plays move type m. The difference in move type distributions between clusters, measured as $\Delta_{\text{aggressive}} = P_{\text{tactical}}(\text{captures}) + P_{\text{tactical}}(\text{checks}) - P_{\text{positional}}(\text{captures}) - P_{\text{positional}}(\text{checks})$, quantifies whether tactical clusters genuinely exhibit more forcing, aggressive play compared to positional clusters.

These metrics collectively enable us to assess whether our selective aggregation strategy successfully balances the competing objectives of performance optimization through knowledge sharing and strategic diversity preservation through cluster-specific specialization. The empirical validation of our approach requires demonstrating that intermediate sharing strategies achieve superior performance and diversity compared to the baseline extremes of complete sharing or complete independence, while maintaining measurable divergence in both model parameters and behavioral characteristics.

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 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, 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.

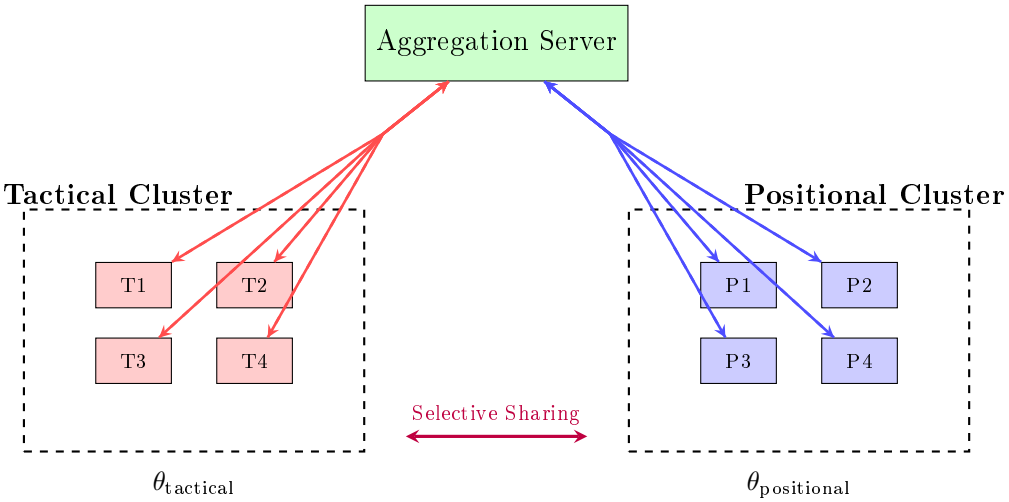

Figure 3.1: System architecture showing the central aggregation server coordinating two clusters of distributed clients. Tactical cluster clients (red) and positional cluster clients (blue) upload model updates to the server and download cluster-specific models. Selective inter-cluster aggregation enables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

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.

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 empha-

size 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 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 $8 \times 8 \times 119$, 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 $\times$ 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 queenside. 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 normaliza-

tion 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 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 $\times$ 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.

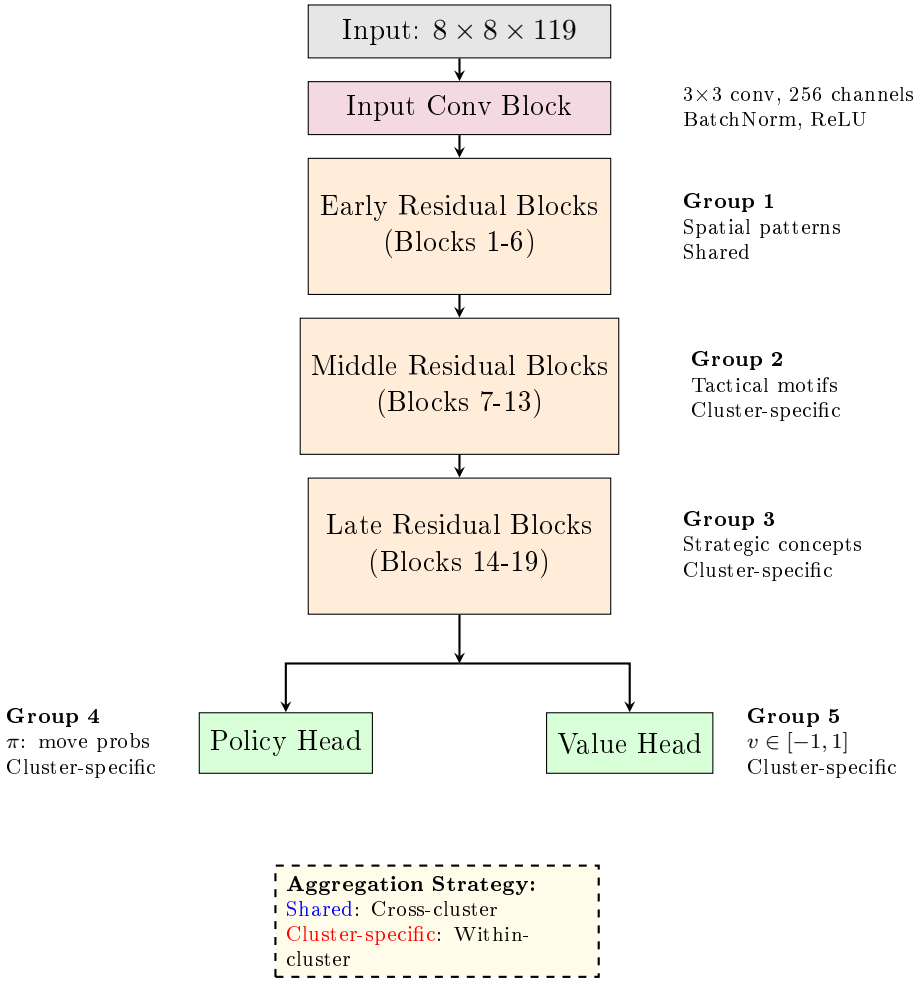

Figure 3.2: Neural network architecture showing the input layer, residual tower with 19 blocks grouped into early, middle, and late stages, and dual output heads for policy and value prediction. Annotations indicate the five layer groups used for selective aggregation, with example aggregation strategies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

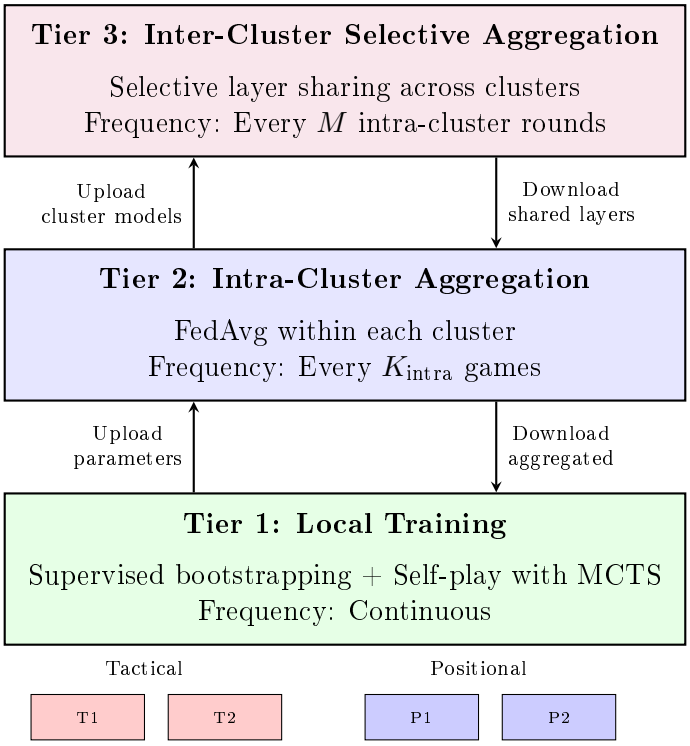

Figure 3.3: Three-tier hierarchical aggregation system showing the flow of information from local client training through intra-cluster aggregation to inter-cluster selective sharing. Arrows indicate bidirectional communication between tiers, with clients uploading parameters and downloading aggregated 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 aggre-

gation 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 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 configura-

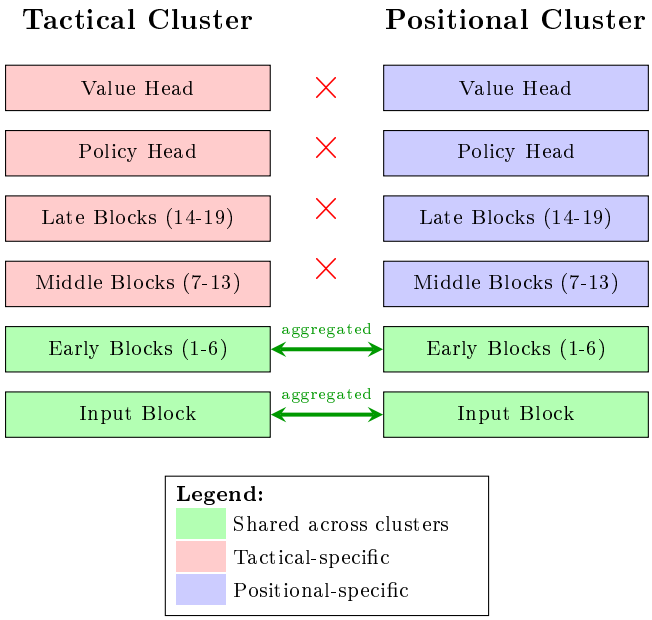

Figure 3.4: Selective layer sharing visualization showing how different layer groups are treated during inter-cluster aggregation in the baseline configuration (B1). Green layers (input block and early residual blocks) are aggregated across clusters, receiving cross-cluster knowledge transfer. Red and 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.

tions. 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 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 $\mathcal{L}_{\text{specific}} = \{\text{middle residual blocks}, \text{late residual blocks}, \text{policy head}, \text{value head}\}$.

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:

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

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:

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

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 se-

lection 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..

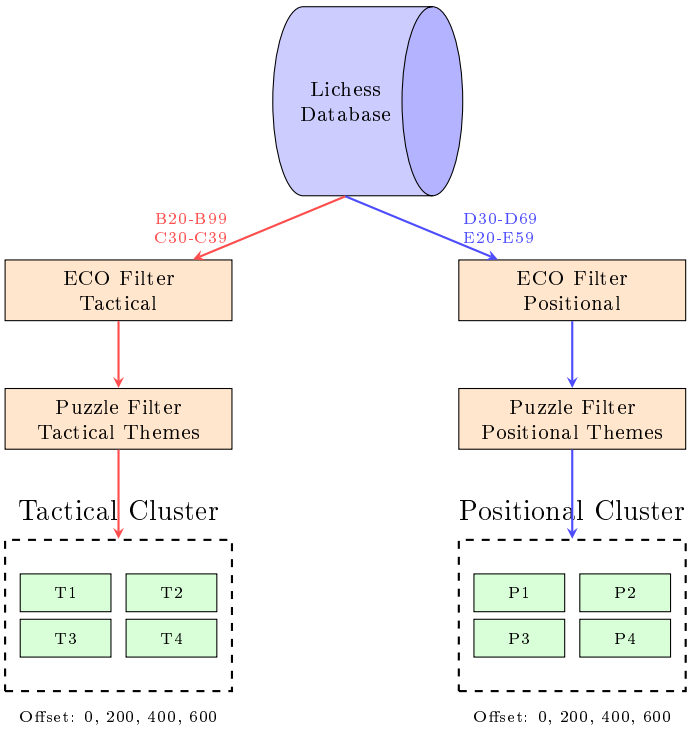

Figure 3.6: Playstyle-aware data filtering pipeline showing dual pathways from the Lichess database to cluster-specific clients. Games are filtered by ECO opening codes (tactical openings like Sicilian Dragon vs positional openings like Queen's Gambit), combined with puzzle theme filtering (tactical 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 com-

pensation), 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 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.

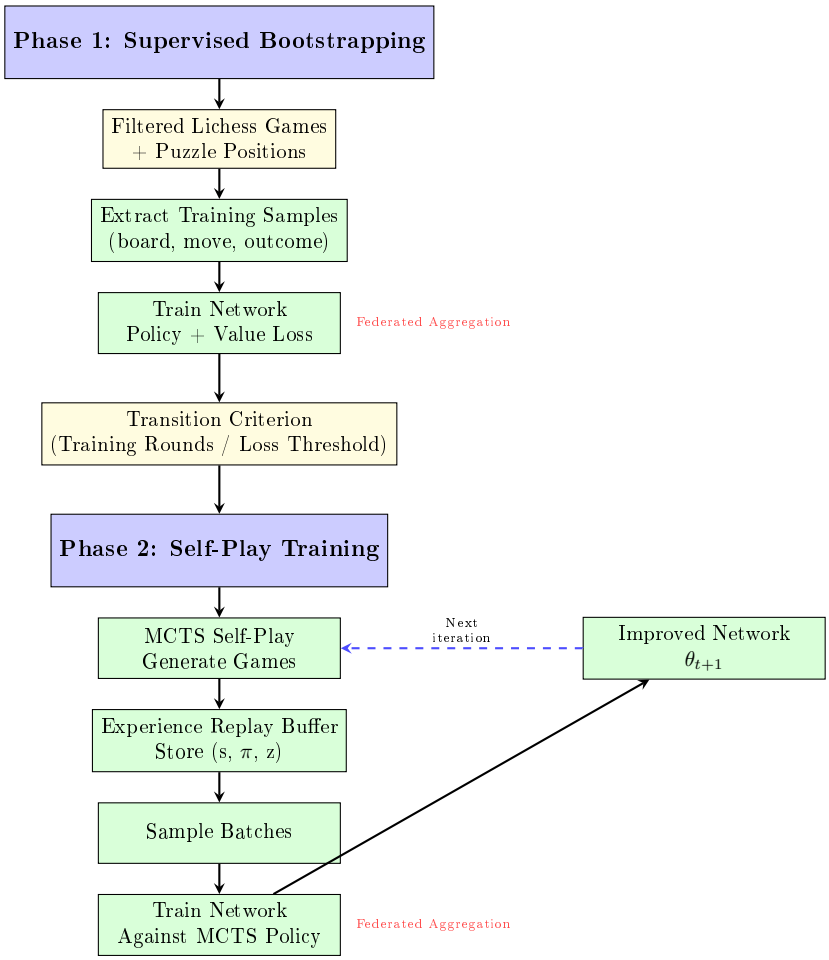

Figure 3.7: Training pipeline flowchart showing the transition from supervised bootstrapping to self-play reinforcement learning. The supervised phase trains on filtered human games and puzzles, while the self-play phase uses MCTS to generate training data. Both phases incorporate federated aggregation 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, 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 3.8) 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 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 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.

3.8 Evaluation Methodology

Our evaluation framework measures three distinct aspects of the federated learning system: playing strength, playstyle preservation, and cluster divergence. These metrics allow us to assess whether selective aggregation achieves the dual objectives of maintaining competitive playing ability while preserving distinct tactical and positional characteristics. Evaluations are conducted periodically during training (every 10 rounds by default) to track the evolution of both strength and style across different aggregation configurations. Each evaluation generates approximately 30 games per cluster through matches against calibrated opponents, with each game analyzed in depth to extract over 40 distinct metrics spanning position evaluation, move classification, opening diversity, and strategic patterns.

3.8.1 Playing Strength Evaluation

Playing strength is quantified through ELO rating estimation based on match results against calibrated Stockfish opponents at multiple difficulty levels. This approach provides an objective, standardized measure of chess ability that can be compared across different training configurations and aggregation strategies.

For each cluster, we play a series of evaluation matches against Stockfish engines configured to operate at specific ELO ratings. The default evaluation protocol uses three opponent strengths: 1000 ELO (novice level), 1200 ELO (beginner level), and 1400 ELO (intermediate level). These ratings span the range where we expect our models to perform during early to mid training, providing meaningful win/loss signals without overwhelming the model with opponents far beyond its current strength. At each ELO level, we play 10 games with alternating colors: 5 games where the AI plays white (receiving first-move advantage) and 5 games where the AI plays black. This yields 30 total evaluation games per cluster per evaluation round, providing sufficient statistical power to estimate playing strength while remaining computationally feasible.

Stockfish engine configuration varies based on the target ELO rating to accurately simulate human players at different skill levels. For target ratings below 1320 ELO, we use Stockfish's skill level setting (ranging from 0 to 20) combined with search depth limiting. The skill level is computed as $\max(0, \min(20, \lfloor (R_{\text{target}} - 800)/100 \rfloor))$, and depth is limited to 1 ply for ratings below 900, 2 ply for ratings 900-1099, and 3 ply for ratings 1100-1319. For target ratings at or above 1320 ELO, we enable Stockfish's UCI strength limiting mode and directly specify the target ELO, allowing the engine to self-regulate its playing strength through controlled evaluation function approximations and occasional suboptimal move selection. All evaluation games use a time control of 0.1 seconds (100 milliseconds) per move for the AI model, providing sufficient time for neural network inference and move selection while enabling rapid evaluation. Stockfish operates under its configured depth or skill limitations rather than explicit time controls.

Match results are recorded as wins (AI achieves checkmate or opponent resigns), draws (stalemate, insufficient material, threefold repetition, or fifty-move rule), and losses (AI is checkmated or resigns). Each game outcome is converted to a score using the standard ELO convention: 1 point for a win, 0.5 points for a draw, and 0 points for a loss. For each opponent level, we compute win rate, draw rate, and loss rate as percentages of the 10 games played at that level, providing insight into the AI's performance profile against opponents of different strengths.

ELO estimation uses an iterative approximation algorithm that finds the single ELO rating that best explains the observed match results across all opponent lev-

els. For a candidate ELO rating R_{test} and an opponent with known rating R_{opp}, the expected score (probability of winning plus half the probability of drawing) is computed using the standard ELO formula:

$$E(R_{\text{test}}, R_{\text{opp}}) = \frac{1}{1 + 10^{(R_{\text{opp}} - R_{\text{test}})/400}} \tag{3.16}$$

This formula reflects the principle that a 400-point ELO difference corresponds to an expected score of approximately 0.91 (91% win expectation for the higher-rated player), while equal ratings yield an expected score of 0.5 (50% win expectation).

We test candidate ratings from 800 to 2400 ELO in increments of 25 ELO points, spanning the full range of plausible ratings for our models. For each candidate rating R_{test}, we compute the total squared error between the expected scores (based on the ELO formula) and the actual scores (observed from match results) across all n opponent levels:

$$\text{Error}(R_{\text{test}}) = \sum_{i=1}^{n} \left(E(R_{\text{test}}, R_{\text{opp},i}) - \frac{S_i}{G_i} \right)^2 \tag{3.17}$$

where S_i is the total score achieved against opponent level i (wins plus 0.5 times draws) and G_i is the number of games played against that opponent (10 in our default configuration). The estimated ELO $\hat{R}$ is the candidate rating that minimizes this squared error:

$$\hat{R} = \underset{R_{\text{test}} \in \{800, 825, 850, \ldots, 2400\}}{\arg\min} \text{Error}(R_{\text{test}}) \tag{3.18}$$

This approach finds the rating that provides the best overall fit to the observed performance across all opponent strengths, accounting for the AI's entire performance profile rather than results against a single opponent.

To quantify uncertainty in the ELO estimate, we compute a confidence range that decreases as more evaluation games are played. The confidence interval width (in ELO points) is given by:

$$\Delta R = \max(50, 400 - 10 \cdot G_{\text{total}}) \tag{3.19}$$

where $G_{\text{total}} = \sum_{i=1}^{n} G_i$ is the total number of evaluation games across all opponent levels. With our default of 30 evaluation games (3 opponents $\times$ 10 games each), the confidence range is ± 100 ELO. This reflects the principle that ELO estimates from limited game samples have inherent uncertainty due to the stochastic nature of game outcomes and the discrete sampling of opponent strengths. The confidence interval narrows to a minimum of ± 50 ELO as the number of evaluation games increases, acknowledging that even with many games, there remains some irreducible

uncertainty in rating estimation. When comparing configurations, we consider ELO estimates statistically distinguishable if their confidence intervals do not overlap.

3.8.2 Playstyle Metrics

Playstyle characterization quantifies the tactical versus positional nature of each cluster's play through comprehensive analysis of self-play games and evaluation games. We extract over 30 distinct metrics from each game, covering position evaluation, move selection patterns, pawn structure, center control, opening diversity, and critical decision-making. These metrics are aggregated across all analyzed games to produce a cluster-level playstyle profile.

Tactical Score Computation

The tactical score is a normalized composite metric that integrates multiple features indicative of tactical or positional play. It ranges from 0.0 (purely positional) to 1.0 (purely tactical), enabling quantitative comparison of playing styles across clusters and configurations.

Position analysis focuses on the middlegame phase where strategic differences are most pronounced. We analyze positions from move 6 through move 25 (plies 12-50), excluding the opening phase where moves are often book knowledge and the endgame phase where limited material constrains tactical opportunities. Positions where either player is in check are optionally skipped to avoid distortions from forced tactical sequences, as check positions may inflate tactical metrics due to the reduced set of legal responses.

Three normalized component metrics contribute to the tactical score, each capturing a distinct aspect of tactical versus positional play:

Attacks Metric: This measures the total material value of opponent pieces under attack, reflecting the degree to which the player targets enemy forces with aggressive piece placement. For each analyzed position, we identify all opponent pieces that are attacked (i.e., can be legally captured by at least one of the player's pieces) and sum their material values using standard chess valuations: pawns = 1 point, knights = 3 points, bishops = 3 points, rooks = 5 points, queens = 9 points. The king is not included in attacked material calculations as it cannot be captured. The attacked material sum is averaged across all analyzed positions in the game, then normalized by dividing by 39 points (the maximum possible attacked material: one queen, two rooks, two bishops, two knights, assuming all pawns have been promoted or captured). The attacks metric is thus:

$$\text{AttacksMetric} = \frac{1}{N}\sum_{t=1}^{N}\frac{\sum_{p\in\text{attacked}(t)}\text{value}(p)}{39} \tag{3.20}$$

where N is the number of analyzed positions, attacked(t) is the set of opponent pieces under attack at position t, and value(p) is the material value of piece p.

Moves Metric: This measures the average number of legal moves available across analyzed positions, capturing the mobility and activity level of the position. Tactical positions typically feature active piece placement with multiple attacking options, yielding high legal move counts (often 35-45 legal moves). Positional positions may have fewer immediately forcing options, with pieces coordinating for long-term advantage rather than immediate threats. We count the number of legal moves at each analyzed position, average across all positions, and normalize by dividing by 40 (a typical middlegame move count for active positions), capping the result at 1.0 to prevent positions with exceptionally high mobility from dominating the metric:

$$\text{MovesMetric} = \min\left(1.0, \frac{1}{N}\sum_{t=1}^{N}\frac{|\text{LegalMoves}(t)|}{40}\right) \tag{3.21}$$

where $|\text{LegalMoves}(t)|$ is the count of legal moves at position t.

Material Metric: This measures the total material captured during the game up through move 25 (ply 50), reflecting the propensity for tactical exchanges and piece sacrifices. Tactical players engage in frequent exchanges to create threats and simplify positions with concrete advantages, while positional players may avoid early exchanges to maintain strategic complexity and piece coordination. We sum the material value of all pieces captured from move 1 through move 25 (covering the opening through early middlegame), normalize by dividing by 20 points (representing significant material exchange, roughly two rooks or a queen plus minor pieces), and cap at 1.0:

$$\text{MaterialMetric} = \min\left(1.0, \frac{\sum_{\text{move}=1}^{25}\text{CapturedMaterial}(\text{move})}{20}\right) \tag{3.22}$$

The three component metrics are combined into a single tactical score through weighted averaging. If material was captured during the analyzed portion of the game (MaterialMetric > 0), all three metrics contribute equally. If no material was captured (rare, but possible in highly closed positions or very short games), only the attacks and moves metrics are averaged, as the absence of captures does not necessarily indicate non-tactical play (e.g., a game could feature strong attacks that

were successfully defended without exchanges):

$$\text{TacticalScore} = \begin{cases} \frac{\text{AttacksMetric}+\text{MovesMetric}+\text{MaterialMetric}}{3} & \text{if MaterialMetric} > 0 \\ \frac{\text{AttacksMetric}+\text{MovesMetric}}{2} & \text{otherwise} \end{cases} \tag{3.23}$$

The tactical score is rounded to three decimal places and classified into one of five discrete categories based on threshold ranges: Very Tactical (TacticalScore > 0.70), Tactical ($0.65 \leq$ TacticalScore ≤ 0.70), Balanced ($0.60 \leq$ TacticalScore < 0.65), Positional ($0.50 \leq$ TacticalScore < 0.60), and Very Positional (TacticalScore < 0.50). These thresholds were established through empirical analysis of human games from the Lichess database, with tactical openings (e.g., Sicilian Dragon, King's Gambit) typically scoring above 0.65 and positional openings (e.g., Queen's Gambit Declined, Nimzo-Indian) typically scoring below 0.60.

For each cluster, we report the mean tactical score across all analyzed games, the standard deviation (indicating consistency of playstyle), the minimum and maximum scores (indicating range), and the distribution of games across the five classification categories. A cluster with strong tactical characteristics should exhibit a mean tactical score above 0.65 with most games classified as Tactical or Very Tactical, while a positional cluster should score below 0.60 with most games classified as Positional or Very Positional.

Move Type Classification and Distribution

Beyond the aggregate tactical score, we perform detailed move-level classification to quantify specific move selection patterns. Each move in each analyzed game is classified into multiple potentially overlapping categories based on its chess properties. The primary move type categories are:

Captures: Moves that remove an opponent piece from the board (board.is_capture(move) returns true). Includes both equal exchanges (trading pieces of equal value), favorable captures (winning material), and sacrifices (capturing less valuable pieces while offering more valuable pieces for recapture). Capture rate is a strong indicator of tactical style, as tactical players actively seek opportunities to win material or create tactical complications through forcing exchanges.

Checks: Moves that place the opponent's king in check, requiring an immediate defensive response. Checks are forcing moves that limit the opponent's options and often initiate tactical sequences. Frequent checking can indicate aggressive, tactical play, though excessive checking without purpose may be inefficient.

Pawn Advances: Pawn moves that are not captures (i.e., pushing pawns for-

ward rather than capturing diagonally). Pawn advances can serve tactical purposes (advancing passed pawns, opening lines for pieces, restricting opponent piece mobility) or positional purposes (controlling space, preparing piece maneuvers, establishing pawn chains). The context and timing of pawn advances distinguish tactical from positional usage.

Piece Development: Moves that develop knights or bishops (moving them from their starting squares to active squares) during the opening phase (plies ≤ 20, corresponding to the first 10 moves). Efficient piece development is a fundamental principle in the opening, establishing piece activity and preparing for middlegame operations. Development rate can indicate opening knowledge and adherence to chess principles.

Castling: Moves that execute kingside (O-O) or queenside (O-O-O) castling, simultaneously moving the king to safety and activating a rook. Castling is typically performed in the opening or early middlegame. The timing and frequency of castling can indicate opening style and risk tolerance.

Quiet Moves: Moves that are neither captures nor checks. Quiet moves include piece repositioning, pawn advances, and preparatory moves that improve position without immediate forcing tactics. A high quiet move percentage may indicate positional play focused on gradual improvement, though quiet moves are also present in tactical sequences (e.g., quiet intermediate moves in combinations).

Aggressive Moves: The union of captures and checks, representing moves that directly threaten opponent material or king safety. Aggressive move percentage is a key discriminator between tactical and positional styles, as tactical players consistently create threats while positional players may prioritize long-term advantages.

For each move type category, we compute three statistics across all analyzed games for each cluster: total count (the number of moves of that type across all games), percentage (the proportion of all moves that fall into that category), and average per game (the mean number of moves of that type in each game). These statistics enable both absolute comparison (e.g., tactical cluster plays 150 total captures vs. positional cluster plays 100) and relative comparison (e.g., captures constitute 25% of tactical cluster moves vs. 20% of positional cluster moves).

Positional Structure Metrics

Several metrics quantify aspects of position structure and piece coordination that characterize positional understanding:

Center Control: The central four squares (d4, d5, e4, e5) are the most important in chess, controlling these squares provides piece mobility, attack lines, and spatial advantage. We measure center control by counting the number of pieces (excluding pawns occupying the center, which are counted separately) that attack

each central square. For each analyzed position, we compute:

$$\text{CenterControl}_{\text{White}} = \sum_{sq \in \{d4,d5,e4,e5\}} |\{p : p \in \text{WhitePieces}, p \text{ attacks } sq\}| \quad (3.24)$$

and similarly for Black. We average center control across all analyzed positions and report separate values for each color when the AI is analyzed. High center control indicates active piece placement and adherence to strategic principles regarding central domination.

Pawn Structure Metrics: Pawns form the strategic skeleton of the position, and pawn structure weaknesses often determine long-term evaluation. We track three pawn structure metrics:

Average Pawn Rank: The mean rank (1-8 from White's perspective, 8-1 from Black's) of all pawns for the analyzed player. Higher average pawn rank indicates advanced pawns, which can be either aggressive (in tactical play, pushing passed pawns) or strategic (in positional play, controlling space).

$$\text{AvgPawnRank} = \frac{1}{|\text{Pawns}|} \sum_{p \in \text{Pawns}} \text{Rank}(p) \quad (3.25)$$

Isolated Pawns: Pawns with no friendly pawns on adjacent files. Isolated pawns are often weak as they cannot be defended by other pawns and may become targets. Positional players typically avoid creating isolated pawns unless they provide compensation (e.g., open files for rooks). We count isolated pawns at each analyzed position and report the average:

$$\text{IsolatedPawns}(t) = |\{p \in \text{Pawns}_t : \text{File}(p{-}1) \text{ and File}(p{+}1) \text{ have no friendly pawns}\}| \quad (3.26)$$

Doubled Pawns: Multiple pawns on the same file, created when a pawn captures and another pawn advances to the same file. Doubled pawns are generally considered weak as they reduce pawn mobility and create targets. We count doubled pawns by identifying files with two or more pawns:

$$\text{DoubledPawns}(t) = \sum_{f \in \{a,b,c,d,e,f,g,h\}} \max(0, |\text{Pawns}_t \text{ on file } f| - 1) \quad (3.27)$$

Move Diversity: We track the number of unique destination squares used across all moves in a game. High move diversity indicates flexible piece placement and exploration of multiple strategic plans, while low diversity may indicate repeti-

tive maneuvering or limited piece activity. Move diversity ratio is computed as:

$$\text{MoveDiversityRatio} = \frac{\text{UniqueDestinationSquares}}{\text{TotalMoves}/2} \tag{3.28}$$

normalized by the number of ply-pairs to account for game length.

Delta Analysis: Tipping Point Metric

The delta metric quantifies the criticality of positions encountered, measuring how much evaluation changes based on move choice. Positions with large delta values represent critical tipping points where the correct move maintains or increases advantage while alternative moves lead to significant disadvantage. Positions with small delta values have multiple reasonable continuations with similar evaluations, reflecting strategic flexibility.

Delta analysis uses the Stockfish chess engine configured at depth 12 (12-ply search) with multi-PV mode set to 2, requesting the engine to report evaluations for the top two moves. For each analyzed position t, Stockfish provides: - The best move m_1 with evaluation score s_1 (in centipawns, where 100 centipawns = 1 pawn) - The second-best move m_2 with evaluation score s_2

The delta is defined as the absolute difference between these evaluations, converted to pawn units:

$$\delta(t) = \frac{|s_1 - s_2|}{100} \tag{3.29}$$

We sample positions for delta analysis rather than analyzing every position due to computational cost. Sampling focuses on the middlegame phase (plies 15-40) where tactical and positional decisions are most consequential. The default sampling rate is every third position (sample every position t where $t \bmod 3 = 0$ and $15 \leq t \leq 40$), balancing statistical coverage against evaluation time. For each cluster, we report: - Average delta $\bar{\delta} = \frac{1}{N_{\text{sampled}}} \sum_t \delta(t)$ - Maximum delta $\max_t \delta(t)$ (the most critical single position) - Minimum delta $\min_t \delta(t)$ (the least critical position) - Number of positions sampled N_{sampled}

Tactical clusters are expected to show higher average delta values, as tactical positions often feature forcing sequences where the correct move wins material or achieves checkmate while alternatives lose. Positional clusters may show lower delta values, as positional play involves gradual maneuvering where multiple plans have similar evaluations.

Opening Diversity and Classification

Opening diversity measures the variety of opening systems played, indicating whether the cluster has developed a narrow repertoire (playing the same openings repeatedly) or a broad repertoire (exploring multiple opening systems). We classify openings using the Encyclopedia of Chess Openings (ECO) code system, which assigns alphanumeric codes (A00-E99) to opening variations based on the initial move sequence.

For each game, we extract the opening moves (typically the first 3-5 moves by each player) and map the move sequence to its corresponding ECO code and opening name using a database of ECO classifications. We maintain a frequency count of how many times each opening appears in the cluster's games:

$$\text{OpeningFrequency}[ECO_i] = \text{count of games with opening } ECO_i \tag{3.30}$$

From the frequency distribution, we identify the top 5-10 most frequently played openings and report their ECO codes, names, and occurrence counts. High concentration on a few openings suggests specialized opening preparation (common in focused training), while broad distribution across many openings suggests diverse strategic exploration.

Game Phase-Specific Analysis

We divide games into three phases based on move number and analyze legal move counts separately for each phase: - Opening: Plies 1-12 (moves 1-6) - Middlegame: Plies 13-40 (moves 7-20) - Endgame: Plies 41+ (moves 21+)

For each phase, we compute the average number of legal moves available across all positions in that phase. Opening positions typically have moderate move counts (20-30) as piece development proceeds. Middlegame positions often have the highest move counts (35-50) with active pieces and complex tactics. Endgame positions have fewer moves (15-30) due to reduced material. Differences in phase-specific move counts can indicate stylistic tendencies: tactical players may have higher middlegame move counts due to active piece placement, while positional players may maintain mobility across all phases through piece coordination.

3.8.3 Cluster Divergence Metrics

Cluster divergence quantifies the degree to which tactical and positional clusters have developed distinct internal representations, validating that selective aggregation successfully preserves playstyle-specific features. We measure divergence at

both the parameter level (comparing neural network weights) and the behavioral level (comparing playstyle metrics).

Parameter-Level Divergence

Parameter-level divergence compares the neural network weights between the tactical and positional cluster models on a layer-by-layer basis. For each layer or group of layers, we extract the weight tensors W_A (tactical cluster) and W_B (positional cluster) and compute three complementary distance metrics.

Cosine Similarity measures the angle between weight vectors, capturing directional alignment regardless of magnitude. Weight tensors are flattened into vectors and their dot product is normalized:

$$\text{CosineSimilarity}(W_A, W_B) = \frac{W_A \cdot W_B}{\|W_A\|_2 \|W_B\|_2} = \frac{\sum_i w_{A,i} w_{B,i}}{\sqrt{\sum_i w_{A,i}^2}\sqrt{\sum_i w_{B,i}^2}} \tag{3.31}$$

Cosine similarity ranges from -1 (vectors point in opposite directions) through 0 (orthogonal vectors) to $+1$ (identical directions). For neural network weights, negative values are rare; values close to $+1$ indicate that both clusters have learned similar weight patterns (suggesting shared knowledge), while values significantly below $+1$ indicate divergent weight patterns (suggesting cluster-specific specialization).

L2 Distance measures the Euclidean distance between weight tensors, capturing both directional and magnitude differences:

$$\text{L2Distance}(W_A, W_B) = \|W_A - W_B\|_2 = \sqrt{\sum_i (w_{A,i} - w_{B,i})^2} \tag{3.32}$$

To enable comparison across layers of different sizes, we normalize the L2 distance by the combined norms of the weight tensors:

$$\text{L2Distance}_{\text{normalized}}(W_A, W_B) = \frac{\|W_A - W_B\|_2}{\|W_A\|_2 + \|W_B\|_2} \tag{3.33}$$

This normalization ensures that large layers (e.g., convolutional layers with millions of parameters) and small layers (e.g., fully connected layers with thousands of parameters) contribute comparably to divergence assessment.

Divergence Index combines cosine similarity and normalized L2 distance into a single metric that increases monotonically as weights become more different:

$$\text{Divergence}(W_A, W_B) = (1 - \text{CosineSimilarity}(W_A, W_B)) \times (1 + \text{L2Distance}_{\text{normalized}}(W_A, W_B)) \tag{3.34}$$

The divergence index is 0 when $W_A = W_B$ (cosine similarity = 1, L2 distance = 0) and increases as weights diverge. The multiplicative combination ensures that both directional differences (captured by $1 - \text{CosineSimilarity}$) and magnitude differences (captured by L2 distance) contribute to the overall divergence score.

We compute these three metrics for each layer in the neural network and aggregate results by layer group as defined in Section 3.3: input block (input convolutional and batch normalization layers), early residual blocks (residual layers 0-5), middle residual blocks (residual layers 6-12), late residual blocks (residual layers 13-18), policy head, and value head. By comparing divergence across layer groups, we can identify where in the network clusters differ most. Under selective aggregation, we expect low divergence in shared layer groups (due to inter-cluster averaging) and higher divergence in cluster-specific groups (due to independent optimization).

Behavioral Divergence

Behavioral divergence measures differences in playstyle metrics between clusters, quantifying whether clusters exhibit distinct playing styles regardless of the underlying weight differences.

Playstyle Divergence is computed as the standard deviation of tactical scores across clusters:

$$\text{PlaystyleDivergence} = \sqrt{\frac{1}{C}\sum_{c=1}^{C}(\text{TacticalScore}_c - \bar{\text{TacticalScore}})^2} \tag{3.35}$$

where C is the number of clusters (2 in our tactical vs. positional setup) and $\bar{\text{TacticalScore}}$ is the mean tactical score across clusters. Higher playstyle divergence indicates that clusters have successfully developed distinct playing styles. For our two-cluster configuration with one tactical and one positional cluster, playstyle divergence simplifies to half the absolute difference in tactical scores.

ELO Spread measures the range of playing strengths across clusters:

$$\text{ELOSpread} = \max_c(\text{ELO}_c) - \min_c(\text{ELO}_c) \tag{3.36}$$

Large ELO spread may indicate that selective aggregation has created strength imbalances between clusters, with one cluster benefiting more from shared knowledge. Ideally, selective aggregation should preserve playstyle diversity (high playstyle divergence) without sacrificing strength balance (low ELO spread).

Move Type Comparison quantifies differences in move selection behavior by computing the absolute difference in each move type percentage between clusters:

$$\Delta_{\text{category}} = |\text{Pct}_{\text{tactical,category}} - \text{Pct}_{\text{positional,category}}| \tag{3.37}$$

for each move category (captures, checks, aggressive moves, quiet moves, pawn advances, castling, piece development). Large differences in capture rate (Δ_{captures}) and aggressive move rate ($\Delta_{\text{aggressive}}$) provide strong evidence of stylistic separation, validating that data filtering and selective aggregation have achieved distinct tactical versus positional characteristics. We also compare positional feature differences ($\Delta_{\text{center control}}$, $\Delta_{\text{isolated pawns}}$, etc.) and opening diversity differences ($\Delta_{\text{opening concentration}}$) to assess comprehensive divergence across all measured dimensions.

3.8.4 Statistical Analysis and Confidence

Statistical validation ensures that observed differences between aggregation configurations and between clusters are meaningful rather than artifacts of random variation, small sample sizes, or evaluation noise.

For ELO estimates, we report confidence intervals as described in the playing strength evaluation subsection. These intervals quantify the uncertainty inherent in estimating a rating from a limited game sample. When comparing two configurations, we consider their playing strengths statistically distinguishable if their confidence intervals do not overlap: $[\hat{R}_A - \Delta R_A, \hat{R}_A + \Delta R_A] \cap [\hat{R}_B - \Delta R_B, \hat{R}_B + \Delta R_B] = \emptyset$. Non-overlapping intervals provide strong evidence ($p < 0.05$ approximately) of a true strength difference. Overlapping intervals suggest the observed ELO difference may be within measurement uncertainty, requiring additional evaluation games for conclusive comparison.

For playstyle metrics, we compute descriptive statistics across the set of analyzed games: mean (central tendency), standard deviation (spread/consistency), minimum (lower bound), and maximum (upper bound). The standard deviation is particularly informative: low standard deviation indicates consistent playstyle across games (all games exhibit similar tactical scores), while high standard deviation indicates variable playstyle (some games highly tactical, others positional). For clusters intended to maintain consistent playstyles, we expect low within-cluster standard deviation combined with high between-cluster difference in means.

The distribution of tactical scores across the five classification categories (Very Tactical, Tactical, Balanced, Positional, Very Positional) provides additional insight into playstyle consistency. A cluster with strong tactical identity should show most games (>70%) classified as Tactical or Very Tactical, with few games in the Balanced or Positional categories. A mixed or transitioning cluster may show a broader distribution across categories. We report the count and percentage of games in each

category, enabling visual assessment of playstyle concentration.

Divergence metrics are evaluated relative to baseline expectations established by the two baseline configurations: full sharing (B2) and no sharing (B1). Under full sharing, all layers are averaged across clusters at every round, so we expect low divergence (cosine similarity near 1.0, small L2 distance) across all layer groups, as both clusters converge toward identical representations. Under no sharing, clusters never exchange weights, so divergence accumulates freely as each cluster independently optimizes for its training data. The maximum observed divergence under no sharing provides an upper bound on how different clusters can become. Selective aggregation configurations (P1-P4) should exhibit intermediate divergence: shared layers should have divergence close to the full-sharing baseline, while cluster-specific layers should have divergence approaching (but potentially not reaching) the no-sharing baseline.

We track all metrics across training rounds to assess convergence and stability. Training loss should decrease monotonically during the supervised phase and stabilize during self-play, indicating successful learning without overfitting or divergence. Playing strength (ELO) should increase during early training as the model acquires chess knowledge, then plateau once the model reaches the performance ceiling given the network architecture, training data, and evaluation opponent strength. Playstyle metrics should stabilize after an initial transient period (typically 20-50 rounds) once clusters have settled into consistent playing styles shaped by their filtered training data. Divergence metrics may increase initially as clusters specialize, then stabilize once distinct representations have formed and the selective aggregation policy (which layers to share) constrains further divergence.

All metrics are logged at regular intervals (every 10 training rounds by default, configurable via the evaluation interval parameter) and stored in structured JSON format for subsequent analysis. Each evaluation round generates a JSON file per cluster containing all playstyle metrics, a JSON file for model divergence metrics, and a compressed JSONL (JSON Lines) event stream recording all metrics chronologically. This comprehensive logging enables longitudinal analysis of training dynamics, comparison of different aggregation strategies throughout the learning process, and post-hoc investigation of unexpected behaviors or performance patterns.

3.9 Experimental Design

Our experimental design systematically tests the central thesis that selective layer aggregation can preserve distinct playstyles while maintaining competitive playing strength. We employ a multi-phase approach comprising baseline experiments (establishing bounds on knowledge sharing), selective aggregation experiments (testing

different layer sharing strategies), and performance evaluation experiments (validating playstyle preservation and generalization). The complete experimental matrix is presented in Figure 3.8.

	B1	B2	P1	P2	P3	P4
Playing Strength (ELO)	✓	✓	✓	✓	✓	✓
Tactical Score	✓	✓	✓	✓	✓	✓
Move Type Distribution	✓	✓	✓	✓	✓	✓
Global Divergence	×	✓	✓	✓	✓	✓
Layer-wise Divergence	×	N/A	✓	✓	✓	✓
Head Divergence Analysis	×	N/A	✓	✓	✓	✓
Temporal Divergence	×	✓	✓	✓	✓	✓
Delta/Tipping Points	✓	✓	✓	✓	✓	✓

✓ = Metric collected
× = Not applicable
N/A = No divergence

Figure 3.8: Experimental design matrix showing which metrics are collected for each experiment configuration. B1 (Full Sharing) and B2 (No Sharing) establish baseline bounds, while P1-P4 test selective aggregation strategies. All configurations measure playing strength and playstyle characteristics; divergence metrics are only meaningful where clusters can diverge.

3.9.1 Baseline Experiments

The baseline experiments establish the performance bounds for our system, demonstrating what happens under extreme aggregation policies: full knowledge sharing (all layers aggregated) versus zero knowledge sharing (independent cluster training). These baselines provide reference points for evaluating selective aggregation strategies.

B1: Full Sharing Baseline

The full sharing baseline (B1) implements standard federated averaging across all network layers, providing a control configuration that demonstrates the effects of complete knowledge transfer without any selective aggregation. In this configuration, both tactical and positional clusters participate in global aggregation at every round, with all layer groups (input block, all 19 residual blocks, policy head, value head) averaged across clusters using the FedAvg algorithm.

This baseline tests the hypothesis that standard federated learning, while effective at leveraging collective training data, destroys cluster-specific patterns that

encode distinct playstyles. We expect clusters to converge toward identical representations as all parameters are continuously averaged, eliminating any divergence that might otherwise develop through playstyle-specific training data. The tactical and positional clusters should exhibit minimal playstyle separation, with tactical scores converging to approximately 0.5-0.6 (balanced play) regardless of the training data filtering.

Training configuration for B1 uses 200 rounds with 4 clients per cluster, each processing 400 games per round from their respective playstyle-filtered datasets. Despite the data filtering (tactical cluster trains on tactical puzzles, positional cluster trains on endgame/positional puzzles), the continuous averaging of all network parameters should prevent clusters from developing and maintaining distinct internal representations corresponding to their training data characteristics.

Expected outcomes for B1 include near-zero global divergence (cosine similarity exceeding 0.99 across all layers), minimal playstyle separation between clusters (tactical score difference less than 0.1), and moderate ELO performance (approximately 1300-1500) reflecting the benefits of aggregating knowledge from both clusters while potentially suffering from conflicting gradient updates due to the different training objectives.

B2: No Sharing Baseline

The no sharing baseline (B2) represents the opposite extreme: complete independence between clusters with no inter-cluster communication. Each cluster performs intra-cluster aggregation using FedAvg to combine knowledge from its 4 constituent clients, but clusters never exchange parameters with each other. This configuration establishes the maximum possible divergence achievable when clusters independently optimize for their respective playstyle-specific training data.

This baseline tests whether independent training on playstyle-filtered data is sufficient to produce measurable and consistent playstyle differences, and quantifies the performance cost (if any) of forgoing knowledge transfer between clusters. We expect B2 to show the highest divergence across all layer groups, with clusters developing completely independent representations optimized for their respective tactical or positional training objectives.

The training configuration for B2 mirrors B1 (200 rounds, 4 clients per cluster, 400 games per round), but aggregation is strictly limited to within-cluster averaging. The tactical cluster aggregates only among tactical clients, and the positional cluster aggregates only among positional clients. This effectively divides the system into two independent federated learning deployments operating on different training data distributions.

Expected outcomes for B2 include maximum global divergence (cosine similarity

potentially dropping below 0.7 for policy and value heads), strong playstyle separation (tactical scores diverging by 0.20 or more between clusters with tactical cluster exceeding 0.65 and positional cluster below 0.50), and potentially lower individual cluster ELO (approximately 1200-1400) due to the reduced effective training data compared to configurations that share knowledge. The B2 baseline demonstrates the upper bound on playstyle preservation at the potential cost of playing strength.

3.9.2 Selective Aggregation Experiments

The selective aggregation experiments (P1-P4) constitute the core empirical contribution of this work, testing different hypotheses about where in the neural network architecture playstyle-specific versus general chess knowledge resides. Each configuration selectively shares different layer groups while keeping others cluster-specific, enabling us to identify which layers benefit from sharing (universal chess patterns) and which should remain independent (playstyle-specific representations).

P1: Share Early Layers Only

Configuration P1 tests the hypothesis that early network layers learn general chess features (piece recognition, basic board patterns, elementary tactical motifs) that are universal across playstyles and thus benefit from sharing, while late layers and output heads encode playstyle-specific strategic preferences that should diverge.

In P1, the shared layer groups comprise the input block (input convolution and batch normalization) and early residual blocks (residual layers 0-5), totaling approximately 2.5 million parameters representing roughly 11% of the network. These layers process raw board representations and extract low-level features such as piece positions, material balance, and simple tactical patterns (attacks, defenses, pins).

The cluster-specific layer groups include middle residual blocks (layers 6-12), late residual blocks (layers 13-18), policy head, and value head, representing approximately 8.5 million parameters (38% of the network). These layers synthesize low-level features into strategic understanding and decision-making, where we hypothesize that tactical versus positional distinctions emerge.

Expected outcomes for P1 include shared early layers showing near-zero divergence (cosine similarity above 0.98) while cluster-specific layers exhibit increasing divergence with network depth, potentially reaching divergence indices of 0.5-0.8 in the policy head. We expect moderate playstyle separation (tactical score difference of 0.15-0.20) and good playing strength (ELO 1400-1600) as clusters benefit from shared foundational features while specializing their decision-making.

This configuration directly tests hypothesis H3 (early layers remain shared while late layers diverge) by explicitly enforcing this structure through the aggregation

policy. Analysis of layer-wise divergence should reveal a clear gradient from low divergence in shared early blocks to high divergence in cluster-specific late blocks.

P2: Share Middle Layers Only

Configuration P2 represents an exploratory experiment testing the counter-intuitive hypothesis that mid-level tactical pattern recognition (forks, pins, skewers, discovered attacks) might be universal across playstyles, benefiting from shared representations, while low-level feature extraction and high-level strategic planning should remain cluster-specific.

In P2, only the middle residual blocks (layers 6-12) are shared between clusters, representing approximately 3.5 million parameters. These middle layers operate on features extracted by the (cluster-specific) input and early blocks, detecting intermediate-level patterns such as piece coordination, tactical motifs, and local threats. All other layer groups—input block, early residual blocks, late residual blocks, and both output heads—remain cluster-specific.

This configuration produces an unusual divergence signature: high divergence at the input (cluster-specific feature extraction), low divergence in the middle (shared tactical patterns), and high divergence at the output (cluster-specific strategic synthesis). If the hypothesis holds, we should observe that sharing middle layers improves both clusters' ability to recognize tactical opportunities, even though they apply this recognition differently based on their cluster-specific strategic frameworks.

Expected outcomes for P2 are less certain than for other configurations, as this is primarily an exploratory experiment to understand layer roles. We anticipate moderate divergence in input and early layers (divergence index 0.3-0.5), low divergence in middle layers (cosine similarity above 0.95), and high divergence in late layers and heads (divergence index 0.6-1.0). Playstyle separation may be lower than B2 but higher than B1 (tactical score difference 0.10-0.15), and playing strength should be moderate (ELO 1350-1500).

P3: Share Late Layers and Heads

Configuration P3 serves as a control experiment testing the inverse hypothesis to P1: that high-level strategic reasoning is universal across playstyles while low-level feature extraction is playstyle-specific. We include P3 primarily to validate (through expected failure) that output heads must remain cluster-specific to encode distinct move preferences.

In P3, the shared layer groups comprise late residual blocks (layers 13-18), policy head, and value head, totaling approximately 5.5 million parameters. These layers perform strategic synthesis and decision-making. The cluster-specific layers include

the input block, early residual blocks (0-5), and middle residual blocks (6-12), representing the feature extraction and pattern detection stages of the network.

We expect P3 to show poor performance on playstyle preservation metrics, as sharing the policy head forces both clusters to make similar move selections despite training on different data distributions. The shared policy head must compromise between tactical and positional preferences, likely resulting in middling play that satisfies neither cluster's training objective well.

Expected outcomes include low divergence in shared late layers and heads (by construction), higher divergence in cluster-specific early/middle layers, minimal playstyle separation (tactical score difference less than 0.08), and relatively low playing strength (ELO 1250-1400) as the shared decision-making heads cannot specialize for either playstyle. This configuration validates hypothesis H4 (policy head divergence is necessary for playstyle expression) through its anticipated failure.

P4: Share Backbone, Separate Heads

Configuration P4 tests what we hypothesize is the optimal selective aggregation strategy: share the entire residual backbone (all feature extraction and pattern recognition layers) while keeping only the output heads cluster-specific. This configuration embodies the principle that chess features are largely universal, but their application to move selection (policy) and position evaluation (value) depends on playstyle preferences.

In P4, the shared layer groups comprise the input block and all 19 residual blocks, totaling approximately 20 million parameters (87% of the network). These layers learn piece recognition, tactical pattern detection, strategic concept representation, and deep positional understanding. The cluster-specific layer groups include only the policy head and value head (approximately 2 million parameters, 9% of the network).

This configuration enables maximum knowledge transfer—both clusters benefit from a shared, rich representation of chess positions developed through training on the combined dataset of tactical and positional games. However, the cluster-specific heads translate this shared representation into cluster-appropriate decisions: the tactical cluster's policy head learns to prioritize forcing moves, sacrifices, and direct attacks, while the positional cluster's policy head learns to prioritize pawn structure, piece coordination, and long-term advantages.

Expected outcomes for P4 include near-zero divergence across the entire backbone (cosine similarity exceeding 0.99 for all residual blocks), high divergence in the policy head (divergence index 0.7-1.2 as clusters learn fundamentally different move preferences), moderate divergence in the value head (divergence index 0.4-0.7 as evaluation criteria differ but share common factors like material and king safety),

strong playstyle separation (tactical score difference 0.18-0.25), and the highest playing strength among all configurations (ELO 1500-1700) due to maximum knowledge sharing combined with playstyle specialization.

P4 directly addresses research questions about where playstyles are encoded (primarily in output heads) and whether knowledge transfer and playstyle diversity are compatible objectives (yes, through selective head specialization).

3.9.3 Research Hypotheses and Validation

Our experimental design tests five primary hypotheses and five secondary hypotheses regarding selective aggregation, playstyle preservation, and knowledge transfer in federated deep reinforcement learning.

Primary Hypotheses

H1: Selective aggregation preserves playstyles better than full aggregation. We validate H1 by comparing playstyle divergence (standard deviation of tactical scores across clusters) between P1-P4 and the B1 baseline. Success requires that at least one selective configuration (we hypothesize P4) achieves significantly higher playstyle divergence than B1 (two-sample t-test, $p < 0.05$), with tactical score differences exceeding 0.15 compared to B1's expected difference of less than 0.10.

H2: Full aggregation destroys cluster-specific patterns. We validate H2 through layer-wise divergence analysis of B1, demonstrating that continuous averaging of all parameters prevents divergence from developing. Success requires that B1 shows uniformly low divergence (cosine similarity > 0.95) across all layer groups throughout training, even though clusters train on playstyle-filtered data.

H3: Early layers remain shared while late layers diverge. We validate H3 by analyzing per-layer-group divergence in P1-P4, particularly P1 which explicitly tests this structure. Success requires observing a clear layer-depth gradient: shared layers maintain high cosine similarity (> 0.95), cluster-specific layers develop significant divergence (cosine similarity < 0.85 for policy head), with intermediate layers showing intermediate divergence if they are cluster-specific.

H4: Policy head diverges more than value head. We validate H4 by comparing divergence indices for policy and value heads across P1, P2, and P4 (configurations where both heads are cluster-specific). Success requires that policy head divergence exceeds value head divergence by at least 20% (e.g., policy divergence index 0.8 vs. value divergence index 0.6), reflecting stronger playstyle-specific patterns in move selection than in position evaluation.

H5: Divergence increases then plateaus during training. We validate

H5 through temporal divergence analysis, tracking mean divergence across cluster-specific layers over 200 training rounds. Success requires detecting a plateau phase where divergence velocity (change per 10 rounds) approaches zero, typically occurring between rounds 80-120, indicating that clusters have settled into stable distinct representations constrained by the selective aggregation policy.

Secondary Hypotheses

H6: Tactical cluster develops preference for aggressive moves. We validate H6 through move type distribution analysis, comparing captures percentage, checks percentage, and aggressive moves percentage between clusters. Success requires that the tactical cluster shows statistically significantly higher rates (two-sample t-test, $p < 0.01$) for these categories, with captures percentage exceeding the positional cluster by at least 3 percentage points.

H7: Positional cluster develops preference for quiet, strategic moves. We validate H7 similarly to H6, testing whether the positional cluster shows significantly higher pawn advances percentage and quiet moves percentage. Success requires the positional cluster to exceed the tactical cluster by at least 5 percentage points in quiet moves and 4 percentage points in pawn advances.

H8: Both clusters maintain competitive playing strength. We validate H8 by requiring that both clusters achieve ELO ratings above 1400 in the best-performing selective aggregation configuration (hypothesized to be P4). This threshold represents intermediate-level chess ability and demonstrates that playstyle preservation does not come at the cost of basic playing competence.

H9: Model divergence correlates with playstyle divergence. We validate H9 by computing Pearson correlation between global divergence index (mean across all cluster-specific layers) and tactical score difference (absolute difference in mean tactical scores) across all configurations and training rounds. Success requires positive correlation ($r > 0.6$, $p < 0.001$), indicating that parameter-level differences correspond to behavioral differences.

H10: Shared layers enable knowledge transfer. We validate H10 by comparing learning curves (ELO over training rounds) between B2 (no sharing) and P1-P4 (selective sharing). Success requires that selective aggregation configurations achieve target ELO values in fewer rounds than B2, demonstrating accelerated learning through knowledge transfer from shared layers.

3.9.4 Experimental Parameters and Logistics

All experiments use identical base training parameters to ensure fair comparison. Each cluster consists of 4 clients, with training proceeding for 200 rounds. Each

client processes 400 games per round from its playstyle-specific filtered dataset (tactical clients use tactical puzzles, positional clients use positional puzzles). Batch size is 64, initial learning rate is 0.003 with ReduceLROnPlateau scheduling (factor 0.5, patience 15 rounds, minimum learning rate 10^{-6}).

Evaluation occurs every 10 rounds, generating 30 games per cluster against Stockfish opponents at ELO levels 1000, 1200, and 1400 (10 games per level with alternating colors). Each evaluation game is analyzed to extract tactical scores, move type distributions, delta metrics, and all other playstyle characteristics described in Section 3.8. Model divergence metrics are computed every 10 rounds by loading both cluster models and performing layer-wise weight comparisons.

The complete experimental timeline spans approximately 6-8 weeks: Phase 1 (baselines B1 and B2) requires 4-5 days of training plus 2-3 days of analysis, Phase 2 (selective aggregation P1-P4) requires 20-25 days of training plus 5-7 days of analysis, and Phase 3 (final performance evaluation and generalization tests) requires 3-5 days. Total computational requirements are approximately 1000-1200 GPU hours on NVIDIA RTX 3090 or equivalent hardware with 24GB VRAM.

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 mock-

ing 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. Fourth, it performs inter-cluster selective aggregation using the `InterClusterAggregator`, sharing only specified layers across clusters while preserving cluster-specific parameters. 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

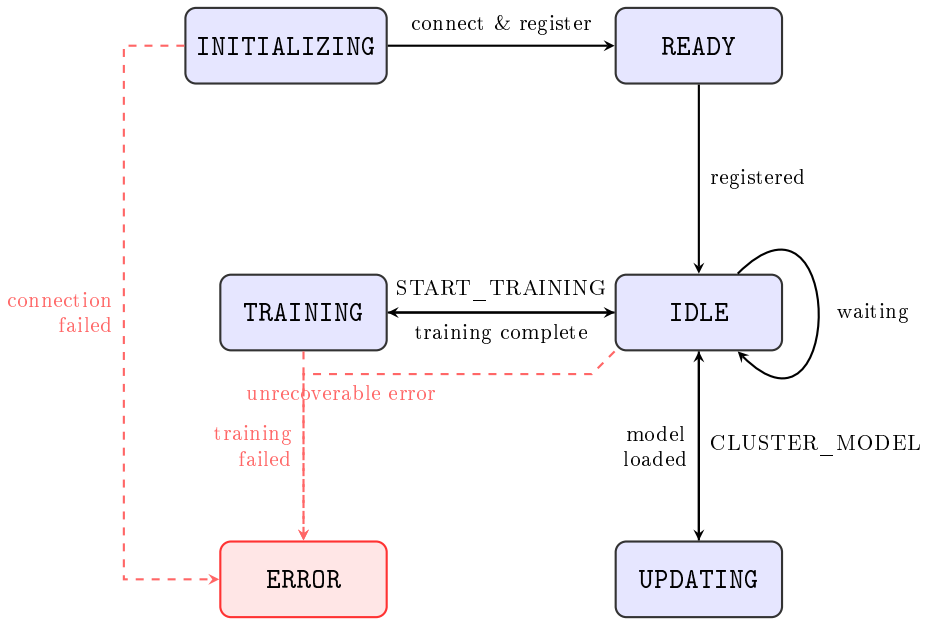

Figure 4.1: Node lifecycle state machine. Solid arrows represent normal state transitions triggered by server messages or training completion. Dashed red arrows indicate error transitions. The `stop()` method can be called from any state to transition to `SHUTDOWN`.

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.

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 the methodology, 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. 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 the methodology chapter, realized as PyTorch modules in `client/trainer/models/alphazero_net.`. 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 resid-

ual 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. This encoding follows the AlphaZero paper's specification, 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) $\times$ 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 $\geq$1, plane 105 if $\geq$2, continuing through plane 118 if $\geq$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. 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 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 im-

plement, 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 FedAvg 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 core innovation of this framework: 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

4.7.1 Model Evaluator

4.7.2 Playstyle Metrics Calculator

4.7.3 Move Type Analyzer

4.7.4 Divergence Calculator

4.7.5 Weight Statistics Tracker

4.8 Storage System

4.8.1 Experiment Tracker

4.8.2 Metrics Store

4.8.3 Model Repository

4.8.4 Plugin System

4.9 Model Serialization

4.9.1 Serialization Format

4.9.2 Parameter Differencing

4.10 Configuration System

4.10.1 Configuration Files

4.10.2 Configuration Dataclasses

4.11 Logging and Monitoring

4.11.1 Structured Logging

4.11.2 Performance Monitoring

Chapter 5

Experimental Setup

5.1 Experimental Design

5.2 Datasets and Benchmarks

5.3 Hyperparameters

Chapter 6

Results and Discussion

6.1 Results

6.2 Analysis

6.3 Discussion

Chapter 7

Conclusion

7.1 Summary

7.2 Contributions

7.3 Future Work

Bibliography

[1] A. Author and B. Author. Example paper title. *Journal Name*, 1:1–10, 2023.

Appendix A

Additional Material