

Progressive Learning Chess Engine Reference Manual

Shyamal Suhana Chandra

Version 1.0
November 17, 2025

Contents

1	Introduction	2
2	Architecture Overview	2
3	Neural Network API	2
3.1	Network Creation	2
3.2	Forward Pass	2
3.3	Backward Pass	3
3.4	Network Destruction	3
4	Optimizer API	3
4.1	Optimizer Types	3
4.2	Optimizer Creation	3
4.3	Weight Update	3
5	Curriculum Learning API	3
5.1	Curriculum Creation	3
5.2	Difficulty Levels	4
5.3	Adding Examples	4
5.4	Level Advancement	4
6	Spaced Repetition API	5
6.1	System Creation	5
6.2	Adding Examples	5
6.3	Getting Next Review	5
6.4	Updating After Review	5
6.5	Long-Term Memory Check	5
7	Pavlovian Learning API	5
7.1	Learner Creation	5
7.2	Stimulus Structures	6
7.3	Pairing Stimuli	6
7.4	Getting Expected Reward	6
7.5	Extinction	6

8	Chess Representation API	6
8.1	Position Creation	6
8.2	FEN Conversion	7
8.3	Matrix Conversion	7
8.4	Move Operations	7
9	Training Engine API	7
9.1	Configuration	7
9.2	Engine Creation	8
9.3	Training	8
9.4	Statistics	8
10	Inference Engine API	8
10.1	Engine Creation	8
10.2	Position Evaluation	8
10.3	Move Prediction	9
10.4	Search	9
11	Multi-Agent Game API	9
11.1	Game Types	9
11.2	Game Creation	9
11.3	Agent Management	9
11.4	Chess as Multi-Agent	9
12	Usage Examples	10
12.1	Basic Training Example	10
12.2	Position Evaluation Example	10
12.3	Curriculum Learning Example	10
13	Error Handling	11
14	Thread Safety	11
15	Performance Considerations	11
16	Memory Management	11
17	Version History	11
18	License	12

1 Introduction

This reference manual provides comprehensive documentation for the Progressive Learning Chess Engine API. The engine implements a hybrid Bayesian-LSTM neural network architecture with curriculum learning, spaced repetition, and Pavlovian conditioning.

2 Architecture Overview

The system consists of the following major components:

- Neural Network (Hybrid Bayesian + LSTM)
- Curriculum Learning System
- Spaced Repetition System
- Pavlovian Learning System
- Chess Representation
- Training Engine
- Inference Engine
- Multi-Agent Game Framework

3 Neural Network API

3.1 Network Creation

```
1 NeuralNetwork* nn_create_hybrid(  
2     size_t input_size,    // Input vector dimension  
3     size_t hidden_size,   // Hidden layer dimension  
4     size_t output_size    // Output vector dimension  
5 );
```

Creates a hybrid neural network with one Bayesian layer and one LSTM layer.

Example:

```
1 NeuralNetwork* nn = nn_create_hybrid(768, 512, 4096);  
2 // 768 = 8 812 chess board representation  
3 // 512 = hidden layer size  
4 // 4096 = possible moves (64 64 )
```

3.2 Forward Pass

```
1 void nn_forward(  
2     NeuralNetwork* nn,  
3     const double* input, // Input vector  
4     double* output       // Output vector (pre-allocated)  
5 );
```

Performs forward propagation through the network.

3.3 Backward Pass

```
1 void nn_backward(  
2     NeuralNetwork* nn,  
3     const double* target, // Target output vector  
4     double* loss          // Computed loss (output)  
5 );
```

Performs backpropagation and computes mean squared error loss.

3.4 Network Destruction

```
1 void nn_destroy(NeuralNetwork* nn);
```

Frees all memory associated with the network.

4 Optimizer API

4.1 Optimizer Types

```
1 typedef enum {  
2     OPTIMIZER_SGD,      // Stochastic Gradient Descent  
3     OPTIMIZER_ADAM,     // Adaptive Moment Estimation  
4     OPTIMIZER_ADAGRAD,  // Adaptive Gradient  
5     OPTIMIZER_RMSPROP   // Root Mean Square Propagation  
6 } OptimizerType;
```

4.2 Optimizer Creation

```
1 Optimizer* optimizer_create(  
2     OptimizerType type,  
3     double learning_rate  
4 );
```

4.3 Weight Update

```
1 void optimizer_update(  
2     Optimizer* opt,  
3     NeuralNetwork* nn  
4 );
```

Updates network weights using the specified optimizer algorithm.

5 Curriculum Learning API

5.1 Curriculum Creation

```
1 Curriculum* curriculum_create(size_t num_levels);
```

Creates a curriculum with the specified number of difficulty levels (typically 10).

5.2 Difficulty Levels

```

1 typedef enum {
2     LEVEL_PRESCHOOL = 0,
3     LEVEL_KINDERGARTEN,
4     LEVEL_ELEMENTARY,
5     LEVEL_MIDDLE_SCHOOL,
6     LEVEL_HIGH_SCHOOL,
7     LEVEL_UNDERGRAD,
8     LEVEL_GRADUATE,
9     LEVEL_MASTER,
10    LEVEL_GRANDMASTER,
11    LEVEL_INFINITE
12 } DifficultyLevelEnum;

```

5.3 Adding Examples

```

1 void curriculum_add_example(
2     Curriculum* curriculum,
3     TrainingExample* example,
4     DifficultyLevelEnum level
5 );

```

Adds a training example to the specified difficulty level.

TrainingExample structure:

```

1 typedef struct {
2     double* input;           // Input vector
3     double* target;         // Target output vector
4     double difficulty;       // Difficulty score (0.0-1.0)
5     size_t input_size;
6     size_t target_size;
7     bool is_correct;        // For spaced repetition
8     size_t attempts;
9     size_t correct_streak;
10    double last_reviewed;
11    double next_review;
12 } TrainingExample;

```

5.4 Level Advancement

```

1 bool curriculum_should_advance(
2     Curriculum* curriculum,
3     double accuracy // Current accuracy (0.0-1.0)
4 );

```

Returns true if accuracy meets the mastery threshold (default 0.85).

```

1 void curriculum_advance_level(Curriculum* curriculum);

```

Advances to the next difficulty level.

```

1 DifficultyLevelEnum curriculum_get_current_level(
2     Curriculum* curriculum
3 );

```

Returns the current difficulty level.

6 Spaced Repetition API

6.1 System Creation

```

1 SpacedRepetition* spaced_repetition_create(
2     size_t capacity,          // Maximum number of examples
3     double ltm_threshold      // Correct streak for LTM (default 5.0)
4 );

```

6.2 Adding Examples

```

1 void spaced_repetition_add_example(
2     SpacedRepetition* sr,
3     TrainingExample* example
4 );

```

6.3 Getting Next Review

```

1 TrainingExample* spaced_repetition_get_next_review(
2     SpacedRepetition* sr
3 );

```

Returns the example that is due for review ($\text{next_review} \leq \text{current time}$).

6.4 Updating After Review

```

1 void spaced_repetition_update_example(
2     SpacedRepetition* sr,
3     size_t index,          // Example index
4     bool is_correct        // Review result
5 );

```

Updates the example based on review result and calculates next review interval using exponential spacing.

6.5 Long-Term Memory Check

```

1 bool spaced_repetition_is_in_ltm(
2     SpacedRepetition* sr,
3     size_t index
4 );

```

Returns true if the example has reached long-term memory ($\text{correct_streak} \geq \text{ltm_threshold}$).

7 Pavlovian Learning API

7.1 Learner Creation

```

1 typedef enum {
2     PAVLOVIAN_CLASSICAL_CONDITIONING,
3     PAVLOVIAN_REWARD_BASED,
4     PAVLOVIAN_INSTRUMENTAL,
5     PAVLOVIAN_HYBRID
6 } PavlovianType;
7
8 PavlovianLearner* pavlovian_learner_create(

```

```

9     PavlovianType type,
10     double learning_rate
11 );

```

7.2 Stimulus Structures

```

1 typedef struct {
2     double* stimulus_vector;
3     size_t stimulus_size;
4     double intensity;
5     double timestamp;
6     size_t occurrence_count;
7 } ConditionedStimulus;
8
9 typedef struct {
10    double* stimulus_vector;
11    size_t stimulus_size;
12    double reward_value; // Positive or negative
13    double intensity;
14    double timestamp;
15 } UnconditionedStimulus;

```

7.3 Pairing Stimuli

```

1 void pavlovian_pair_stimuli(
2     PavlovianLearner* learner,
3     const ConditionedStimulus* cs,
4     const UnconditionedStimulus* us
5 );

```

Pairs a conditioned stimulus (e.g., chess position) with an unconditioned stimulus (e.g., reward/punishment) using the Rescorla-Wagner model.

7.4 Getting Expected Reward

```

1 double pavlovian_get_expected_reward(
2     PavlovianLearner* learner,
3     const ConditionedStimulus* cs
4 );

```

Returns the expected reward value for a given conditioned stimulus based on learned associations.

7.5 Extinction

```

1 void pavlovian_extinction(
2     PavlovianLearner* learner,
3     const ConditionedStimulus* cs
4 );

```

Performs extinction by presenting CS without US, causing association strength to decay.

8 Chess Representation API

8.1 Position Creation

```

1 ChessPosition* chess_position_create(void);

```

Creates a new chess position initialized to the starting position.

```
1 ChessPosition* chess_position_from_fen(const char* fen);
```

Creates a chess position from a FEN string.

Example:

```
1 const char* start_fen =
2   "rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1";
3 ChessPosition* pos = chess_position_from_fen(start_fen);
```

8.2 FEN Conversion

```
1 void chess_position_to_fen(
2   ChessPosition* pos,
3   FENString* fen // Output FEN string
4 );
```

8.3 Matrix Conversion

```
1 void chess_position_to_matrix(
2   ChessPosition* pos,
3   double* matrix // 8 812 output matrix
4 );
```

Converts position to a 3D matrix where each 8×8 plane represents one piece type and color combination.

8.4 Move Operations

```
1 void chess_position_generate_moves(
2   ChessPosition* pos,
3   Color color,           // WHITE or BLACK
4   ChessMove* moves,      // Output array
5   size_t* num_moves      // Number of moves generated
6 );
```

```
1 bool chess_position_is_legal_move(
2   ChessPosition* pos,
3   const ChessMove* move
4 );
```

```
1 void chess_position_make_move(
2   ChessPosition* pos,
3   const ChessMove* move
4 );
```

```
1 void chess_position_unmake_move(ChessPosition* pos);
```

9 Training Engine API

9.1 Configuration

```
1 typedef struct {
2   OptimizerType optimizer_type;
3   double learning_rate;
4   double momentum;
5   double weight_decay;
```



```
6     size_t batch_size;
7     size_t max_epochs;
8     double early_stopping_threshold;
9     bool use_curriculum;
10    bool use_pavlovian;
11    bool use_spaced_repetition;
12    double mastery_threshold;
13    size_t patience;
14 } TrainingConfig;
```

9.2 Engine Creation

```
1 TrainingEngine* training_engine_create(
2     NeuralNetwork* nn,
3     TrainingConfig* config
4 );
```

9.3 Training

```
1 void training_engine_train_with_curriculum(
2     TrainingEngine* engine
3 );
```

Trains the network using curriculum learning, Pavlovian conditioning, and spaced repetition as configured.

9.4 Statistics

```
1 typedef struct {
2     double current_loss;
3     double average_loss;
4     double accuracy;
5     size_t epoch;
6     size_t examples_seen;
7     DifficultyLevelEnum current_level;
8     double training_time;
9     double validation_accuracy;
10 } TrainingStats;
11
12 TrainingStats* training_engine_get_stats(
13     TrainingEngine* engine
14 );
```

10 Inference Engine API

10.1 Engine Creation

```
1 InferenceEngine* inference_engine_create(NeuralNetwork* nn);
```

10.2 Position Evaluation

```
1 double inference_engine_evaluate_position(
2     InferenceEngine* engine,
3     const ChessPosition* pos
4 );
```

Returns an evaluation score for the position (positive favors white, negative favors black).

10.3 Move Prediction

```
1 ChessMove* inference_engine_predict_move(  
2     InferenceEngine* engine,  
3     const ChessPosition* pos  
4 );
```

Predicts the best move for the current position.

10.4 Search

```
1 ChessMove* inference_engine_search_move(  
2     InferenceEngine* engine,  
3     const ChessPosition* pos,  
4     size_t depth    // Search depth  
5 );
```

Performs minimax search to specified depth.

11 Multi-Agent Game API

11.1 Game Types

```
1 typedef enum {  
2     GAME_CHESS,  
3     GAME_FOOTBALL,  
4     GAME_BASKETBALL,  
5     GAME_BASEBALL,  
6     GAME_HOCKEY,  
7     GAME_SOCCER,  
8     GAME_TENNIS,  
9     GAME_GENERIC  
10 } GameType;
```

11.2 Game Creation

```
1 MultiAgentGame* multi_agent_game_create(  
2     GameType game_type,  
3     size_t num_agents  
4 );
```

11.3 Agent Management

```
1 Agent* agent_create(  
2     size_t agent_id,  
3     AgentType type,  
4     size_t action_space_size  
5 );
```

11.4 Chess as Multi-Agent

```
1 MultiAgentGame* chess_as_multi_agent_create(void);
```

Creates a chess game where white and black are separate agents.

12 Usage Examples

12.1 Basic Training Example

```

1 // Create network
2 NeuralNetwork* nn = nn_create_hybrid(768, 512, 4096);
3
4 // Configure training
5 TrainingConfig config;
6 config.optimizer_type = OPTIMIZER_ADAM;
7 config.learning_rate = 0.001;
8 config.use_curriculum = true;
9 config.use_pavlovian = true;
10 config.use_spaced_repetition = true;
11 config.max_epochs = 100;
12
13 // Create training engine
14 TrainingEngine* engine = training_engine_create(nn, &config);
15
16 // Train
17 training_engine_train_with_curriculum(engine);
18
19 // Get statistics
20 TrainingStats* stats = training_engine_get_stats(engine);
21 printf("Accuracy: %.2f%%, Level: %d\n",
22        stats->accuracy * 100, stats->current_level);
23
24 // Cleanup
25 training_engine_destroy(engine);

```

12.2 Position Evaluation Example

```

1 // Create inference engine
2 NeuralNetwork* nn = nn_create_hybrid(768, 512, 4096);
3 InferenceEngine* engine = inference_engine_create(nn);
4
5 // Load position
6 ChessPosition* pos = chess_position_from_fen(
7     "rnbqkbnr/pppppppp/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1"
8 );
9
10 // Evaluate
11 double eval = inference_engine_evaluate_position(engine, pos);
12 printf("Position evaluation: %.4f\n", eval);
13
14 // Predict move
15 ChessMove* move = inference_engine_predict_move(engine, pos);
16 printf("Best move: %d -> %d\n", move->from, move->to);
17
18 // Cleanup
19 chess_position_destroy(pos);
20 inference_engine_destroy(engine);

```

12.3 Curriculum Learning Example

```

1 // Create curriculum
2 Curriculum* curriculum = curriculum_create(10);
3
4 // Add examples at different levels

```

```

5 TrainingExample example;
6 example.input_size = 768;
7 example.target_size = 4096;
8 example.input = new double[768];
9 example.target = new double[4096];
10 // ... populate example ...
11
12 curriculum_add_example(curriculum, &example, LEVEL_PRECHOOL);
13
14 // Train and check advancement
15 double accuracy = 0.90;
16 if (curriculum_should_advance(curriculum, accuracy)) {
17     curriculum_advance_level(curriculum);
18     DifficultyLevelEnum level = curriculum_get_current_level(curriculum);
19     printf("Advanced to level %d\n", level);
20 }
21
22 curriculum_destroy(curriculum);

```

13 Error Handling

All functions that return pointers return NULL on failure. Functions that modify state should be checked for success. Memory management follows standard C++ patterns: objects created with `create` functions must be destroyed with corresponding `destroy` functions.

14 Thread Safety

The current implementation is not thread-safe. Concurrent access to the same network, curriculum, or training engine from multiple threads requires external synchronization.

15 Performance Considerations

- Network forward/backward passes are $O(n \times m)$ where n is input size and m is hidden size
- Curriculum level advancement is $O(1)$
- Spaced repetition review selection is $O(k)$ where k is number of examples
- Position evaluation is $O(1)$ after network forward pass
- Move search is $O(b^d)$ where b is branching factor and d is depth

16 Memory Management

All dynamically allocated memory is managed by the library. Users should:

- Always call `destroy` functions for created objects
- Not free memory returned by the library
- Copy data if persistence beyond object lifetime is needed

17 Version History

- **Version 1.0** (Current): Initial release with full API

18 License

Copyright (C) 2025, Shyamal Suhana Chandra. All rights reserved.