README.md

Backgammon

Python modules to play backgammon (human or computer).

System Requirements

- **Python 3**
  (You may need to change the commands below to `python3 ...` if that is how you run Python 3 on your machine)

How to Run the Game

- **Human vs Computer**:
  Run `python single_player.py`, then choose the computer strategy to play against.
- **Human vs Human**:
  Run `python two_player.py`.
- **Computer vs Computer**:
  Run `python main.py` (the two 'players' can have different strategies).
- **Tournament**:
  Run `python tournament.py`.

The tournament mode runs many games with different starting players and returns the probability of the strategies being equally good.

How to Run Training

Our RL training pipeline is now interactive and offers two modes:

1. **Pretraining**:
   In this mode, the system generates random board samples from games played by random-move players. These samples are used to train the neural network to approximate our handcrafted heuristic function. Pretraining uses:
   - **Learning Rate**: 0.001
   - **Batch Size**: 50
   - **Sample Generation**: 5,000 games

    - **Early Stopping**: Triggered if improvement is less than 0.0001 for 10 consecutive epochs

2. **Outcome-Based Training**:
   This mode refines the pretrained model using board samples generated by a best-of-four strategy. The training is based on game outcomes:

    - Samples are generated by having the heuristic player select from the top 4 move sequences.
    - Each sample is labeled with a target value (1 for winning boards, 0 for losing boards).
    - The model is trained for a fixed number of epochs per round (e.g., 100 epochs) and tested over 10 games.
    - Training continues until the model wins all test games (an average win rate of 1.0) or until a maximum number of rounds is reached.
    - Progress is visualized via a plot of average wins per training round.

To run the training, execute `RL_training.py`. You will be prompted to choose an option:

- Enter **1** for Pretraining.
- Enter **2** for Outcome-Based Training.

Feature Vector and Phase Encoding

Our backgammon project extracts rich features from a board position to feed into our evaluation and reinforcement learning algorithms. The core idea is to represent each board state as a fixed-length numerical vector that captures important aspects of the game and then augment this representation with phase information (early, mid, or late game).

16-Dimensional Base Feature Vector

The base feature vector consists of **16 features**, where the first 8 represent the current player and the last 8 represent the opponent. These include:

- **Occupied Points**: Number of board points (1-24) occupied by two or more pieces.
- **Borne-Off Pieces**: Pieces that have exited the board.
- **Pieces on the Bar**: Pieces waiting to re-enter play.
- **Blot Distance Sum**: Vulnerability of single pieces (blots).
- **Total Distance**: Sum of distances of all pieces to home.
- **Excess Distance**: Extra distance beyond the home zone.
- **Blot Count**: Number of single-piece positions.
- **Pieces on Board**: Total number of pieces in play.

Phase Calculation and Extended Feature Vector

In addition to the base 16 features, the game phase is determined by analyzing the current player's board state:

- **Pieces Out of Home:**
  Calculated as
  ```
  pieces_out = curr_on_board - curr_pieces_home
  ```
  (i.e., the total pieces on board minus those already in the home area).

Based on this value, the phase is assigned as follows:

- **Late Game:** If fewer than 3 pieces are out of home.
- **Early Game:** If more than 8 pieces are out of home.
- **Mid Game:** Otherwise.

Phase Calculation: We represent the game phase using an integer: 0: Early game 1: Mid game 2: Late game

To integrate the phase with the base feature vector, the phase integer is converted into a one-hot vector (e.g., early becomes [1, 0, 0]) and is then used to "gate" the base 16-dimensional vector. “gate” the 16 features with the phase vector. That is, we replicate the 16 features three times (one block for each phase) and multiply each block by the corresponding one-hot element. For example, if the game is in the early phase, the final augmented feature vector becomes:

[1 × (16 features), 0 × (16 features), 0 × (16 features)]

yielding a 48-dimensional vector. This phase-aware representation allows our neural network to learn a single evaluation function that adapts to different stages of the game.

Normalization

For training stability, the features can be normalized by dividing by known maximum values:

- **TOTAL_PIECES:** 15 (each side has 15 pieces)
- **BOARD_SIZE:** 25 (used in distance computations)
- **MAX_OCCUPIED_POINTS:** 7 (maximum possible occupied points)
- **MAX_DISTANCES:** 360 (maximum sum of distances)
- **MAX_EXCESS_DISTANCE:** 270 (maximum sum of excess distances)

When normalization is enabled, each feature is scaled to the range [0, 1].

Implementation Overview

The key functions in `feature_vector.py` are:

- **`board_to_vector(current_color, board, should_normalize)`**
  Computes the 16-dimensional base feature vector (for both the current player and the opponent) along with the phase (as an integer 0, 1, or 2).

- **`get_phase(...)`**
  Determines the phase of the current player's board using the logic described above and returns the phase (as an integer 0, 1, or 2).

- **`board_to_extended_vector(player, board)`**
  Converts the phase integer into a one-hot vector and uses it to produce the final 48-dimensional phase-aware vector.

This design allows our reinforcement learning and evaluation modules to work with a concise, normalized, and phase-aware representation of the board, enabling effective learning and game analysis.

Heuristic Weights and Strategies

Heuristic Weights

Our heuristic evaluation uses a set of phase-dependent weights that adjust the importance of each feature based on the game phase. The weights are stored in a 16×3 NumPy matrix where each row corresponds to a feature (in the same order as the 16-dimensional feature vector) and each column corresponds to a phase:

Feature	Early	Mid	Late	Explanation
Occupied Points	1.3	1.7	0.8	Important early to establish strong positions, less valuable late when pieces are moving home.
Borne-Off Pieces	3.0	5.0	8.0	Increases in importance as the game progresses since winning is determined by bearing off first.
Pieces on Bar	0.0	0.0	0.0	We do not directly reward or penalize pieces on the bar in this heuristic.
Blot Distance	-0.5	-0.3	-0.1	Vulnerability decreases in importance as blots become rarer in later phases.
Total Distance	-0.6	-0.4	-0.2	Distance matters most early when trying to establish a lead, but becomes less relevant when bearing off.

Feature	Early	Mid	Late	Explanation
Excess Distance	-0.7	-0.9	-1.3	More important late game as pieces need to move home faster.
Blot Count	-0.5	-0.7	-0.3	Avoiding blots is crucial in midgame, where being hit is most costly.
Pieces on Board	1.4	1.2	1.7	Generally valuable, especially late game when every piece needs to bear off.
Opponent Pieces on Board	0.0	0.0	0.0	Not considered in this heuristic.
Opponent Blot Count	0.0	0.0	0.0	Not considered in this heuristic.
Opponent Excess Distance	0.0	0.0	0.0	Not considered in this heuristic.
Opponent Total Distance	0.8	1.1	0.5	Important midgame when trying to slow the opponent down.
Opponent Blot Distance	0.0	0.0	0.0	Not considered in this heuristic.
Opponent Pieces on Bar	2.2	2.7	3.2	More opponent pieces on the bar is always beneficial.
Opponent Borne-Off Pieces	0.0	0.0	0.0	Not directly considered in this heuristic.
Opponent Occupied Points	0.0	0.0	0.0	Not considered in this heuristic.

When evaluating a board, the phase (returned as an integer between 0 and 2) is used to select the corresponding column of weights. The board evaluation is computed by taking the dot product between the 16-dimensional normalized feature vector and the selected weight vector. The result is then normalized to the range [0, 1] using pre-defined MIN and MAX score norms.

Why Min-Max Normalization?

We apply Min-Max normalization to our heuristic scores to ensure consistent scaling and stable training in our reinforcement learning system. The primary reasons are:

1. **Consistency of Scale:**
   The heuristic produces scores that vary across different board states. Min-Max normalization ensures all scores fall within a fixed range ([0,1]), making comparisons across positions meaningful.

2. **Preserving Relative Order:**
   Since our heuristic is based on weighted sums of board features, we need to maintain the order of evaluations—i.e., if one board position is better than another before normalization, it should remain better after normalization. Min-Max scaling preserves these relationships.

3. **Stability in Neural Network Training:**
   Helps stabilize training by keeping inputs in a well-defined range. Many machine learning models, including reinforcement learning agents, train more effectively when their input values remain between ([0,1]), preventing extreme values from dominating the learning process.

4. **Handling Negative and Positive Scores:**
   Our heuristic function produces both positive and negative scores. Min-Max normalization shifts and scales these scores so that all values are mapped to a non-negative range while maintaining their relative differences.

The normalization formula is:

```
score = (score - MIN_SCORE_NORM) / (MAX_SCORE_NORM - MIN_SCORE_NORM)
```

Where:

- `MAX_SCORE_NORM` is chosen as the sum of the maximum positive contributions from the heuristic weights.
- `MIN_SCORE_NORM` is chosen as the sum of the maximum negative contributions.

This transformation ensures that the heuristic values remain in a predictable range, making them more suitable for training reinforcement learning models.

Strategy Classes

`BestMoveStrategy`

This is the base class that implements move selection by recursively evaluating all possible moves. It:

- Uses the feature extraction (with phase information) to evaluate board states.
- Explores moves recursively to choose the sequence with the highest evaluation.

`BestMoveHeuristic`

This class extends `BestMoveStrategy` by implementing the `evaluate_board` method:

- It obtains the 16-dimensional feature vector and the phase from `board_to_vector()`.
- It selects the appropriate 16-dimensional weight vector from the heuristic weight matrix using the phase.
- It computes the evaluation score as the dot product of the feature vector and the weights.
- Finally, it applies Min-Max normalization to map the score to [0, 1].

`BestMoveModel`

This class extends `BestMoveStrategy` to use a neural network model for board evaluation:

- It receives a pretrained model in its constructor and sets the model to evaluation mode.
- In its `evaluate_board` method, it extracts the extended board vector from the current board state using `board_to_extended_vector()`, converts it into a tensor (adding a batch dimension), and then obtains the evaluation score from the model.
- The resulting score is a scalar value in the range [0,1] (as the training process ensures target normalization).

`BestOfFourStrategy`

This class also leverages the `BestMoveHeuristic` evaluation but adds a **sampling step from the top four move sequences**:

- Recursively generates possible move sequences using a method similar to `BestMoveStrategy`.
- Sorts these sequences by evaluation score and takes only the top four.
- From those top sequences, it chooses one via a **softmax-like selection** (exponential weights), balancing exploration with the highest-valued sequence.
- If fewer than four move sequences exist, it simply picks the best one.

Neural Network for Board Evaluation

We use a fully connected feedforward neural network to evaluate board positions. This network is designed to learn an evaluation function based on a two-stage training process.

Network Architecture

Our model consists of 3 fully connected layers:

Layer	Input Size	Output Size	Activation
fc1	48	64	ReLU
fc2	64	64	ReLU
fc3	64	1	None (Linear)

Note: We do not use any activation on the final layer. Instead, all target values are pre-normalized to the [0,1] range using Min-Max scaling.

Training Process

Part 1: Heuristic Pre-training

- **Data Collection:**
  Sample board positions from games played by two random-move players.
- **Target Values:**
  Use our handcrafted heuristic function (which is normalized to [0,1]) as the target for each board.
- **Objective:**
  Train the network in a supervised manner so that it learns to approximate our heuristic evaluation.

Part 2: Winner/Loser Training

- **Data Collection:**
  Generate board samples using the heuristic player that selects from the 4 best actions.
- **Target Values:**
  Label boards based on the eventual outcome: boards from winning positions get high target values, and boards from losing positions get low target values.
- **Objective:**
  Refine the network by training it to predict outcome-based values, effectively learning to distinguish winning from losing board positions.

Why This Architecture and Process?

- **3 Layers (with two hidden layers of size 64):**
  Provides enough capacity to capture complex board features and strategic nuances.
- **No Activation on the Final Layer:**
  Allows the model to produce a raw scalar output, while target values are normalized to [0,1] during training.
- **Two-Stage Training:**

- **Stage 1:** Provides a strong starting point by mimicking the handcrafted heuristic.
- **Stage 2:** Improves performance by focusing on actual game outcomes (winner vs. loser) using a more selective sample of board positions.

This approach ensures the network first learns a good approximation of our heuristic and then refines its evaluation to better predict winning outcomes.

Pretraining and Sample Generation

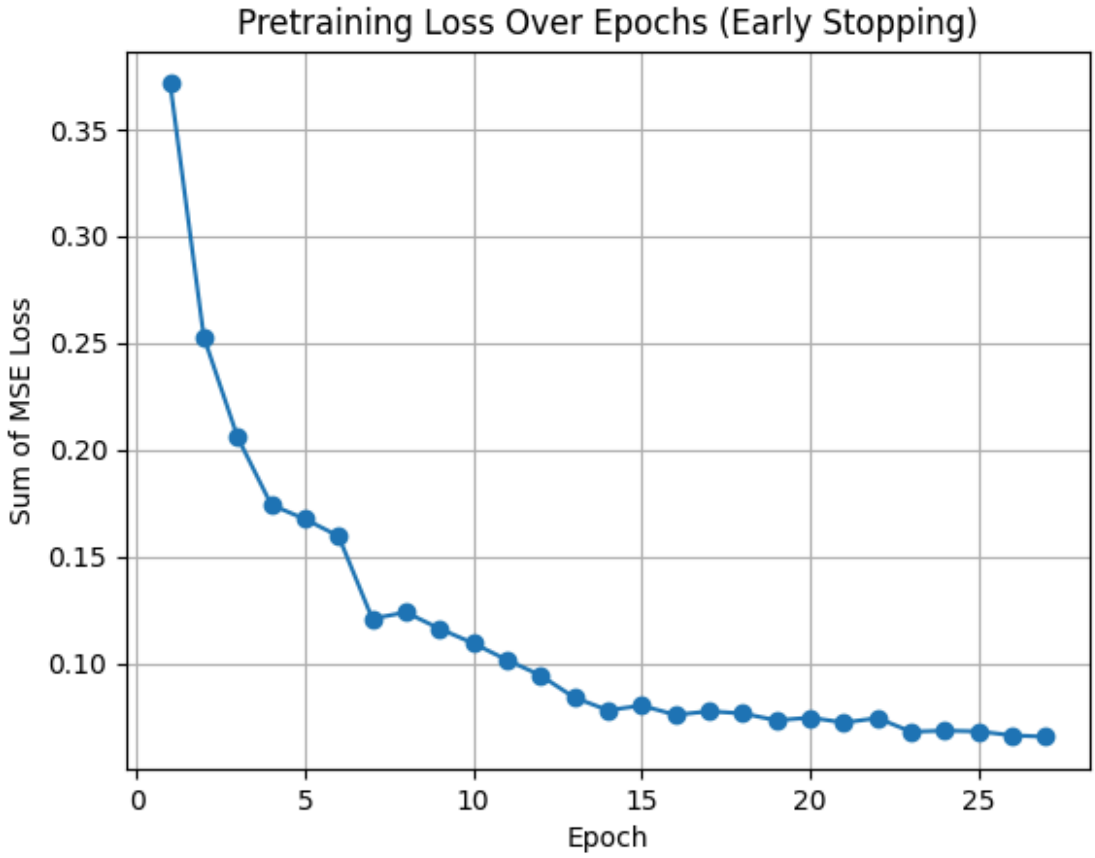
We have introduced a new training pipeline for our heuristic evaluation network. Key additions include:

- **train.py**:
  - Implements functions for pretraining the `HeuristicNN` model using random board samples.
  - Samples are generated via `sample_random` (located in `RL/game.py`).
  - Uses MSE loss with `"sum"` reduction and mini-batch updates.
  - Plots and saves the sum of MSE loss over epochs, with early stopping based on improvement.
- **RL_TRAINING.py**:
  - Serves as the main entry point to run the pretraining process.

Training Process Overview

Stage 1: Pretraining

- Random board samples are generated from games played by random-move players.
- The model is trained in a supervised manner using our heuristic function values (normalized to [0,1]) as targets.
- Training progress is monitored via a loss curve that updates every epoch.



Hyperparameter Choices

- **Learning Rate (0.001):**
  This value strikes a good balance between convergence speed and stability. It allows the optimizer (Adam) to make steady progress without overshooting minima, which is crucial given our network's architecture and the variability in our sampled board positions.

- **Batch Update Size (50):**
  A batch size of 50 provides sufficiently representative gradient estimates while keeping memory usage and computation time manageable. This size helps smooth the training process, especially when working with the relatively small input vector size (48) and a moderate dataset.

Data Shuffling

- **Shuffling the Examples:**
  Each epoch, the training examples are shuffled before creating mini-batches. This step is critical to:
  - **Break correlations** between consecutive samples, ensuring that gradient updates are not biased by the order of the data.
  - **Improve generalization** by presenting a more diverse mix of samples in each mini-batch.

- **Stabilize training**, especially when the data distribution may vary across the game samples.

All training outputs (e.g., loss plots, model checkpoints) are saved in the `RL/pretrain_output` directory.

Stage 2: Outcome-Based Training

Objective:

Refine the pretrained model using game outcomes so that it better predicts winning and losing board positions.

Data Collection:

New board samples are generated using a best-of-four strategy. Each board is labeled with: 1: If the board is generated by the winning side. 0: If generated by the losing side.

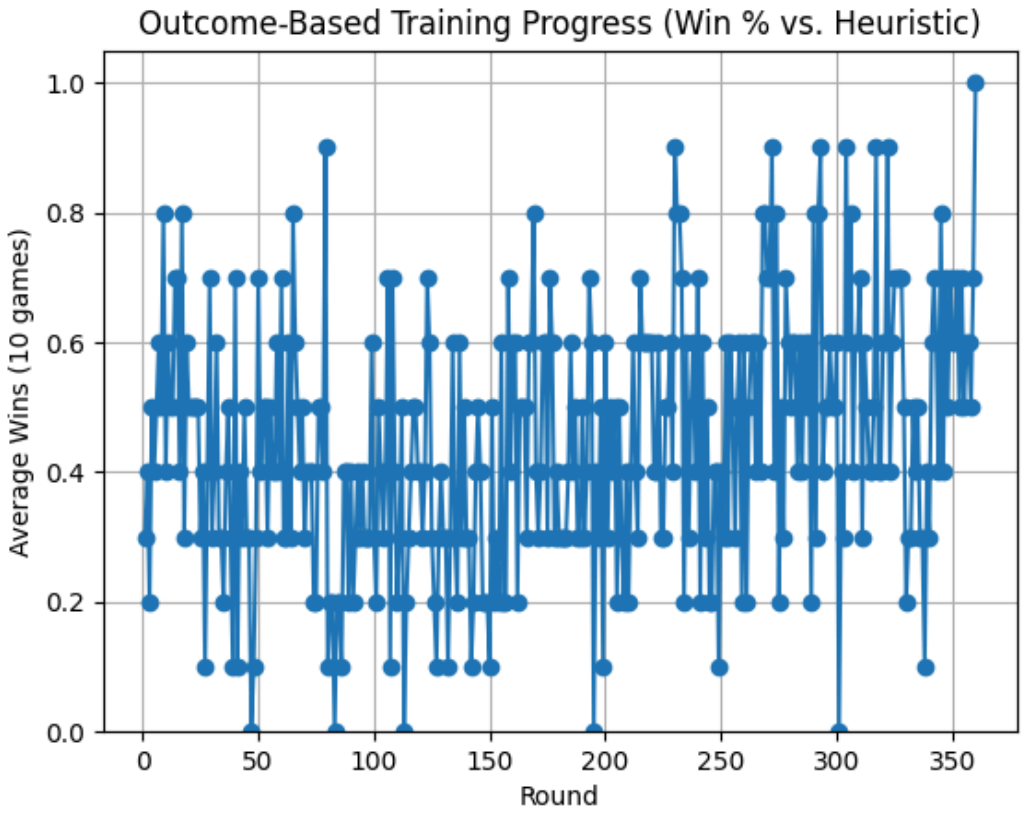
Training Details:

The network is trained on these outcome-based samples for a fixed number of epochs per training round. Testing is performed after each round against both the heuristic player and previous versions of the model.

Monitoring Progress:

Two key plots are generated:

- **Win Rate Plot:** Displays the average win percentage of the model against a heuristic opponent over training rounds.
- **Improvement Plot:** Shows the win rate of the new model version compared to the previous version, ensuring that each round brings improvements.



Replay Buffer and Sample Management

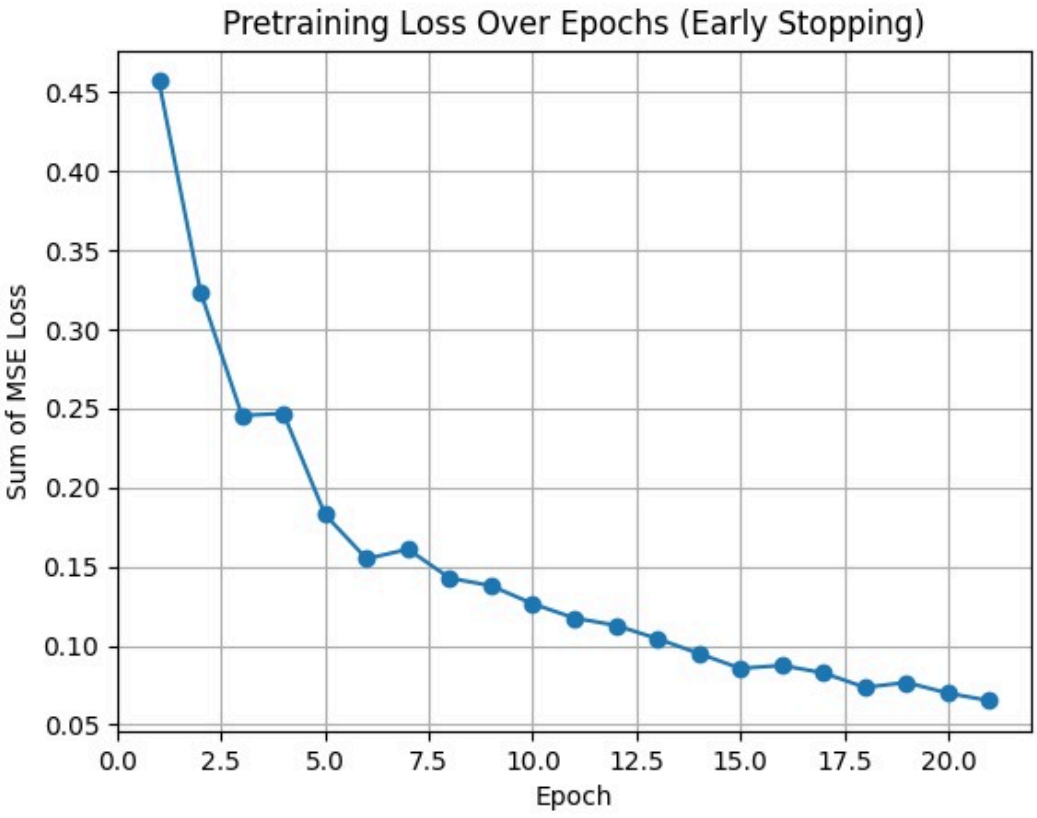
To maintain a diverse yet manageable set of training samples, the system uses a replay buffer implemented as a file (e.g., RL/board_samples). A helper function, trim_samples, is employed to ensure the file contains no more than a fixed number of entries (e.g., 10,000). This mechanism helps in:

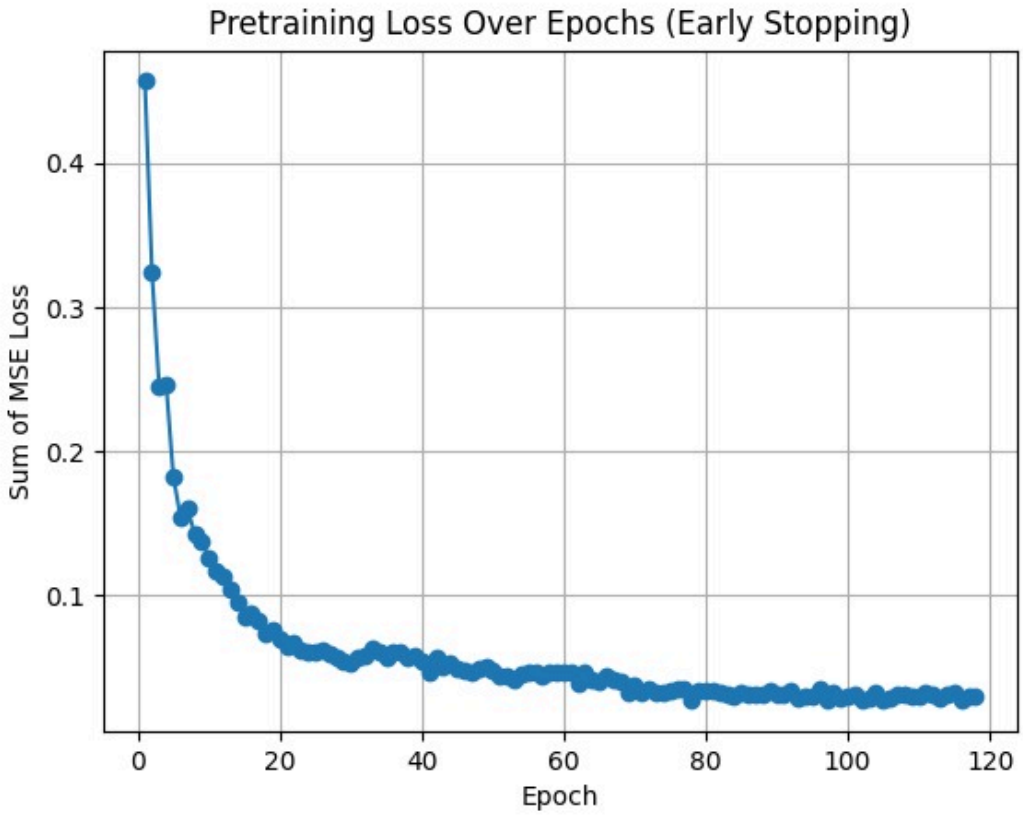
- **Retaining Recent and Valuable Samples:** Ensuring the model is trained on both recent experiences and a diverse set of past game situations. Preventing Dataset Overgrowth: Keeping file I/O and memory usage within practical limits.

results

pretraining results

- **pretrained model is winning 100% against random player**
- **difference Variance: 0.0025739907287061214**
- **difference Average: 0.299162894487381**
- **win rate against our huristics:** 51%
- **win rate against simple huristic:** 50%
- **win rate against hard huristic:** 30%
- **graph**





training results

best training model is a model that was pretrained on **our huristc** but trained against **other** huristics

- **after training** the model statistics are:
- **win rate against our huristics:** small improvement from 0.51 to 0.55 win rate
- **win rate against simple huristic:** improvement from 0.5 to 0.72 win rate
- **win rate against hard huristic:** improvement from 0.3 to 0.45 win rate
- **win rate trained against pre trained model:** 60%

conclusions

- we are getting improvements when training against other huristics
- the learning is not very stable but it happens over time could be so that we can improve it by training it for longer, with better replay buffer, and more huristic players.