Robo Arm

Frederik Schwarz
Informatik
Hof-University

Marie Müller
Informatik
Hof-University

Kitti Kern
Informatik
Hof-University

Eugen Ganscha
Informatik
Hof-University

***Abstract*—This work presents a robotic system capable of autonomously playing Nine Men's Morris by integrating computer vision and strategic decision-making. The system employs a robotic manipulator to physically move pieces on the board while leveraging YOLO-based visual perception to detect the current board state in real time. A custom-trained model evaluates the game state and selects optimal moves, enabling the robot to play strategically against human or simulated opponents. We detail the design of the perception pipeline, the interfacing of the robot arm with the board, and the training of the game model, emphasizing how these components interact to achieve seamless gameplay.**

I. Introduction

Robotic systems that interact with physical game environments provide a compact yet challenging testbed for perception, decision-making, and actuation. In this work, we present a modular robotic system capable of playing the board game Nine Men's Morris using a robotic arm. The project combines computer vision for board state detection, artificial intelligence for strategic gameplay, and precise robotic manipulation to execute moves on a physical board. The game was chosen due to its relatively simple ruleset, discrete state space, and strong dependence on long-term strategy, making it well suited for evaluating the interaction between symbolic game logic and real-world execution.

To manage system complexity and enable parallel development, the project was split into three largely independent components. The robotic arm control was developed by Eugen and Karla, focusing on reliable grasping and placement of pieces. Board state detection was implemented by Marie and Kitti, who were responsible for perceiving the current game state from sensor input. The AI component, including game logic and training of a strategy to play Nine Men's Morris, was developed by Frederik. To ensure seamless integration, all components were implemented as Python modules exposing a single class with one primary function that performs the respective task. This design allowed each subsystem to be developed, tested, and refined independently while maintaining a consistent integration interface.

A common high-level API was defined based on an abstract representation of the game state and actions. All components operate on a shared board representation consisting of an array of 24 elements, corresponding to the 24 valid positions on a Nine Men's Morris board. Each index represents a fixed board position, numbered sequentially starting from the top-left position and continuing in a consistent order. This unified abstraction decouples perception, decision-making, and actuation, enabling the AI to reason purely symbolically while the perception and robotic subsystems translate between physical reality and the abstract game model. The resulting architecture demonstrates how clear abstractions and modular interfaces can simplify the development of integrated robotic game-playing systems.

II. Computer Vision

A. Model Selection

For the vision-based perception module of this robotics project, we selected YOLOv8 because it is designed for real-time inference and is supported by a mature, integrated toolchain. In addition to strong empirical performance, YOLOv8 offers comparatively consistent training and inference. It is a lightweight variant and efficient enough for development on resource-constrained hardware. Owing to its accessible workflows and straightforward usage, this YOLO variant is also a practical choice for teams with limited prior experience in deep learning, while still providing some advanced features [1]. In this application context, detecting a game board and pieces, newer YOLO variants may introduce additional complexity and higher computational requirements that do not yield a proportional benefit for this required detection task. At the same time, YOLOv8 offers competitive inference speed, which is essential for responsive robotic systems that must react reliably under time constraints [2].

We implemented object detection via the Ultralytics Python library. Ultralytics provides an end-to-end pipeline that covers the full inference chain from image input to final detection, including preprocessing and postprocessing. Conceptually, YOLO (You Only Look Once) is a one-stage detector. Given an input image, it produces bounding boxes, class labels, and confidence scores within a single forward pass.

In practice, the inference workflow begins with preprocessing. The given input image is resized to a fixed resolution, and the pixel intensities are normalized. The preprocessing is followed by the forward pass through the network's backbone, neck, and head, which yields raw detection candidates.

The backbone serves as the feature extractor that transforms raw pixels into increasingly abstract feature maps. Early backbone stages primarily encode low-level structures such

as edges and textures, while deeper stages represent more complex object components. This transformation is realized via efficient convolutional blocks and progressive downsampling, which produces multiple resolution levels. The resulting multi-scale representation is critical for robust detection across object sizes, enabling the model to localize both small pieces and larger structures within the same scene. These feature maps are then propagated downstream to the remaining detection components [3].

The neck aggregates and fuses information across scales by combining high-resolution, detail-rich features with lower-resolution, semantically stronger features. As a result, the head receives enriched feature maps at multiple scales, which improves localization and classification in scenes with varying object sizes and local clutter [3].

Finally, the head generates the final predictions. YOLOv8 employs an anchor-free, decoupled head. In classical anchor-based detectors, anchors are predefined bounding-box templates of fixed sizes and aspect ratios that are tiled across the image. The model then learns offsets to adapt these templates to objects. In contrast, anchor-free detection predicts bounding boxes directly from feature-map locations without relying on predefined templates. Moreover, in YOLOv8, the head is decoupled, meaning that box regression and classification are in separate branches, which often leads to more stable optimization behavior in practice. Nevertheless, the raw output typically contains a large number of partially redundant and strongly overlapping bounding boxes [4].

Therefore, the postprocessing stage applies confidence filtering and non-maximum suppression (short: NMS). NMS refers to a class of algorithms that selects a single representative prediction from a set of overlapping candidates. In this context, it retains the bounding box with the highest confidence score. This yields consistent final detections per object and reduces ambiguity in downstream processing [5].

The image detection part of this project is divided into three separate YOLOv8 models. We use separate models for stones, the board, and stacks because the target objects differ substantially in their visual characteristics. Training a single joint model would introduce unnecessary competition between these heterogeneous classes and could therefore reduce class separability, particularly under challenging conditions such as varying illumination or dense piece configurations. The board-detection module first reliably isolates the relevant region of interest and subsequently enables perspective normalization. As a result, downstream object detection becomes substantially less sensitive to camera pose and image distortion. A dedicated stack detector is justified because stacks differ in appearance from individual stones and would otherwise be more prone to class confusion and training instabilities caused by class imbalance. This modular separation also improves maintainability and iteration speed, as each model can be retrained and optimized independently.

B. Dataset and Labeling Protocol

For data acquisition, we captured custom images of a Nine Men's Morris board. To promote generalization to realistic operating conditions, we varied camera positions, viewing distance, and lighting setups, thereby introducing variance in perspective and brightness. The dataset comprises 561 images in total: 150 images for board detection, 211 images for white and black stones, and 200 images for white and black stacks. For stones and stacks, each image contains between 1 and 18 pieces.

Annotation was performed manually using bounding boxes. For board detection, the class board was labeled. For piece detection on the perspective-normalized board view, the classes are stone_white, stone_black, stack_white, and stack_black. Labeling was conducted using the browser-based tool Make Sense, which supports exporting annotations in YOLO format. For each image, a corresponding label file stores the normalized bounding-box parameters (x_center, y_center, width, height) as well as the associated class.

C. Board Detection

Board detection constitutes the entry point to the perception pipeline and is executed as YOLOv8-based object detection (e.g., using models/board_best.pt or models/board_last.pt). For each frame, the detector outputs a set of candidate boxes to keep postprocessing simple and to minimize heuristic decision logic. The pipeline selects the prediction with the highest confidence (best_board_box) for subsequent processing. The result is an axis-aligned board bounding box, from which four corner points are derived as the interface to the subsequent perspective normalization stage. This normalization step reduces sensitivity to camera pose and projective distortions and thereby stabilizes downstream detection of stones and stacks by operating on a canonical, rectified board representation.

D. Perspective Normalization via Homography

Reliable board-state estimation requires a consistent coordinate frame under changes in camera pose (tilt, in-plane rotation, distance). This is addressed via planar perspective normalization: once the board region is detected, it is rectified into a fixed, top-down view using a homography (projective transformation). Homographies are applicable when the observed structure is well-approximated as planar under a projective camera model [6], [7]. A homography maps points between two planes according to

$$p' \sim Hp, \quad H \in \mathbb{R}^{3\times 3}, \tag{1}$$

where image points are written in homogeneous form $p = (x, y, 1)^T$. The symbol $\sim$ denotes equality up to a non-zero scale factor. After transformation, the inhomogeneous coordinates are recovered by normalizing with the third component w, i.e., $x' = \frac{\tilde{x}}{\tilde{w}}$ and $y' = \frac{\tilde{y}}{\tilde{w}}$ [6], [8]. This additional third coordinate enables translation and projective effects (e.g., foreshortening) to be expressed as a single matrix operation. This 3×3 setup and its degrees of freedom are standard in multi-view geometry [6], [7].

In the implemented pipeline, board localization is performed by a learned detector that returns a bounding box. From that box, the four corners are extracted in a fixed order (top-left, top-right, bottom-right, bottom-left) and mapped to a predefined square output geometry. The homography H is computed and applied with OpenCV's perspective trans-

form utilities (getPerspectiveTransform and warpPerspective) [9], [10]. warpPerspective performs inverse-mapped, per-pixel resampling under the estimated homography, producing a rectified board image in a fixed target grid [9].

The canonical board view is rendered at a steady resolution (e.g., 1000×1000 px) so that downstream steps - index mapping, thresholding, and plausibility checks – operate in a stable pixel metric.

A design trade-off comes with estimating corners from a bounding box: it is simple and fast, but not as robust to extreme viewpoints as methods that explicitly regress corners or trace the board edge. In practice, that means the camera pose needs to stay within a limited range for reliable rectification.

E. Stone/stack Detection and Candidate Extraction

Object detection is split into three parts: (1) board detection, (2) stone detection on the board, and (3) stack detection off the board. The stone and stack detectors use two classes (black and white) to directly infer color during detection. Inference is executed via the Ultralytics YOLO runtime interface, which exposes per-detection bounding boxes, class IDs, and confidence scores [11].

A critical geometric constraint applies to stone detection: stones are detected only on the rectified board image. To prevent a mismatch between training and runtime use induced by perspective distortion, stones-model training images are warped into the same canonical geometry prior to labeling and training. This ensures that the learned appearance distribution matches the runtime input distribution (top-down board appearance).

Stack detection is performed in the original camera frame (off-board region) and therefore operates in unrectified image coordinates; no pre-warping is applied to stack training data. Stack detections are used to estimate off-board piece availability

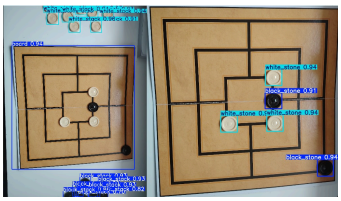

Fig. 1. Qualitative detection example Left: raw camera frame with board bounding box and off-board stack detections. Right: rectified top-down board view used for on-board stone detection and subsequent index mapping.

Candidate extraction converts each predicted bounding box into a point hypothesis using its center (c_x, c_y). Ultralytics provides box conversions including $[x_{\text{center}}, y_{\text{center}}, w, h]$, which enables direct retrieval of centers for subsequent mapping [12].

Detections are filtered in two stages:

1) Minimum confidence filter: anything below a fixed confidence threshold $c_{\min}$ is discarded.
2) Spatial plausibility filter (after mapping): once a detection is assigned to the nearest board index (see Section C), any detection whose center lies farther than a maximum distance $d_{\max}$ from that index is rejected. This distance threshold $d_{\max}$ is scaled by the estimated board grid spacing in pixels, so it stays consistent across different canonical resolutions.

If several detections land on the same board position, the best candidate wins: one hypothesis per index, usually the one with the highest confidence. If there is a tie, the closer center-to-index distance breaks it. This produces a sparse and stable set of per-position candidates suitable for discrete state reconstruction.

F. Index Mapping and State Building

The normalized board view provides a deterministic mapping from image-space hypotheses to the discrete Nine Men's Morris state. Playable positions are represented as a fixed set of indices with normalized coordinates $(x_{\text{rel}}, y_{\text{rel}}) \in [0, 1]$ and stored in a CSV file. At runtime, those coordinates are mapped to pixel coordinates in the canonical image via $(x_i, y_i) = (x_{\text{rel}}S, y_{\text{rel}}S)$, where S is the board's canonical side length in pixels.

Each detected stone center (c_x, c_y) is assigned to the nearest index by Euclidean distance:

$$i^* = \arg\min_i \sqrt{(c_x - x_i)^2 + (c_y - y_i)^2}. \tag{2}$$

To increase robustness against spurious detections, a scale-aware acceptance radius is applied. This radius is based on the board geometry, e.g., the typical grid spacing, computed as the median distance to the nearest neighbor among the indices in pixel space. Detections exceeding a chosen fraction of this spacing are rejected as implausible placements.

The final output is a fixed-length occupancy array over the 24 positions using a ternary encoding compatible with the game logic: 0 for empty, −1 for black, +1 for white. In addition, temporal state tracking compares the estimated occupancy array against the previously known game state and reports a delta (changed indices) only after the same changed state is observed consistently across several consecutive frames, thereby reducing flicker-induced false updates.

G. Assumptions and Limitations

Performance depends on the validity of the planar-board assumption and on accurate board localization. Errors in bounding-box localization directly affect corner derivation and therefore propagate into the homography, shifting mapped stone centers and increasing index-assignment ambiguity. [6], [7] Motion blur, strong specular reflections, partial board edge occlusion, and low contrast backgrounds can cause additional degradation. These limitations motivate either tighter constraints on camera placement and lighting or the replacement of box-derived corners with corner regression/segmentation-based localization under wider viewpoint variations.

III. AI

A. State Encoding

To prepare the game data, the positions of the game pieces are represented using one-hot encoding. In addition to the

position-related information, global features are integrated into the state vector. These include the current game phase (“placing”, “moving”, “jumping”), which is also one-hot encoded, the number of pieces still to be placed in normalized form, and binary indicators specifying whether a piece must be removed in the current turn and whether the game is currently in the jumping phase.

The encoding of the game phase is deliberately redundant in order to explicitly convey to the model that the movement rules change once a player has only three pieces remaining on the board. In this way, the game state is always coupled to two inputs: first, information about whether pieces still need to be placed (placing phase), and second, the explicit state encoding of the current game phase. This results in two separate input representations: a tensor-shaped representation of dimension (3, 24) for modeling the piece positions, and a vector of length 11 describing the global state features.

The output space of the model is defined as a one-dimensional array of length 24×25. Each possible combination of source and target positions is represented by a distinct probability. Indices 0 to 23 correspond to the actual positions on the game board, while index 24 is used as a placeholder when no source or target position exists, such as during the placing or removal of a piece. The corresponding index in the output array is computed as `index = source × 24 + target` where source denotes the starting position and target the destination position.

The eleventh global feature, indicating whether a stone must be removed, was introduced at a later stage of development. As a consequence, the special index 24 in the output space became overloaded to also represent removal actions, which is a suboptimal but empirically functional design choice.

B. Neural Network Architecture

The encoded input information is fed into separate input layers. The positional information of the game pieces is flattened beforehand, as the chosen architecture is a simple feed-forward network that does not provide inherent support for structured or sequential data.

After processing in the respective input layers, the resulting outputs are concatenated and subsequently passed through two hidden layers. Each hidden layer consists of 256 neurons. Since no well-founded prior knowledge regarding the optimal dimensionality of the feature representations was available, the number of neurons was chosen heuristically.

In the final processing step, a policy head is defined that outputs logits. These logits can then be interpreted by the action mapper to derive concrete actions. In addition, a separate value head was implemented for training purposes. This head serves to approximate the state value or, respectively, the advantage of the current game state. Accordingly, the output value should be close to 1 when the model is in a near-winning situation and close to -1 when a loss is imminent.

The value head first reduces the representation via an additional hidden layer with 128 neurons and then projects it onto a scalar output. The hyperbolic tangent function (tanh) is used as the activation function to explicitly constrain the output range to the interval $[-1, 1]$. For all remaining layers, the ReLU activation function is employed.

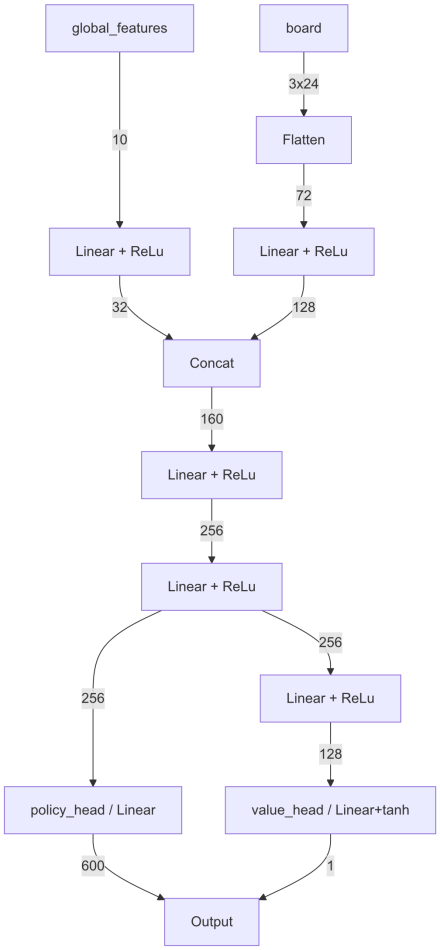

Fig. 2. Model Architektur

a) *Possible Model Architecture Improvements:* Currently, the model is implemented as a simple feedforward network. However, since the game of Nine Men’s Morris is essentially a graph and capturing graph-based relationships may contribute to a better modeling of the game structure, the use of a Graph Neural Network (GNN) could be advantageous.

b) *Feed Forward Neural Network:* A feedforward neural network is an artificial neural network in which information processing is strictly unidirectional. The input values are multiplied by weighting factors layer by layer and transformed into output values.

The feedforward structure is a fundamental prerequisite for the application of the backpropagation algorithm and was one of the first network architectures to be implemented. [13]

A deliberately simple neural network architecture was chosen to demonstrate the fundamental capabilities of neural

networks and to assess the minimum architectural complexity required to achieve reasonable performance, without relying on more elaborate or externally proposed model designs.

c) *Multi-layer perceptron:* A multilayer perceptron (MLP) refers to a class of modern feedforward neural networks (as explained above) consisting of fully connected neurons organized into multiple layers and employing nonlinear activation functions. A defining characteristic of MLPs is their ability to model and classify data that are not linearly separable.

Multilayer perceptrons represent a fundamental architecture in deep learning and are applied across a wide range of different domains. [14]

d) *Graph Neural Networks(GNN):* Graph Neural Networks (GNNs) are a specialized class of artificial neural networks designed for tasks in which the input data are represented in the form of graphs. [15]

C. Training

The training loop comprised several stages. It began with the agent playing a game and generating rewards at each step. Illegal moves were masked and penalized, while positive rewards were assigned for actions such as blocking mills, winning, gaining a positional advantage, forming mills (removing opponent pieces), and creating potential mills. Experience was accumulated until a predefined threshold was reached, at which point backpropagation was performed and the updated weights were applied. The techniques described below were employed to implement these training and reward mechanisms

a) *Reinforcement Learning:* Reinforcement Learning is a subfield of machine learning and optimal control that addresses the question of how an intelligent agent selects actions in a dynamic environment in order to maximize a long-term reward signal. In contrast to supervised and unsupervised learning, which are based on the analysis of labeled and unlabeled data respectively, learning in reinforcement learning occurs through direct interaction between the agent and its environment. The agent receives feedback in the form of rewards or penalties, on the basis of which it gradually learns an optimal action strategy (policy). [16]

b) *Self-play:* Self-play is a paradigm in which an agent plays against itself in order to improve its strategy. This allows us to train our Nine Men's Morris AI without using a dataset or requiring any manual intervention during training. [17]

c) *Actor-Critic:* The actor–critic algorithm represents a class of reinforcement learning methods that combine two central components: the actor, which selects actions, and the critic, which evaluates these decisions. This dual structure makes the agent's learning process more efficient, as decision-making and feedback are balanced. While the actor learns how to make decisions, the critic assesses their quality. This enables the agent to explore new actions while simultaneously building on previously acquired experience, resulting in a more stable and effective learning process. The policy corresponds to the actor, which selects actions, whereas the value function represents the critic, which evaluates the quality of these actions. [18]

d) *GAE:* Generalized Advantage Estimation (GAE) is used to compute the advantage function, providing a balance between bias and variance in policy gradient updates. By weighting temporal-difference errors over multiple time steps, GAE produces more stable and efficient learning signals. This allows the agent to learn more effectively from sparse or delayed rewards while reducing variance in the gradient estimates. [19]

e) *Entropy Regularization:* Entropy regularization is a method for regularizing model outputs, in which an additional term is added to the loss function that encourages the model to produce outputs with higher entropy. Entropy, in the sense of information theory, quantifies the uncertainty or randomness of a probability distribution. By maximizing the entropy of the output distribution, the model is encouraged to make more diverse and less overconfident predictions. [20]

f) *Reward Shaping:* Reward shaping is a method in which additional rewards are introduced to accelerate an agent's learning. The goal is to guide the agent step by step in the right direction, especially in situations where the original reward occurs very rarely or sparsely. [21] In our case, the agent is guided such that gaining a game advantage and closing mills are encouraged through positive rewards, while illegal moves receive negative feedback.

It should be noted that this constitutes one of the most important strategies for enhancing agent performance. In this setup, the agent is incentivized to maximize the rewards it receives rather than necessarily achieving victory in the game.

g) *Gradient Clipping:* Gradient clipping refers to a technique aimed at ensuring the stability of training neural networks by limiting the magnitude of the gradients. During the backpropagation process, gradients are computed, which determine the direction and strength of the weight updates in the network. However, if excessively large gradients occur —referred to as "gradient exploding"—the resulting weight changes can cause numerical instabilities, such as the occurrence of NaN values or overflow errors. Gradient clipping addresses this problem by constraining the norm of the gradients to a predefined maximum value. In this way, it ensures that the updates of the model parameters remain controlled without significantly limiting the network's learning capability, thereby contributing substantially to a stable and efficient training process. [22]

h) *Temperature Scaling:* Temperature scaling is used to calibrate the probabilities output by the policy network. A learnable temperature parameter T is applied to the softmax outputs, which dampens overly confident probabilities and represents uncertainties more realistically. In the game of Nine Men's Morris, this prevents the agent from favoring incorrect moves with excessive confidence, encourages the exploration of alternative strategies, and stabilizes training without altering the actual action decisions. [23]

i) *ε-Greedy Exploration:* The ε-greedy algorithm is a strategy that allows the agent to select actions in order to balance exploration (trying out new or infrequently chosen actions) and exploitation (leveraging already known, promising actions). With a probability of ε, the agent selects a random action (exploration), while with a probability of $1-\varepsilon$, it chooses the action with the highest estimated action value

(exploitation). In this way, the agent can both gain new insights and efficiently utilize known reward potentials. [24]

IV. Robot Execution Layer

A. Design Constraints and Integration Rationale

The robot execution layer was developed as part of a larger Sense–Plan–Act pipeline, making subsystem decoupling a primary objective. In particular, introducing additional vision dependencies into the execution layer would have increased integration complexity and risk, especially during system integration. Therefore, the execution component was designed to operate without a dedicated camera-based localization module beyond the perception subgroup.

The physical setup imposed further constraints. The employed game set exhibited noticeable manufacturing tolerances, including non-uniform chip heights, which complicate stable grasping. Although a magnetic end-effector can increase robustness under such variability, mechanical grasping was retained to ensure compatibility with standard, non-magnetic playing pieces. Consequently, the execution pipeline emphasizes (i) lightweight geometric calibration from a small set of reference measurements, (ii) a conservative approach-from-above motion primitive to reduce collision risk and stack disturbance, and (iii) empirical tuning of grasp parameters to improve robustness under piece-height variation.

B. Geometric Board Calibration and Target Pose Calculation

The vendor-provided `PyNiryo` API [25] was used as the primary control interface. The robot is commanded in task space using Cartesian end-effector poses (`x`, `y`, `z`, `roll`, `pitch`, `yaw`), represented in code via the `PoseObject` data structure [26]. Translational coordinates are specified in meters and orientations in radians. End-effector motion is executed in the robot's base coordinate frame using `move_pose(PoseObject)`.

The `get_pose()` function returns the current end-effector pose directly in this base frame. As a result, the pose parameters are immediately suitable for calibration, motion target generation, and execution, and no additional coordinate-frame conversion is required.

The first calibration point O serves as the origin. The second and third points, B and C, define the board's width and height directions, respectively. Two vectors are computed from the origin, $w = B - O$ and $h = C - O$. Since the relative layout of Nine Men's Morris is known and fixed, each playable position can be expressed by predetermined scalar coefficients (u_i, v_i) indicating how far to move along (i.e., how strongly to scale) the width and height vectors [27]. The target position for board index i is then computed as

$$P_i = O + u_i w + v_i h, \tag{3}$$

where u_i and v_i are taken from a lookup table.

In our implementation, the mapping is applied in the x–y plane only, and the height is kept constant by setting $z := O_z$. The value O_z is determined during calibration, either as an average height derived from the recorded reference points (e.g., using their mean) or as a predefined constant specified in `niryo_config.toml` for even more reliability if the board height is known.

C. Robust Motion Execution via a Hover–Descend–Retreat Pattern

```python
# from_idx/to_idx: int in [0..23] for board fields, or None for stack
def move(from_idx: int | None, to_idx: int | None):
  assert from_idx is None or 0 <= from_idx < 24
  assert to_idx is None or 0 <= to_idx < 24
  src = pose_stack("unplaced") if from_idx is None else pose_board(from_idx)
  dst = pose_stack("removed") if to_idx is None else pose_board(to_idx)

  def via_above(p, action):
    robot.move_pose(p.at(z = SAFE_Z)) # hover
    robot.move_pose(p) # descend to target
    action() #grasp/release
    robot.move_pose(p.at(z = SAFE_Z)) # retreat

  robot.release_with_tool()
  via_above(src, robot.grasp_with_tool)
  unplaced_count -= (from_idx is None)
  via_above(dst, robot.release_with_tool)
  removed_count += (to_idx is None)

  if back_to_idle: robot.move_pose(IDLE_POSE)
```

Listing 1. Approach-from-above pick-and-place routine (pseudocode).

The executed motion strategy is summarized in Listing 1 as pseudocode. Each manipulation step follows a fixed four-phase primitive (hover → descend → actuate → retreat) using a predefined safe height (equal to the configured idle pose height). This design reduces collision risk and improves repeatability, since horizontal movements are performed only at a clearance height. In particular, the approach-from-above strategy avoids sweeping motions near the board surface and reduces the likelihood of disturbing adjacent pieces or destabilizing stacks during pick-and-place operations.

D. Grasp Parameterization and Empirical Validation

Reliable manipulation was challenged by non-uniform chip heights originating from manufacturing tolerances of the physical game set. To mitigate sensitivity to such variation in chip-height without introducing additional perception dependencies, grasping was treated as a parameterized routine with configuration values stored in `niryo_config.toml`. The most relevant parameters include the nominal chip height (`chip_height`), an optional fixed board height (`fixed_z`), and small vertical clearance offsets for pickup and placement (`pick_z_offset`, `place_z_offset`). These offsets implement controlled over-travel during pickup and a gentler approach during placement, improving tolerance to piece-height variability.

Parameter settings were validated using a dedicated stress-test routine (`stresstest.py`) designed to exercise the

pick-and-place pipeline under representative operating conditions. The test executes a fixed sequence of 20 `ned2.move` operations, covering (i) repeated stack-to-board placements, (ii) repeated board-to-stack removals up to the maximum expected stack height, and (iii) additional board-to-board transfers for coverage. Throughout the procedure, the end-effector follows the approach-from-above motion primitive (Listing 1), thereby restricting lateral motion to a predefined clearance height. The routine was used iteratively to tune the grasp offsets and the stack-height compensation (parameterized by `chip_height`) until stable pickup and placement were achieved without dropping pieces or destabilizing stacks.

References

[1] Ultralytics, "Introducing Ultralytics YOLOv8." Accessed: Jan. 11, 2026. [Online]. Available: https://www.ultralytics.com/blog/introducing-ultralytics-yolov8

[2] Ultralytics, "YOLOv9 vs. YOLOv8: Architecture, Performance, and Applications." Accessed: Jan. 11, 2026. [Online]. Available: https://docs.ultralytics.com/compare/yolov9-vs-yolov8/

[3] Ultralytics, "Backbone." Accessed: Jan. 11, 2026. [Online]. Available: https://www.ultralytics.com/glossary/backbone

[4] Ultralytics, "Detection Head." Accessed: Jan. 11, 2026. [Online]. Available: https://www.ultralytics.com/glossary/detection-head

[5] J. Prakash, "Non Maximum Suppression: Theory and Implementation in PyTorch." Accessed: Jan. 12, 2026. [Online]. Available: https://learnopencv.com/non-maximum-suppression-theory-and-implementation-in-pytorch/

[6] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*, Second. Cambridge University Press, 2004.

[7] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. Springer Cham, 2022. doi: 10.1007/978-3-030-34372-9.

[8] 16-385 Computer Vision, Carnegie Mellon University, "Image homographies (Lecture 8 slides)." Accessed: Jan. 26, 2026. [Online]. Available: https://16385.courses.cs.cmu.edu/spring2024content/lectures/08_homographies/08_homographies_slides.pdf

[9] OpenCV, "Geometric Image Transformations." Accessed: Jan. 26, 2026. [Online]. Available: https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html

[10] OpenCV, "Geometric Transformations of Images." Accessed: Jan. 26, 2026. [Online]. Available: https://docs.opencv.org/4.x/da/d6e/tutorial_py_geometric_transformations.html

[11] Ultralytics, "Python Usage." Accessed: Jan. 26, 2026. [Online]. Available: https://docs.ultralytics.com/usage/python/

[12] Ultralytics, "Reference for ultralytics/engine/results.py." Accessed: Jan. 26, 2026. [Online]. Available: https://docs.ultralytics.com/reference/engine/results/

[13] "Feedforward neural network." Accessed: Jan. 14, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Feedforward_neural_network

[14] "Multilayer perceptron." Accessed: Jan. 14, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Multilayer_perceptron

[15] "Graph neural network." Accessed: Jan. 14, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Graph_neural_network

[16] "Reinforcement learning." Accessed: Jan. 14, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Reinforcement_learning

[17] "Self-play." Accessed: Jan. 14, 2026. [Online]. Available: https://en.wikipedia.org/wiki/Self-play

[18] "Actor-Critic Algorithm in Reinforcement Learning." Accessed: Jan. 14, 2026. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/actor-critic-algorithm-in-reinforcement-learning/

[19] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation." [Online]. Available: https://arxiv.org/abs/1506.02438

[20] "Definition and Significance of Entropy Regularization." Accessed: Jan. 14, 2026. [Online]. Available: https://www.numberanalytics.com/blog/entropy-regularization-machine-learning

[21] "Epsilon-Greedy Algorithm in Reinforcement Learning." Accessed: Jan. 14, 2026. [Online]. Available: https://rljclub.github.io/posts/reward-shaping/

[22] "Understanding Gradient Clipping." Accessed: Jan. 14, 2026. [Online]. Available: https://www.geeksforgeeks.org/deep-learning/understanding-gradient-clipping/

[23] "What Is Temperature Scaling for LLM Calibration?." Accessed: Jan. 14, 2026. [Online]. Available: https://markaicode.com/temperature-scaling-calibrate-llm-confidence-scores/

[24] "Epsilon-Greedy Algorithm in Reinforcement Learning." Accessed: Jan. 14, 2026. [Online]. Available: https://www.geeksforgeeks.org/machine-learning/epsilon-greedy-algorithm-in-reinforcement-learning/

[25] "PyNiryo 1.2.3 Documentation." Accessed: Jan. 26, 2026. [Online]. Available: https://niryorobotics.github.io/pyniryo/v1.2.3/index.html

[26] "PyNiryo 1.2.3 Examples: Movement." Accessed: Jan. 26, 2026. [Online]. Available: https://niryorobotics.github.io/pyniryo/v1.2.3/examples/examples_movement.html

[27] "Scalar (Mathematics)." Accessed: Jan. 26, 2026. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Scalar_(mathematics)