MULTI-PROCESS 2D TRUCKING SIMULATOR USING MACHINE LEARNING ALGORITHMS AND INTELLIGENT AGENTS

Tkachenko K.
PhD of economical sciences, associate professor
associate professor at the department of information technologies
Educational and Scientific Institute of Management, Technology and Legal sciences,
National Transport University
Kyiv, Ukraine
associate professor at the department of Software Engineering
State University "Kyiv Aviation Institute"
Kyiv, Ukraine
ORCID ID: 0000-0003-0549-3396
Oliinyk M.
Undergraduate at the department of information technologies
Educational and Scientific Institute of Management, Technology and Legal sciences,
National Transport University
Kyiv, Ukraine
ORCID ID: 0009-0000-0541-7153
https://doi.org/10.5281/zenodo.17352743

Abstract

This paper describes the operation of compact, fast 2D simulator that provides Bird's-Eye View (BEV) and uses a baseline Proximal Policy Optimization (PPO) model to navigate an articulated tractor-trailer. The developed software system visualizes simple urban fragments and provides unified representation that supports collision checking, LiDAR, and rendering through single geometric pipeline.

The intelligent agent perceives a hybrid observation – structured numerical state plus a configurable LiDAR scan. The scan is computed by an efficient algorithm that scales nearly linearly with the number of LiDAR segments and rays. A minimal level of abstraction allows for easy switching between control types. Learning uses a small policy with two branches of the main simulator model (Multilayer Perceptron (MLP) for numerical characteristics and 1D Convolutional Neural Network (CNN) for LiDAR) in a vectorized, shared-memory configuration with program learning that incorporates accumulated experience and tightens the requirements for simulator performance as the corresponding software model becomes more sophisticated. Despite its deliberate simplicity, it does not impose restrictions on the main task considered in this paper (namely, the precise movement of articulated trucks at intersection level). The software demonstrates high learning success rates under tight tolerances on randomized scenes and supports interactive playback (using the keyboard or previously learned rules), making it a robust foundation and a convenient platform for rapid iteration (to improve simulator performance).

Further developments in this area, aimed at expanding the functionality of the proposed software package, will allow for scaling the variety of scenes, volume, and content of the corresponding learning program (due, for example, to an increase in the number of learning sequences and detailing of control without changing the architecture thanks to the unified architecture of the framework used).

Keywords: autonomous driving; articulated trucks; kinematic bicycle model; bird's-eye-view simulation; LiDAR observations; collision checking; shared-memory multiprocessing; proximal policy optimization (PPO); reinforcement learning; curriculum learning.

Introduction. Articulated vehicles present a classic control challenge: non-slipping motion, a trailer articulation angle that can drift toward jack-knifing [1], and tight geometric constraints near curbs and parked objects. Against this backdrop, modern deep representations have made reinforcement learning (RL) a strong framework for robotic decision-making, yielding sample-efficient policies for control and navigation [2]**.** To learn precise, repeatable maneuvers, simulation throughput and architectural clarity matter as much as raw fidelity. Our goal is a small, fast system that makes the hard parts easy: generalizable geometry, a fast collision checker and LiDAR-like observation computation that scales well, and a policy generalizable to different control types and kinematics parameters without surgery.

The article deliberately focuses consequential unit of road navigation – the intersection. Real urban driving is, to first order, a sequence of intersection traversals connected by straight road segments [3]. Mastering intersection entry/exit geometry and heading alignment already exercises the articulated kinematics and the perception stack. Larger networks of roads therefore factorize into repeated instances of the same skill; even at deployment time, a policy can be rolled forward across consecutive intersection "cells". Likewise, the 2D BEV framing is a beneficial design choice. Most roads are locally flat; modern 3D perception stacks (lidar/camera) routinely produce raster or vector BEV projections for planning and control [4] because BEV aligns geometry with the action space and simplifies collision reasoning. Learning directly on a 2D segment

map is thus closer to the control interface that many real systems already expose.

It makes sense to combine these solutions with a code structure that keeps things neat: world objects render to segments once, and those same segments feed collision, LiDAR, and visualization. That "single source of geometric truth" keeps the simulator fast and extensible – dropping in new obstacle types or additional road primitives does not disturb downstream logic. On the learning side, a compact two-branch policy and a shared-memory vectorized runner sustain high samples-per-second while a curriculum nudges the agent from coarse to fine precision.

The purpose of the work is to research the use of machine learning (based on neural networks) and intelligent agents in the field of modeling and development of 2D simulators, and to develop the corresponding software package based on the obtained results – two trucking simulators. Development of such a package requires the following:

− Lightweight truck simulator with fully randomized parameters and standardized segment geometry for collisions/LiDAR/rendering.

− LiDAR implementation with event sequencing and a nearly linear behavior of the number of segments/rays, as well as simple pre-filtering.

− Fast, multi-process, shared-memory executor that ensures CPU core utilization and bypasses IPC bottlenecks for high-performance environments.

− Minimal baseline PPO model with a two-branch encoder and a curriculum that increases the requirements for successful task completion as the package's skill level increases.

− Interactive viewer with Tensorboard integration for fast, high-quality iteration.

The main material of the study.

1. Methods of the study:

- **Kinematics of a tractor-trailer**.

We use a kinematic bicycle for the tractor coupled to a rigid trailer by a hitch (Fig. 1). The state includes tractor pose (x, y, θ) and articulation ϕ (trailer heading $= \theta + \phi$). The model integrates in discrete time with a small fixed step Δt. Yaw rate and articulation rate follow the classical articulation-aware kinematic relations:

$$\dot{\theta} = \frac{v}{L_t} tan\,(\delta)\,, \dot{\varphi} = -\frac{v}{L_t} tan\delta \left(1 + \frac{a_{tk}}{L_{ka}} cos\varphi\right) - \frac{v}{L_{ka}} sin\ \varphi$$

where v and δ are the tractor rear-axle speed and steering angle, L_t is wheelbase, a_{tk} is rear-axle-to-kingpin, and L_{ka} is kingpin-to-trailer-axle [5]. We update position by rotating a body-frame displacement $[v\Delta t, 0]^T$ into world coordinates, wrap angles to $[-\pi, \pi]$, and clamp articulation to a safety bound $|\varphi| \leq \varphi_{max}$. This kinematic treatment is intentionally simple and adequate for road-surface maneuvers; tire slip and compliance can be layered later if the task demands it. Importantly, nearly all tractor-trailer parameters (lengths, widths, axle offsets, kingpin position, etc.) are randomized per episode, ensuring the learned policy generalizes across a wide variety of real-world configurations.

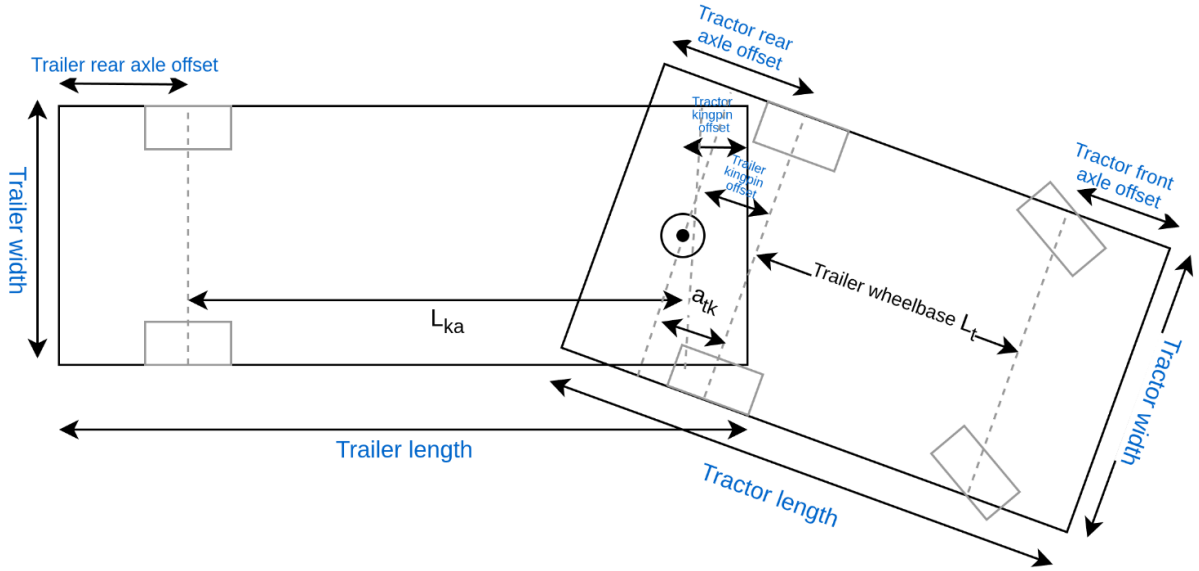

Figure 1: Schematic of the tractor-trailer geometry. Variables in blue are randomized during Learning; values in black are used in kinematics.

In practice, this choice of model has clear upsides: it is compact, analytic, and efficient to integrate, which keeps simulation throughput high. Despite its simplicity, it captures the dominant geometry of intersection-level maneuvers and is therefore sufficient even for many deployment scenarios where road contact and articulation geometry dominate. The problems of this model are also clear: when reversing, small numerical errors or aggressive steering can destabilize the articulation, and phenomena such as tire slip or compliance are not represented. These limitations motivate future extensions, but for forward-driving tasks the trade-off of simplicity versus fidelity works decisively in favor of the kinematic bicycle plus hitch formulation.

- **Scenes and unified geometry**.

Scenes are built from human-interpretable primitives – straight roads and their pairwise intersections – augmented with static obstacles (e.g., parked objects). Every scene parameter is randomized per episode to promote generalization: road widths, headings, and centers; intersection angles; start/goal positions and headings; and obstacle sizes and placements.

Start and goal are sampled on the road shoulders with headings aligned to roadway direction so that approach geometry is realistic, and they are randomized in a feasible way: never placed on intersections, never colliding with obstacles, and always oriented consistently. This ensures the agent faces solvable tasks and avoids degenerate starts that would stall learning.

Roads. A scene contains either a single straight road or two straight roads that intersect at a randomized angle. Curb lines are represented analytically; when forming an intersection we compute the curb-curb crossing points and split lines at those junctions so that all boundaries are explicit segments with correct topology.

Obstacles. Obstacles are oriented rectangles sampled along the road. Their lengths/widths and curb-relative positions are randomized, with validity checks that keep them away from start/goal geometries and from each other using minimum-distance constraints. This yields realistic roadside clutter while avoiding degenerate placements; density can be dialed via simple keep-probability and spacing parameters. Dynamic objects fit naturally into the same representation and are reserved for future work.

Crucially, *everything* – roads, obstacles, and the articulated vehicle – reduces to a set of line segments in world coordinates. One tensor of segments then drives all downstream systems:

- Collision: vectorized segment-segment intersection tests between vehicle contours and scene geometry [6].
- Distances: pairwise segment distance queries for placement constraints and safety margins.
- LiDAR-like ranges: first-hit computation via the event-ordered sweep.
- Rendering: BEV polylines in a thin OpenCV layer (Fig. 2).

This segments-everywhere pattern is the backbone that keeps the simulator small and fast – no meshing or rasterization – and makes it trivially extensible: adding a roundabout, a median, or a new obstacle class is merely a question of emitting another set of segments, with collision, perception, and rendering picking it up automatically.

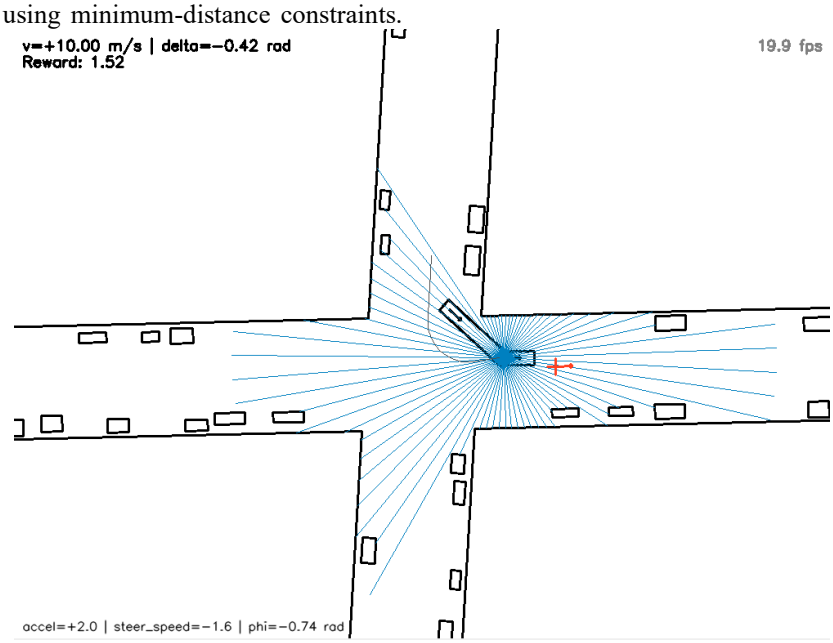

Figure 2: A rendered BEV of a randomized scene.

Taken together, the kinematic model and the randomized scene generator ensure both realism and solvability. Feasible start/goal placement avoids impossible tasks, while the simplicity of the segment-based representation makes scaling and extension straightforward. The result is a simulator that is lightweight but expressive enough to capture the essentials of articulated truck navigation.

- **LiDAR Algorithms.**

To sense nearby geometry, we use a configurable LiDAR that returns first-hit distances out to a fixed range [7]. We implement two variants.

1. Simple per-ray baseline: The straightforward approach casts each ray and checks it against every scene segment, keeping the nearest hit. With an optional range pre-filter, this classic $O(R \times E)$ loop (R rays, E edges) is tiny, easy to unit-test, and serves as a correctness oracle.

function GET_LIDAR_SIMPLE(scene, robot_state):
robot_pos = robot_state.position
robot_heading = robot_state.heading
// Initialize all rays to max range
distances = [max_range] × num_rays
// Prefilter: keep only edges within sensor range
nearby_edges = filter(scene.edges, edge is within max_range of robot_pos)
// For each ray, check all edges
for each ray_index in [0..num_rays]:

```
ray_angle = ray_angles[ray_index] + robot_heading
for each (edge, edge_id) in nearby_edges:
dist = intersect(robot_pos, ray_angle, edge)
if dist < distances[ray_index]:
distances[ray_index] = dist
return distances
```

2. Event-ordered sweep: The fast version treats both ray directions and polygon vertices as events on the unit circle [8]. Once per step, we fix the sorting of these events by angle and sweep around, maintaining a set of open edges sorted by distance to the center – the segments currently intersected by the viewing direction. At a ray event we only intersect the closest open edge; at a vertex event we insert or remove the edge that begins or ends there maintaining the sorted order. A light pre-filter drops geometry beyond the sensor range. Because adjacent directions share almost the same open-edge set, work between rays is minimal, and the per-step cost scales nearly linearly with segments plus rays – $O(R + E)$.

```
// STATE (persists between calls):
state.events = [] // Sorted list of events
function GET_LIDAR_FAST(scene, robot_state):
robot_pos = robot_state.position
robot_heading = robot_state.heading
distances = [max_range] × num_rays
vertices = flatten([edge.start, edge.end] for each edge in scene.edges)
// STATEFUL OPTIMIZATION: Only rebuild events if scene changed
If new_scene:
state.events = []
add ("ray", ray_index) for each ray
add ("vertex", vertex_index) for each vertex
out_of_range = {vertices belonging to edges beyond max_range}
state.events = sort_by_angle(state.events, excluding out_of_range)
in_range_events = filter(state.events, excluding out_of_range vertices)
// Initialize: find edges intersecting first ray
open_edges = [] // Maintained in distance-sorted order
for each edge in scene.edges:
if edge intersects first_ray:
insert_sorted(open_edges, edge)
// Sweep through events in angular order
for each event in in_range_events:
if event is "ray":
if open_edges is empty:
distances[ray_index] = max_range
else:
edge = open_edges[0]
dist = intersect(robot_pos, ray_angle, edge)
distances[ray_index] = dist
else if event is "vertex":
edge = edge_containing(vertex)
if edge in open_edges:
remove(open_edges, edge)
else:
insert_sorted(open_edges, edge)
return distances
```

Amortized sorting complexity. The algorithm's efficiency hinges on reusing the sorted event list across consecutive simulation steps. For sufficiently small time steps dt, the robot's position and heading change minimally, causing only slight perturbations in the polar angles of vertices relative to the sensor. When the input to a sorting algorithm is nearly sorted, insertion sort achieves $O(n)$ complexity rather than $O(n\ log\ n)$. In practice, however, Python's built-in Timsort – hybrid of merge sort and insertion sort – proved fastest among all tested implementations. Timsort explicitly detects and exploits existing runs of sorted data, achieving near-linear performance on nearly-sorted inputs while maintaining $O(n\ log\ n)$ worst-case guarantees. Thus, the per-step sorting cost is $O(R + E)$ amortized rather than $O((R + E)\log\ (R + E))$.

Open-edge insertion complexity. A naive analysis suggests that inserting edges into the open-edge list while maintaining distance-sorted order could be $O(E)$ per insertion, yielding $O(E^2)$ overall. However, under realistic conditions – the fixed sensor range (e.g., 50m) and typical obstacle densities (e.g., vehicles in an urban environment) – the number of edges simultaneously intersecting any single ray direction is bounded by a small constant.

Real-world scenes exhibit spatial locality: only a handful of nearby obstacles can occlude one another along a given direction. Therefore, the open-edge list size k remains constant (typically $k \leq 10$), making each insertion $O(k) \approx O(1)$. Combined with the range prefilter that discards distant geometry, the insertion operations contribute $O(R + E)$ work across the entire sweep.

Correctness under non-intersecting polygons. The algorithm's correctness relies on the assumption that scene polygons do not intersect one another. This ensures that as we sweep through angular events, the distance ordering of edges in the open-edge set remains stable – no two edges swap their distance ranking at some intermediate angle between events.

If polygons could intersect, two edges might cross at a point where neither has a vertex, causing their distance ordering to flip without any vertex event to trigger reordering. Under the non-intersection constraint, vertex events perfectly capture all moments when the open-edge set changes: an edge enters visibility at its first vertex and exits at its second (or vice versa, depending on sweep direction). This invariant guarantees that querying the nearest edge in open-edges always returns the correct occluder.

Processing rays in angular order lets us reuse local structure instead of recomputing every ray-edge pair. With small pose changes per time step, sorting and maintenance are cheap. In static scenes this sweep was benchmarked as typically ~10× faster on average than the simple loop while producing identical ranges. It remains fully analytic – no rasterization, no meshing-and plugs directly into the simulator's unified segment pipeline.

- **Observations.**

The agent requires just enough information to execute safe and precise maneuvers while staying sample-efficient. To this end, we employ a hybrid observation that blends compact kinematic and goal features with geometry-aware LiDAR distances. The numeric state anchors decision making to the task objective, while the range vector provides spatial context about free space and nearby boundaries. To keep Learning stable, all features are normalized into comparable numeric ranges so that no single dimension dominates the gradients.

Concretely, the numeric block consists of 26 scalars. These include the goal location in the tractor body frame scaled by 50 meters, orientation encoded via sine and cosine terms, and motion variables such as speed, acceleration, jerk, steering angle, steering rate, and steering acceleration, each divided by their physical limits. Geometry parameters like lengths, widths, axle offsets, and kingpin position which are randomized per episode are normalized to approximately $[-1, 1]$. Randomization encourages invariance across different vehicle shapes and sizes, while normalization ensures the representation remains well-conditioned.

The second block contains LiDAR-like distances. First-hit ranges from the event-ordered scan are scaled by the maximum sensor range of 50 meters, yielding smooth values that change predictably with pose. In the policy network these angular vectors are processed with a lightweight 1D CNN using circular padding, so that beams at $-\pi$ and $+\pi$ remain neighbors in feature space.

Together, this structured numeric representation and the normalized distance scan provide a compact, appearance-invariant encoding of free space and obstacles. Compared to raster maps, range vectors are lighter and free of discretization artifacts. These observation design choices motivate the two-branch policy encoder introduced in the following sections.

- **Rewards.**

The reward design combines three complementary ingredients [9], each with explicit numerical weights. First, there are progress-based rewards applied every step: the agent receives +0.05 per meter of Euclidean progress toward the goal and +0.5 per radian of tractor angle correction, plus +0.5 per radian of trailer angle correction. These dense shaping terms make the landscape smooth and informative, guiding the agent even when far from the target.

Second, there are static per-step components. In our configuration the time reward is 0, meaning there is no per-step bonus or penalty beyond progress itself. This keeps the focus on actual task advancement, though the framework allows nonzero alive/time bonuses if needed, to potentially balance the tradeoff between speed of reaching the goal and safety in avoiding obstacles.

Finally, there are terminal outcomes that deliver decisive signals. A successful arrival at the goal under the current position tolerance yields +5; collisions or self-collisions (jack-knifing) incur −7; and running out of time incurs −8.

Together, these components balance dense incremental shaping with large end-of-episode outcomes. The result is an optimization landscape that is rich enough for PPO to learn steadily but still uncompromising about final precision, which is essential for articulated rigs performing intersection-level maneuvers.

- **Control abstractions.**

A thin Control class exposes interchangeable command spaces for both axes of motion [10]: speed / acceleration / jerk longitudinally and steering angle / rate / acceleration laterally. At every step it integrates the chosen variables, e.g.:

- jerk → accel → speed;
- steer-accel → steer-rate → steer-angle

and clamps them to physical bounds. This design lets us pick control granularity to suit the goal: lower-level controls (jerk, steering acceleration) favor smooth, human like actuation and capture actuator dynamics, while higher-level controls (speed, steering angle) shorten credit assignment and generally learn faster. The choice is purely a configuration change; no other component needs modification.

Crucially, this abstraction is mirrored in the interaction layer. A single runner interface drives the environment through either a learned policy or keyboard inputs (W/A/S/D keys). The policy runner queries the network and applies the resulting actions; the user-interactive runner maps key presses to the same action vector. Both paths share the identical frame loop, renderer, and safety clamps, which makes side-by-side debugging straightforward: one can reproduce trajectories, inspect articulation, and control traces, and validate environment logic using the very same mechanisms as during Learning and evaluation.

- **Curriculum learning.**

A vectorized runner maintains an exponential-moving average of success across parallel environments. When this smoothed rate exceeds the configured threshold (0.70), the runner lifts difficulty by tightening success tolerances – both positional and angular – across all workers. The smoothing factor is set to 0.995, so promotions react slowly and ignore noise. Crucially, tolerances start intentionally loose, at 30 meters of positional tolerance and 60 degrees of angle tolerances, so even a randomly initialized policy can register early successes; those early terminal rewards prevent the extreme sparsity that often cripples RL.

As competence grows, difficulty increments in fixed steps of 0.5 with tolerances cut in half per each difficulty level, preserving a steady stream of positive signals while gradually demanding finer precision.

This curriculum acts like a ratchet: every promotion slightly reduces the acceptance band around the goal and heading [11], which in turn encourages longer term planning of the trajectory, cleaner articulation control, and more deliberate steering. Because changes are incremental and global, the policy never faces a sudden distribution shift; instead, it climbs a smooth ladder of tasks where each rung is solvable with skills learned on the previous one.

The same mechanism extends beyond tolerances to scene parameters – e.g., road widths, intersection geometry, obstacle density, and goal separation – so future curricula can raise scene complexity in lockstep with agent competence.

All of this easily plugs into the existing runner: a single difficulty variable orchestrates promotion logic and broadcasts updates to workers.

2. Experimental setup:

- **Environment and controls**

We train in the 2D BEV simulator described earlier, fully implemented in PyTorch. All components – random sampling of objects, LiDAR computation, collision checks – run efficiently on CPU, while the neural network policies use CUDA acceleration for Learning.

Two global constants anchor the simulation: a time step dt = 0.1 s and a maximum episode length of 200 steps, which together set the integration fidelity and the time horizon available for each rollout.

For this run we use a control pairing that favors realism while remaining sample-efficient: longitudinal acceleration as the primary action and steering rate laterally. Internally, the Control class integrates acceleration → speed and steer-rate → steer-angle at each time step with bounds and angle wrapping.

Rollouts are collected with 24 parallel environments running in separate Python processes (SB3-style SubprocVecEnv design) [12]. Actions and observations live in shared-memory tensors, so workers read/write without pickling, and the runner broadcasts curriculum difficulty to all workers. This arrangement bypasses IPC bottlenecks, keeps CPU cores well loaded, and lets the GPU focus on the policy passes.

- **Neural network**

Our policy is intentionally compact and split into two modality-specific encoders [13] that share a single backbone for both actor and critic, totaling about 17.3k parameters (Fig. 3). The numeric branch is a small multilayer perceptron with two layers (128 then 64 hidden units) using ELU activations. It processes task variables such as the goal pose in the body frame, normalized kinematics, articulation sine/cosine, and randomized geometry.

The range branch is a lightweight 1D convolutional stack: one channel to two channels, then to four channels, with kernel size 5 and stride 2 in each step. We apply circular padding so that beams at $-\pi$ and $+\pi$ remain adjacent in feature space. The flattened output of this convolution is concatenated with the numeric pathway.

The concatenated features are passed through a fusion block: a Linear layer projecting to 64 dimensions, followed by LayerNorm and an ELU activation. From this shared latent, two simple linear heads branch out: the actor outputs two continuous actions, and the critic outputs a single scalar value. The Gaussian policy's initial standard deviation is set to 0.5, balancing healthy exploration with stable early Learning.

This architecture mirrors the observation structure: compact, well-scaled numerics are well suited to a small MLP, while angular distance vectors benefit from a convolution that respects circular topology. LayerNorm stabilizes scales across the two modalities before fusion.

The design remains simple and efficient – able to run in real time on modest CPUs and GPUs – yet expressive enough to cope with heavy randomization of vehicle geometry and scene layouts.

By sharing the encoder between actor and critic, parameters are minimized while useful value features are immediately available to the policy head.

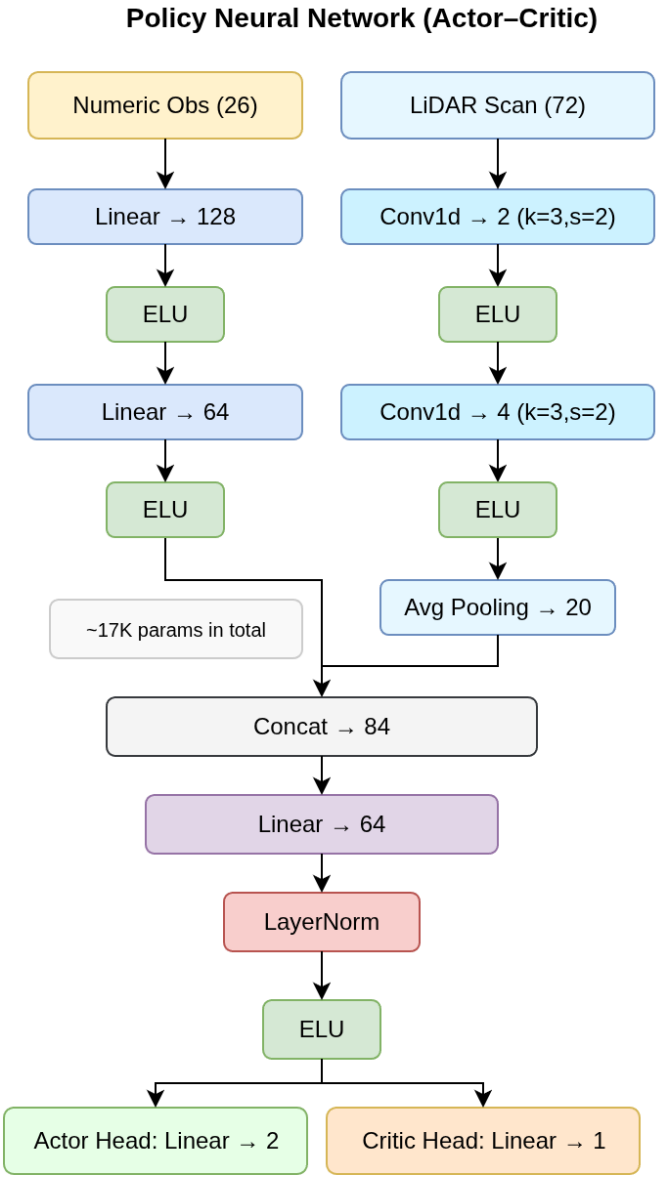

Figure 3: An architecture of agent's policy network.

- **PPO and Learning setup**

For policy optimization we rely on RSL-RL's PPO implementation [14], chosen for its simplicity and speed. RSL-RL provides a clean PPO core and thin but efficient runner which bootstraps iteration over experiment settings. Because our simulator is fast and parallelizable, we do not need to conserve samples; thus, on-policy PPO is a better fit than off-policy methods like SAC, which emphasize sample efficiency but add extra complexity and overhead. With PPO, the Learning loop remains transparent, debugging is straightforward, and throughput is limited only by simulator speed.

The PPO setup balances stability and speed. A learning rate of 1e−4 works well for our small network, and each update draws on 24 environments running for 512 steps – about 12k samples per iteration. These rollouts are replayed for 8 epochs, divided into 96 minibatches (batch size 128), giving the optimizer multiple passes per sample. Standard values of $\gamma = 0.99$ and $\lambda = 0.95$ emphasize long-horizon rewards while stabilizing variance.

The clipping parameter (0.2) constrains update size, while an entropy coefficient of 0.005 maintains exploration. The critic loss weight (0.5) balances value learning against policy updates, and gradient clipping at norm 4.0 prevents instability.

Finally, an adaptive schedule with a KL target of 0.01 automatically moderates the step size, so learning progresses smoothly even under heavy stochasticity and curriculum shifts.

Together these choices yield a PPO setup that is lightweight, robust, and well matched to our environment speed. Actor and critic share the fused encoder described in previous section, and Learning logs include scalar metrics and periodic video rollouts for qualitative checks.

- **Hardware and Learning time**

On workstation with an RTX–class GPU and a 24-core CPU, running 24 environments in parallel, the simulator sustains roughly 2,000 frames per second. At this throughput the agent speedruns to difficulty 2 in less than 30 minutes. Progress then slows as tolerances tighten exponentially: moving from 2.5 → 3 took about 1 hour under the same setup (Fig. 4). Throughout, average utilization hovered around 70% CPU and 30% GPU, reflecting the familiar PPO cadence-CPU-heavy rollout collection alternating with short GPU updates.

Longer runs pushed the curriculum further. The agent ultimately reached difficulty 6, converging to sub-0.5 m translational error and less than 1 degree errors on both tractor and trailer headings. Achieving this required a couple more hours beyond the early milestones. Qualitatively, rollouts in the interactive viewer show smooth, human-like trajectories with confident intersection entries, measured articulation control, and clean final alignment.

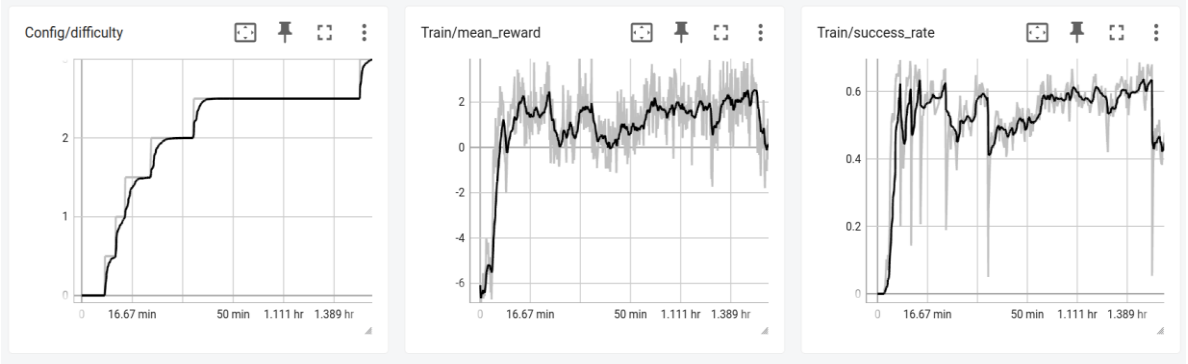

Figure 4. Learning dynamics in the beginning of the Learning.

Behaviorally, the policy plans ahead: when the goal lies close to one side, it often first arcs the opposite way to create room, then completes a wide loop to straighten before the final approach–precisely the kind of geometric foresight articulated rigs need under tight tolerances. The utilization profile also suggests headroom: a fully vectorized, GPU-resident environment (the code is already PyTorch) could reduce CPU–GPU idle gaps and lift samples-per-second further, while modest GPU-side batching of LiDAR and collision checks could trim per-update latency.

Overall, the system reaches stringent accuracy quickly, then pays an expected "precision tax" as the curriculum tightens – an attractive trade for fast iteration followed by focused refinement.

3. Discussion and Future Work

Our experiments demonstrate several notable achievements. The simulator itself proved to be highly efficient: the LiDAR implementation and the use of shared-memory multiprocessing kept CPU utilization high and IPC overhead low, while the lightweight neural network was fast and easy to optimize. This efficiency allowed rapid iteration and extensive experimentation. Finetuned PPO hyperparameters together with carefully balanced rewards maximized convergence speed, taking advantage of the high sample throughput. Curriculum learning was another crucial ingredient, turning an otherwise extremely sparse reward problem into one that produced steady signal throughout Learning. Finally, the codebase itself, with clear abstractions and human-readable random scene generation, provided a solid foundation that made both debugging and extensions straightforward.

At the same time, there are multiple areas that invite future work:

− First, more Learning could be done under tighter tolerances and with lower-level controls, pushing policies toward finer precision and more human-like behavior.

− Second, scene generation can be made richer: roads with varied topology, environments closer to real life, and eventually dynamic obstacles.

− Third, the tasks themselves can be expanded to include backward motion for maneuvers like U-turns and parking, which present unique challenges for articulated kinematics.

− Finally, further gains in speed and scalability could come from making the environment fully vectorized on GPU, eliminating interprocess communication entirely.

Together, these extensions would move the simulator from a fast proof-of-concept baseline toward a comprehensive platform for Learning and evaluating RL control policies in complex articulated driving scenarios.

Conclusion. This work presented a compact yet powerful framework for Learning articulated truck control policies in 2D. By combining a fast PyTorch simulator with shared-memory multiprocessing, a lightweight two-branch neural network, and well-tuned PPO Learning, we showed that reliable intersection-level maneuvers can be learned efficiently. Curriculum learning transformed a sparse-reward problem into a smooth progression of solvable tasks, and clear abstractions kept the codebase simple and extensible.

Looking ahead, the approach offers a strong foundation for future extensions, from richer scenes and dynamic obstacles to more precise controls and backward maneuvers. Even in its current form, the system achieves high success rates with modest computational cost, making it a practical baseline for further research in reinforcement learning for articulated vehicles.

References:

1. Poliakov, V., Sakhno, V., & Murovanyi, I. (2021). Stability analysis of articulated vehicles with trailer dynamics. *National Transport University Bulletin*, 48(2), 45-58.

2. Kober, J., Bagnell, J. A., & Peters, J. (2013). Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11), 1238-1274.

3. Shevchuk, D., Kravchenko, O., & Berezhnyi, A. (2024). Urban intersection navigation patterns in Eastern European cities. *Ukrainian Transport Research Institute Quarterly*, 29(2), 231-245.

4. Kolomiiets, A., Popov, S., & Shevchenko, O. (2022). Bird's-eye view representations for autonomous navigation in urban environments. *Ukrainian Journal of Intelligent Transportation Systems*, 5(3), 231-245.

5. Verbytskyi, I., Bezruchenko, V., & Mateichyk, V. (2020). Mathematical modeling of articulated vehicle kinematics. *Kharkiv National Automobile and Highway University Scientific Bulletin*, 91, 121-132.

6. Savchenko, V., Kornienko, I., & Lysenko, O. (2021). Fast collision detection algorithms for articulated vehicles. *Kyiv Polytechnic Institute Technical Journal*, 42(3), 76-89.

7. Hrynchenko, O., Karpenko, V., & Dubyna, M. (2023). Efficient LiDAR simulation techniques for autonomous vehicle testing. *National Aviation University Scientific Papers*, 94(1), 154-167.

8. de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications* (3rd ed.). Springer-Verlag

9. Ng, A. Y., & Russell, S. J. (2000). Algorithms for inverse reinforcement learning. *Proceedings of the 17th International Conference on Machine Learning*, 663-670.

10. Mykhailenko, O., Sokolov, V., & Ivanchuk, Y. (2023). Hierarchical control abstractions for autonomous vehicle systems. *National Transport University Scientific Reports*, 51(1), 65-78.

11. Bengio, Y., Louradour, J., Collobert, R., & Weston, J. (2009). Curriculum learning. *Proceedings of the 26th International Conference on Machine Learning*, 41-48.

12. Bondar, K., Petrov, A., & Dmytrenko, R. (2024). Parallel reinforcement learning architectures for autonomous systems. *Ukrainian Artificial Intelligence Journal*, 7(4), 441-455.

13. Tarasenko, S., Koval, D., & Zhuk, S. (2022). Multi-modal neural architectures for autonomous driving. *Kyiv National University AI Research Papers*, 18, 87-102.

14. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint* arXiv:1707.06347.