SortingEnv: An Extendable RL-Environment for an Industrial Sorting Process

Paper **link**: 2503.10466

Authors: Tom Maus, Nico Zengeler, Tobias Glasmachers

Github: SortEnv

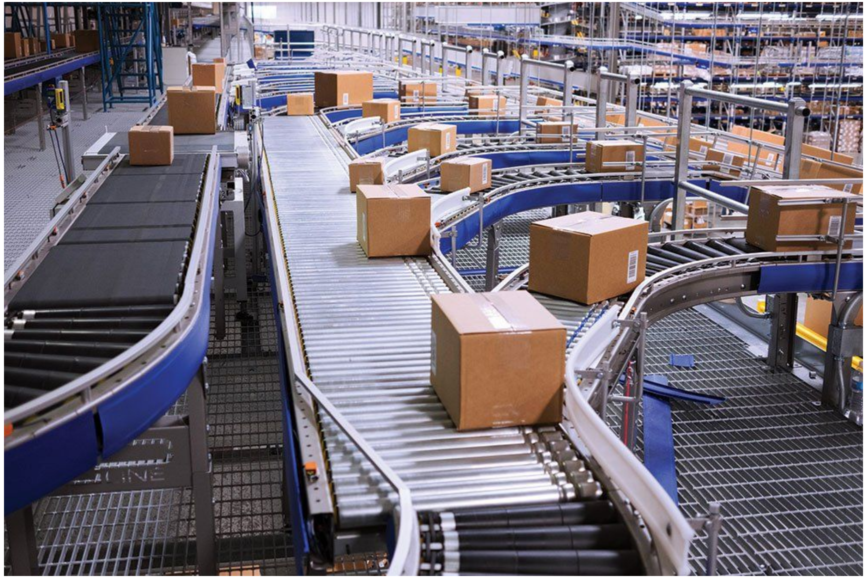
Group 4: Bruno Garofalo & Kerim Sever

Introduction and Motivation

Introduction and Motivation

Use Reinforcement Learning (RL) to solve complex industrial systems to optimize **material sorting** performance based on:

1. ***Belt speed***
2. ***Material occupancy***
3. ***Sorting accuracy***



Problem:
Industrial sorting systems are complex, dynamic, and require frequent grades (e.g., new sensors, machinery).

Challenge:
Traditional rule-based systems lack adaptability in real-time environments.

Why It Matters:
Inaccurate decisions in industrial sorting can lead to quality issues, inefficiencies, and increased costs.

Motivation for RL:
RL offers adaptive, trial-and-error-based optimization for dynamic environments.

RL Application and Overview

RL Application Overview

Application Context:
Simulates material flow in an industrial sorting line: Input → Conveyor Belt → Sorting Machine → Storage.

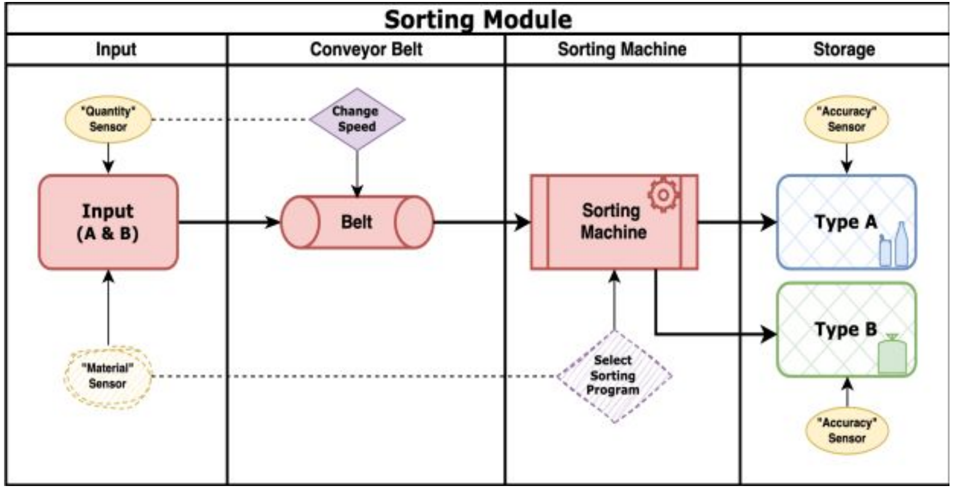
Goal:
Maximize belt speed and purity (accuracy of sorted materials).

Two Environments:

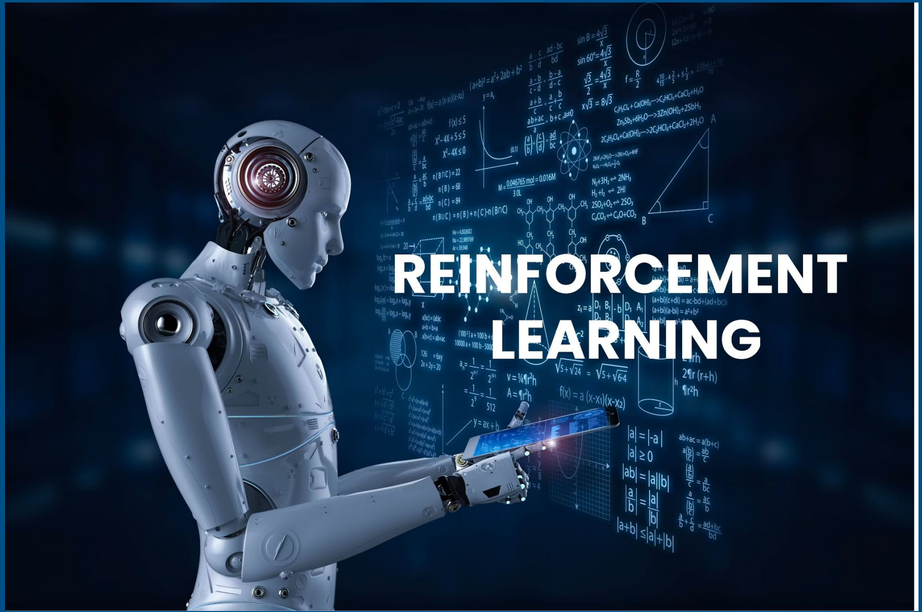
- **Basic:** Adjusts only belt speed
- **Advanced:** Adds sorting modes and material composition sensing

Why It's Challenging:

- High variability (input randomness, sensor noise)
- Real-time decision-making
- Balancing speed vs. sorting accuracy trade-offs



RL Model and Approach



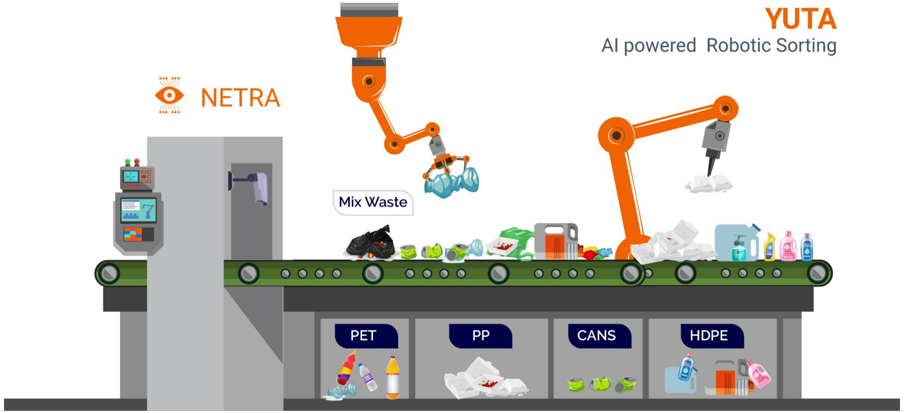
RL Model and Approach

- **Algorithms Used:**
  - PPO (Proximal Policy Optimization)
  - DQN (Deep Q-Network)
  - A2C (Advantage Actor-Critic)
- **Baseline for Comparison:**
  A Rule-Based Agent (RBA) with no learning or adaptability
- **Training Setup:**
  - 100k timesteps
  - Evaluated across 4 scenarios (random/seasonal input, noise, action penalties)

RL Model and Approach

Novel Techniques and Modifications

- **Custom Environment Built in Gymnasium**
- **Action Space Modifications:**
  - Basic: 10 belt speed levels
  - Advanced: 30 total actions (10 speeds × 3 sorting modes)
- **Reward Function:**
  - Balances speed + accuracy
  - Penalizes low accuracy and frequent speed changes
- **Observation Space Expanded in Advanced Mode:**
  - Material ratio included
  - Sorting accuracy affected by selected mode (correct mode = higher accuracy)



Experimental methodology

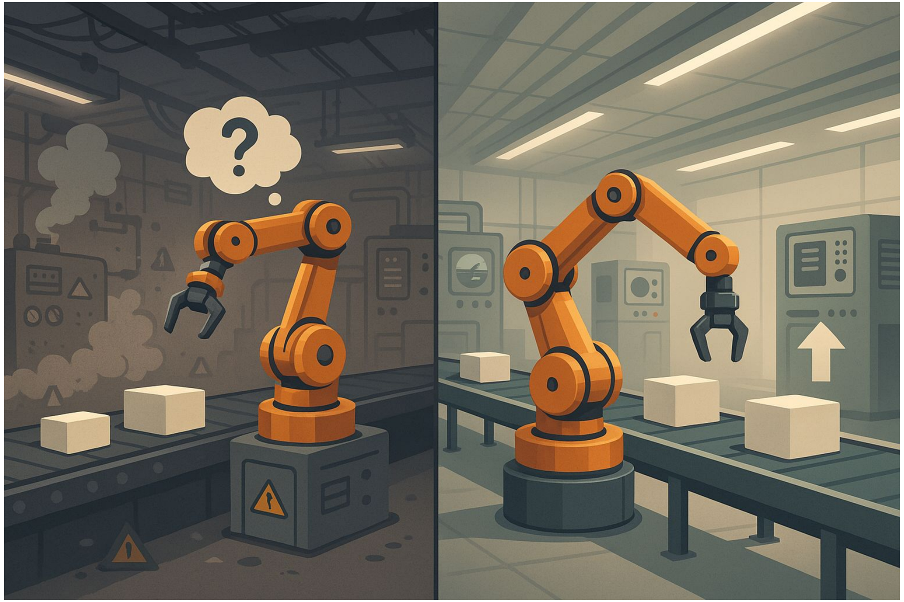
Experimental Methodology:

Baseline Environment:

The experiments were conducted in two custom-built environments:

- **Basic Environment:** Focused on discrete belt speed adjustments to maintain optimal sorting accuracy.
- **Advanced Environment:** Introduced sorting modes (basic, positive, negative) and additional sensor input (material composition), increasing complexity.

Both environments were implemented using Gymnasium (v0.29.1) to ensure compatibility with RL agents and reproducibility.



Experimental Methodology

Effectiveness of RL approach:

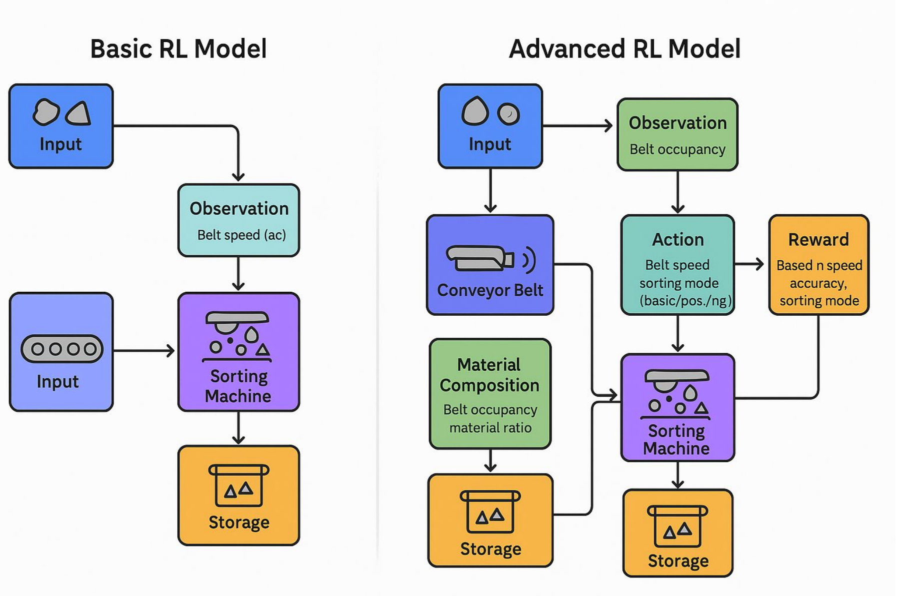
The authors tested three popular RL algorithms using the **Stable-Baselines3 (v2.2.1)** library:

- **Proximal Policy Optimization (PPO)**
- **Deep Q-Networks (DQN)**
- **Advantage Actor-Critic (A2C)**

Each agent was trained for **100,000 timesteps**, with 250 steps per episode during training, and 50 steps per episode during evaluation.

A **Rule-Based Agent (RBA)** was implemented as the baseline. It:

- Generated a lookup table mapping observations to the best immediate reward
- Did not consider long-term reward accumulation or patterns
- Was effective only in simple scenarios but lacked adaptability



Experimental Methodology

To simulate realistic and varying conditions, four environment setups were tested:

- **Random Input:** Unpredictable material ratios.
- **Seasonal Input:** Periodic patterns in material flow, mimicking real-world production cycles.
- **Noise:** Added to simulate sensor variability and real-world uncertainty.
- **Action Penalty:** Applied to discourage frequent speed changes (to reflect mechanical wear in real life).

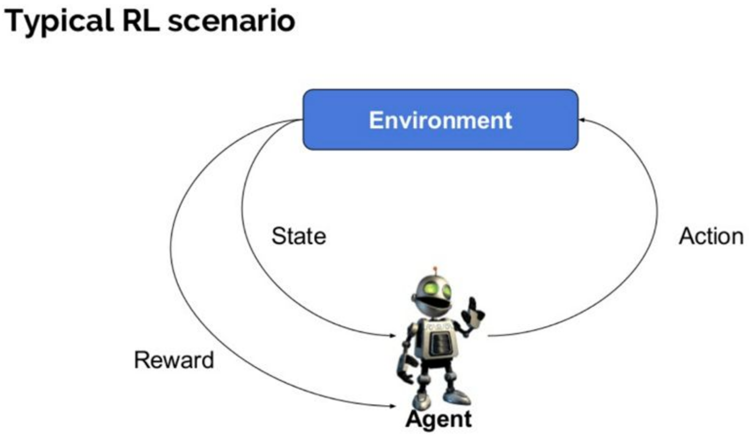
Index	Env	Algorithm	Input	Noise	Action Penalty	Speed (Mean)	Purity (Mean)	Reward	Notes
A1	Basic	RBA	R	0.0	0.0	55	85	**26.56**	
A2	Basic	DQN	R	0.0	0.0	44	85	23.9	Fig. 2
A3	Basic	PPO	R	0.0	0.0	53	85	26.32	
A4	Basic	A2C	R	0.0	0.0	45	85	24.39	
A5	Adv	RBA	R	0.0	0.0	59	94	**36.48**	
A6	Adv	DQN	R	0.0	0.0	40	93.5	31.01	
A7	Adv	PPO	R	0.0	0.0	55	94	35.29	
A8	Adv	A2C	R	0.0	0.0	50	94	34.12	
B1	Basic	RBA	S	0.0	0.5	72	83.5	11.95	
B2	Basic	DQN	S	0.0	0.5	53	80.5	23.46	
B3	Basic	PPO	S	0.0	0.5	66	83.5	**28.06**	
B4	Basic	A2C	S	0.0	0.5	50	53	18.33	static
B5	Adv	RBA	S	0.0	0.5	77	91.5	27.33	Fig. 6
B6	Adv	DQN	S	0.0	0.5	48	91.5	30.23	
B7	Adv	PPO	S	0.0	0.5	68	91.5	**36.85**	Fig. 5
B8	Adv	A2C	S	0.0	0.5	44	91.5	30.61	

Experimental Methodology

Main performance metrics:

- **Mean Reward**: Combination of speed and sorting accuracy.
- **Sorting Purity**: Proportion of correctly sorted material (precision).
- **Belt Speed**: Mean speed selected by the agent.

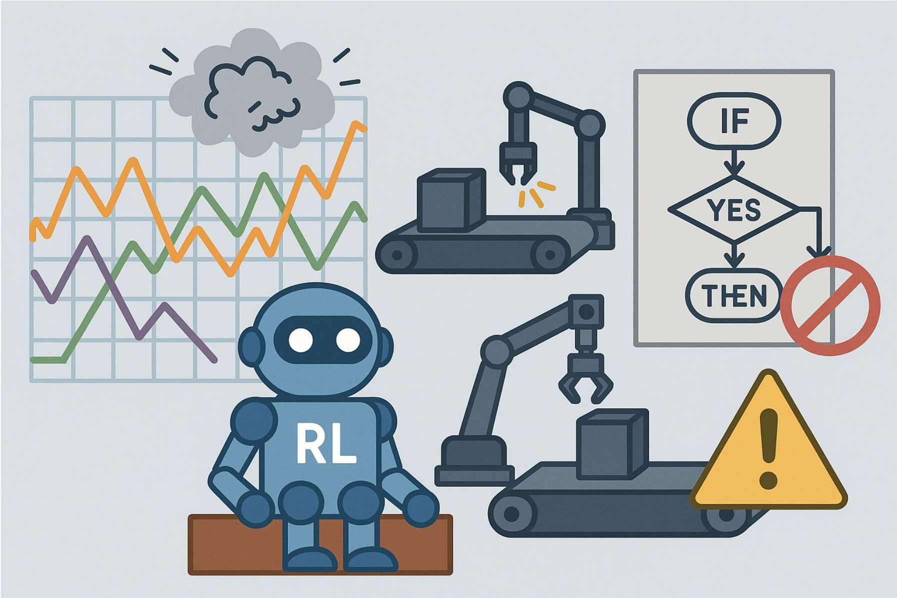
Each agent's behavior and performance were measured across **10 deterministic test environments**, and their results were averaged for comparison.



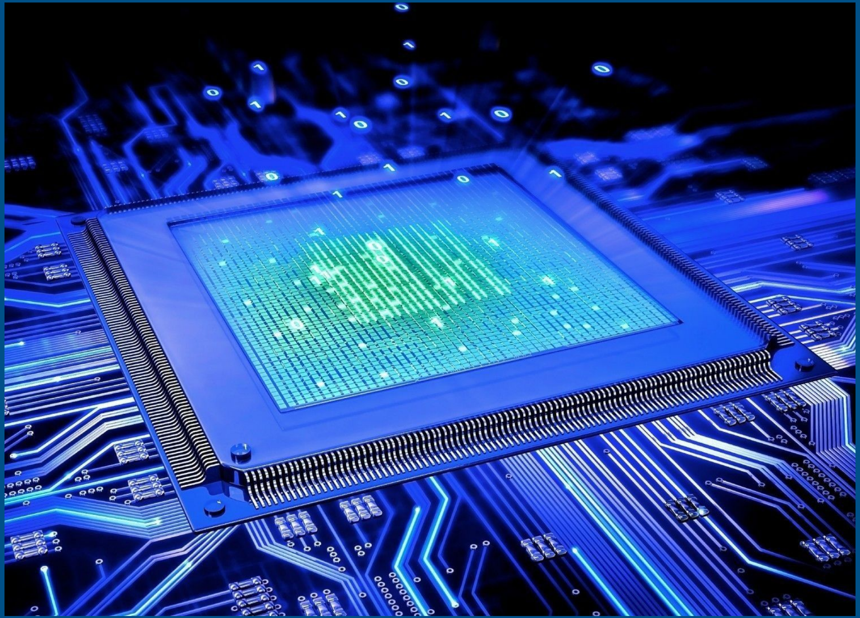
Experimental Methodology

Why Experiments Are Meaningful:

- The test conditions simulate **dynamic, noisy, and evolving environments**, making the challenge more realistic.
- By using **both simple and advanced setups**, the experiments demonstrate how well RL agents can adapt to **increasing complexity**.
- The **comparison against a traditional rule-based system** highlights the advantages of RL in learning patterns and optimizing long-term performance.
- Including noise and penalties simulates **real industrial constraints**, such as sensor errors and mechanical limitations.



Algorithms Overview



Rule Based Method: the baseline

Rule-Based Agent #1 (Simple Environment):

1. Initialize the environment: occupancy, speed
2. Create_reward_table: Iterates over all possible combinations of: **occupancy levels**, **belt speeds**. For each combination, calls calculate_reward and stores the result as *(occupancy, speed, reward)*
3. Get_reward_from_table: Looks up the precomputed reward from the reward table for the given occupancy and speed.
4. Get_best_action: returns the belt speed that gives the highest reward from the reward table.
5. Predict: returns the best action based on the reward table
6. Save reward matrix to CSV file

Rule-Based Agent #2 (Advanced Environment):

1. **Initialize the Environment:** occupancy, speed, **mode**, **ratio**
2. Create_reward_table: Iterates over all possible combinations of: occupancy levels , belt speeds, **modes, ratio**, For each combination, calls Calculate_reward and stores the result as *(occupancy, speed, **mode, ratio_category**, reward)*
3. Get_reward_from_table: Retrieves the reward from the precomputed reward table for the given inputs.
4. Get_best_action:
    a. Compares rewards across all possible belt speeds
    b. Returns the belt **speed** that yields the maximum **reward**.
5. Predict: returns the best action based on the reward table.
6. Save reward matrix to CSV file

Proximal Policy Optimization PPO model

1. `_setup_model` method initializes the model, then converts `clip_range` and `clip_range_vf` into schedules
2. **Training loop:**
    a. Compute **clipping ranges**
    b. For each epoch:
        i. shuffles the **rollout buffer**
        ii. divides the **buffer into mini-batches**.
        iii. updates the **policy on each minibatch**.
    c. For each mini-batch:
        i. get **action probabilities/logits** and **value function predictions**.
        ii. Calculate the losses: 1. **policy loss** 2. **value loss** 3. **entropy loss**
        iii. Combine all into the **total loss**:

```
loss = policy_loss + self.ent_coef * entropy_loss + self.vf_coef * value_loss
```

        iv. **Gradient** update: The gradients are computed and clipped (if necessary)
3. **KL Divergence**: PPO also includes early stopping based on the KL divergence between the new and old policies. If the divergence exceeds a threshold, training stops early.

Policy loss: learn better decisions

Value loss: improve future predictions

Entropy loss: encourage exploration

Advantage Actor Critic (A2C) model

1. Constructor `__init__` initializes the algorithm with several parameters: **policy**: (MLP, CNN, etc.). **env**: Gym environment, **learning_rate**, **n_steps**, **gamma**, **etc.**
2. **Train method**:
   a. **Data processing**: The agent collects data from the **rollout buffer**, which stores the experiences collected during episodes (observations, actions, rewards, etc.)
   b. **Policy evaluation**:
      i. **values:** expected returns from the current state
      ii. **log_prob:** log probability of the taken actions under the current policy
      iii. **entropy:** a measure of randomness in the policy, used for encouraging exploration
   c. **Policy gradient loss calculation:** calculated as the negative of the product of the advantage and the log probability of the taken actions
   d. **Value loss calculation:** calculated using the MSE between the predicted values and the actual returns
   e. **Entropy loss calculation:** entropy encourages exploration by penalizing deterministic policies
   f. **Total loss calculation**

```
loss = policy_loss + self.ent_coef * entropy_loss + self.vf_coef * value_loss
```

   g. **Backpropagation and gradient update:** performs **gradient descent** on the total loss

Policy loss: learn better decisions

Value loss: improve future predictions

Entropy loss: encourage exploration

```
# Policy gradient loss
policy_loss = -(advantages * log_prob).mean()
```

```
# Value loss using the TD(gae_lambda) target
value_loss = F.mse_loss(rollout_data.returns, values)
```

```
# Entropy loss favor exploration
if entropy is None:
    # Approximate entropy when no analytical form
    entropy_loss = -th.mean(-log_prob)
else:
    entropy_loss = -th.mean(entropy)
```

Deep Q-Network DQN model

1. Constructor `__init__` initializes the algorithm with several parameters: **policy**, **env**: Gym environment, **learning_rate**, **buffer_size**, **n_steps**, **gamma**, **etc.**
2. **Train method:**

   For each gradient step:

   a. **Batch sampling:** randomly samples transitions *(s,a,r,s',done)* from the replay buffer
   b. Compute **target Q values**:
      i. **predict next Q-values** using the target network for the next states
      ii. **select the max Q-value** across actions (greedy action)
      iii. **Zero out** future rewards if the episode is done
      iv. Calculate **target values** using the Bellman equation
   c. **Get current Q value:**
      i. Predicts Q(s,a) using the **main Q-network**.
      ii. Extracts only the Q-values for the **actions actually taken**
   d. Compute the **loss** (Huber loss)
   e. **Backpropagation and gradient update**

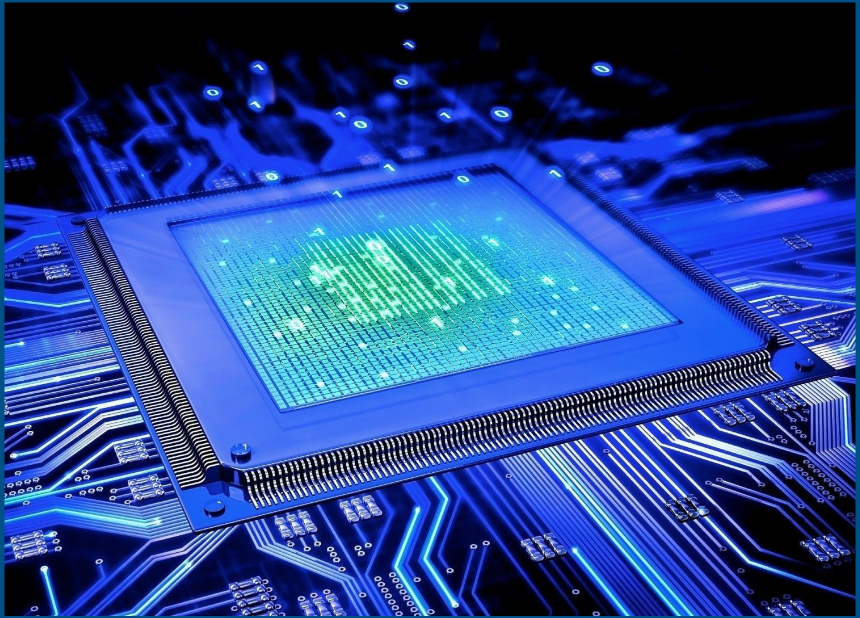
```
with th.no_grad():
    # Compute the next Q-values using the target network
    next_q_values = self.q_net_target(replay_data.next_observations)
    # Follow greedy policy: use the one with the highest value
    next_q_values, _ = next_q_values.max(dim=1)
    # Avoid potential broadcast issue
    next_q_values = next_q_values.reshape(-1, 1)
    # 1-step TD target
    target_q_values = replay_data.rewards + (1 - replay_data.dones) * self.gamma * next_q_values
```

```
# Get current Q-values estimates
current_q_values = self.q_net(replay_data.observations)

# Retrieve the q-values for the actions from the replay buffer
current_q_values = th.gather(current_q_values, dim=1, index=replay_data.actions.long())
```

```
# Compute Huber loss (less sensitive to outliers)
loss = F.smooth_l1_loss(current_q_values, target_q_values)
losses.append(loss.item())
```

Sorting_Env Model



Main idea of the model

Environments: Depending on COMPLEX:

1. COMPLEX =0: SortingEnvironment: basic sorting logic
2. COMPLEX =1: SortingEnvironmentAdv: advanced sorting logic

Input Options:

1. INPUT = r: random inputs, s3 or s9: seasonal input (simple or complex).
2. THRESHOLD = 0.7 : Threshold for accuracy
3. NOISE: noise level in observations (range 0 - 1).
4. ACTION_PENALTY: adds cost for extra actions to encourage efficiency.
5. TIMESTEPS = 100_000 :Total Training Steps (Budget)
6. STEPS_TRAIN = 250 : Steps per Episode (Training)
7. STEPS_TEST = 50 : Steps per Episode (Testing)
8. SEED = 42 : Random Seed for Reproducibility

```
# -----------------------------------------------------------*/
if TRAIN:
    MODELS = ["A2C", "PPO", "DQN"]
else:
    MODELS = ["RBA", "A2C", "PPO", "DQN"]

TAG_ADD = "B_Base"        # Additional Tag for spedific runs

COMPLEX = 0             # Complex Environment (1) or Simple Environment (0)
INPUT = "r"             # Input Type r=random, s3=simple_saisonal, s9=complex_saisonal
THRESHOLD = 0.7         # Threshold for Accuracy
NOISE = 0               # Noise Range (0.0 - 1.0)

if INPUT == "r":
    ACTION_PENALTY = 0      # Action Penalty for Taking Too Many Actions
else:
    ACTION_PENALTY = 0.5    # Action Penalty for Taking Too Many Actions

TIMESTEPS = 100_000     # Total Training Steps (Budget)
STEPS_TRAIN = 250       # Steps per Episode (Training)
STEPS_TEST = 50         # Steps per Episode (Testing)
SEED = 42               # Random Seed for Reproducibility

SAVE = 1                 # Save Images
DIR = "./img/figures/"   # Directory for Image-Logging
```

Main idea of the model

Environment creation:

1. Create_environment creates either a simple (SortingEnvironment) or complex environment (SortingEnvironmentAdv).
2. Configured with a number of steps per episode

```
def create_environment(max_steps=STEPS_TEST, seed=None):
    """Function to create a new environment based on parameters"""
    print(f"Creating environment with: complexity={COMPLEX}, steps={TIMESTEPS}, input_type={INPUT},\
          noise_level={NOISE}, seed={seed}")

    if COMPLEX:
        return SortingEnvironmentAdv(max_steps=max_steps, input=INPUT, action_penalty=ACTION_PENALTY,
                                     noise_lv=NOISE, seed=seed, threshold=THRESHOLD)
    else:
        return SortingEnvironment(max_steps=max_steps, input=INPUT, action_penalty=ACTION_PENALTY,
                                  noise_lv=NOISE, seed=seed, threshold=THRESHOLD)
```

Main idea of the model

Model execution MODES:

1. TRAIN: trains new selected RL model(s)
2. TEST: run a random or trained agent
3. BENCHMARK: Compare performance of multiple RL algorithms
4. RULE_BASED: uses a pre-defined RBA model
5. LOAD: loads pre-trained RL models

```
# 1. Select Mode
# ------------------------------------------------------------*/
TEST = 0                    # Test a random run
TRAIN = 1                   # Train a new model
BENCHMARK = 0               # Benchmarking multiple models
SMALL_CHECK = 0             # Small Check for Testing
LOAD = 0                    # Load a pre-trained model

RULE_BASED = 0              # Rule Based Agent
INTERACTIVE = 0             # For Interactive Mode (Manual Control)
VIDEO = 0                   # Record Video, for Test and Load Mode
ENV_ANALYSIS = 0            # Analyse Environment
TUNING = 0                  # Tuning
```

Main idea of the model

If MODE = RULE_BASED:

- No training cycle
- **Environment** created
- Rule-based **agent initialized**
- Run **diagnostics** and **plotting**
- Reinitialize the environment with a fixed seed for consistent testing. This ensures **test_model()** runs on a known and repeatable setup.
- **Test** the rule-based agent and tracks **performance**

```
if RULE_BASED:
    env = create_environment()
    agent = agent_model(env)
    agent.run_analysis()
    env = create_environment(seed=SEED)
    test_model(agent, env=env, tag=TAG, save=SAVE, title=f"(Rule-Based Agent {TAG})", steps=STEPS_TEST, dir=DIR)
```

Main idea of the model

If MODE = LOAD:

- Create new **environment**
- Load **pre-trained** model
- **Test** model

```python
if LOAD:

    for modelname in MODELS:

        if "DQN" in modelname:
            env = create_environment(seed=SEED)
            model = DQN.load(f"models/{modelname.lower()}_sorting_env_{TAG}")
            test_model(model, env=env, tag=TAG, save=SAVE, title=f"({modelname} {TAG})", steps=STEPS_TEST, dir=DIR)
        elif "PPO" in modelname:
            env = create_environment(seed=SEED)
            model = PPO.load(f"models/{modelname.lower()}_sorting_env_{TAG}")
            test_model(model, env=env, tag=TAG, save=SAVE, title=f"({modelname} {TAG})", steps=STEPS_TEST, dir=DIR)
        elif "A2C" in modelname:
            env = create_environment(seed=SEED)
            model = A2C.load(f"models/{modelname.lower()}_sorting_env_{TAG}")
            test_model(model, env=env, tag=TAG, save=SAVE, title=f"({modelname} {TAG})", steps=STEPS_TEST, dir=DIR)
        elif "RBA" in modelname:
            train_env = create_environment(max_steps=STEPS_TRAIN, seed=100)
            agent = agent_model(train_env)
            env = create_environment(seed=SEED)
            test_model(agent, env=env, tag=TAG, save=SAVE,
                       title=f"(Rule-Based Agent {TAG})", steps=STEPS_TEST, dir=DIR)

        else:
            raise ValueError(f"Unsupported model type: {modelname}")
```

Main idea of the model

Model execution modes:

If TRAIN mode enabled:

1. RL_Trainer() executes the selected reinforcement learning algorithm
2. Test_model() evaluates the trained model on this test environment

```
if TRAIN:
    train_env = create_environment(max_steps=STEPS_TRAIN)
    model = RL_Trainer(model_type=MODELS[0], env=train_env, total_timesteps=TIMESTEPS, tag=TAG)
    env = create_environment(seed=SEED)
    test_model(model, env=env, tag=TAG, save=SAVE,
                title=f"(Trained Run, {MODELS[0]} {TAG})", steps=STEPS_TEST, dir=DIR)
```

Main idea of the model

If TEST or BENCHMARK modes selected:

1. A new **environment** is instantiated with fixed seed.
2. The trained model is **evaluated** over a specified number of steps

```
if TEST or BENCHMARK: #enviroment testing with and without visualization
    env = create_environment(seed=SEED)
    if VIDEO:
        env_simulation_video(env=env, tag=TAG, steps=STEPS_TEST)
    else:
        test_env(env=env, tag=TAG, save=SAVE, title=f"(Random Run, {TAG})", steps=STEPS_TEST, dir=DIR, seed=42)
```

For BENCHMARK mode only:

1. Create train and evaluation **environments**
2. Initialize and **run** the benchmark **model**
3. benchmark.run_benchmark(dir=DIR) **trains** and **evaluates** all listed models

```
if BENCHMARK:
    train_env = create_environment(max_steps=STEPS_TRAIN)
    eval_env = create_environment(max_steps=STEPS_TEST, seed=SEED)
    benchmark = RLBenchmark(models=MODELS, total_timesteps=TIMESTEPS, n_eval_episodes=10, train_env=train_env,
                            eval_env=eval_env, tag=TAG, agent_model=agent_model)
    benchmark.run_benchmark(dir=DIR)
    LOAD = 1
```

Model tuning

If the variable TUNING is set to **True** then the tuning process of model hyperparameters is executed

Hyperparameters:

- Learning rate
- Entropy coefficient
- Gamma (discount factor)
- Noise
- Action penalties

```python
if TUNING:
    models = ["RBA", "A2C", "PPO", "DQN"]  # List of models to tune
    tag = "experiment_1"  # Tag for this run

    tuner = Tuning(models=models, tag=tag)
    tuner.run_tuning()

    print("Tuning completed. Results saved. 🎧")
```

Rule-Based Agent (RBA) Tuning:

- The RBA is treated separately from the RL models.
- For each combination of **input_type**, **noise**, and **action_penalty**, an environment is created and the agent is trained and evaluated.
- The results are stored in the **results** dictionary and appended to the CSV file.

Optuna Integration:

- **Tuning_Optuna** a hyperparameter optimization library.

__main__ function

- **Comparative analysis:** the repeated blocks allow the testing and comparison of multiple scenarios, configurations, or conditions within the same environment.

```
# Main Function
# ---------------------------------------------------------------*/
if __name__ == "__main__":

    # Environment Analysis
    run_env(ENV_ANALYSIS=1)

    # # A: Random Input
    run_env(BENCHMARK=0, TAG_ADD="A", COMPLEX=0, INPUT="r", NOISE=0, ACTION_PENALTY=0)
    run_env(BENCHMARK=0, TAG_ADD="A", COMPLEX=1, INPUT="r", NOISE=0, ACTION_PENALTY=0)

    # # B: Seasonal Input
    # run_env(BENCHMARK=1, TAG_ADD="B", COMPLEX=0, INPUT="s9", NOISE=0, ACTION_PENALTY=0.5)
    # run_env(BENCHMARK=1, TAG_ADD="B", COMPLEX=1, INPUT="s9", NOISE=0, ACTION_PENALTY=0.5)

    # # C: Random Input with Noise
    # run_env(BENCHMARK=1, TAG_ADD="C", COMPLEX=0, INPUT="r", NOISE=0.3, ACTION_PENALTY=0)
    # run_env(BENCHMARK=1, TAG_ADD="C", COMPLEX=1, INPUT="r", NOISE=0.3, ACTION_PENALTY=0)

    # # D: Seasonal Input with Noise
    # run_env(BENCHMARK=1, TAG_ADD="D", COMPLEX=0, INPUT="s9", NOISE=0.3, ACTION_PENALTY=0.5)
    # run_env(BENCHMARK=1, TAG_ADD="D", COMPLEX=1, INPUT="s9", NOISE=0.3, ACTION_PENALTY=0.5)
```

EXAMPLE:

```
# # A: Random Input
run_env(BENCHMARK=0, TAG_ADD="A", COMPLEX=0, INPUT="r", NOISE=0, ACTION_PENALTY=0)
run_env(BENCHMARK=0, TAG_ADD="A", COMPLEX=1, INPUT="r", NOISE=0, ACTION_PENALTY=0)
```

- No benchmark
- Model 1 = base, model 2 = advanced
- Input = random in both models (r=random, s3=simple_saisonal, s9=complex_seasonal)
- Noise = 0 (range 0 to 1)
- ACTION_PENALTY (for taking too many actions) = 0 (options: 0 or 0.5)

Paper Results

Results: premise

Index	Env	Algorithm	Input	Noise	Action Penalty	Speed (Mean)	Purity (Mean)	Reward	Notes
A1	Basic	RBA	R	0.0	0.0	55	85	**26.56**	
A2	Basic	DQN	R	0.0	0.0	44	85	23.9	Fig. 2
A3	Basic	PPO	R	0.0	0.0	53	85	26.32	
A4	Basic	A2C	R	0.0	0.0	45	85	24.39	
A5	Adv	RBA	R	0.0	0.0	59	94	**36.48**	
A6	Adv	DQN	R	0.0	0.0	40	93.5	31.01	
A7	Adv	PPO	R	0.0	0.0	55	94	35.29	
A8	Adv	A2C	R	0.0	0.0	50	94	34.12	
B1	Basic	RBA	S	0.0	0.5	72	83.5	11.95	
B2	Basic	DQN	S	0.0	0.5	53	80.5	23.46	
B3	Basic	PPO	S	0.0	0.5	66	83.5	**28.06**	
B4	Basic	A2C	S	0.0	0.5	50	53	18.33	static
B5	Adv	RBA	S	0.0	0.5	77	91.5	27.33	Fig. 6
B6	Adv	DQN	S	0.0	0.5	48	91.5	30.23	
B7	Adv	PPO	S	0.0	0.5	68	91.5	**36.85**	Fig. 5
B8	Adv	A2C	S	0.0	0.5	44	91.5	30.61	
C1	Basic	RBA	R	0.3	0.0	55	73.5	19.77	
C2	Basic	DQN	R	0.3	0.0	42	85	23.34	
C3	Basic	PPO	R	0.3	0.0	47	84	**23.79**	
C4	Basic	A2C	R	0.3	0.0	48	83	23.1	
C5	Adv	RBA	R	0.3	0.0	61	85	29.5	
C6	Adv	DQN	R	0.3	0.0	41	93	31	
C7	Adv	PPO	R	0.3	0.0	49	93.5	**33.62**	
C8	Adv	A2C	R	0.3	0.0	50	92	32.74	
D1	Basic	RBA	S	0.3	0.5	73	76	10.21	
D2	Basic	DQN	S	0.3	0.5	53	77	**22.98**	
D3	Basic	PPO	S	0.3	0.5	64	64	21.62	
D4	Basic	A2C	S	0.3	0.5	50	53	18.33	Static
D5	Adv	RBA	S	0.3	0.5	78	83	22.83	
D6	Adv	DQN	S	0.3	0.5	49	89.5	27.96	
D7	Adv	PPO	S	0.3	0.5	69	91.5	**35.53**	
D8	Adv	A2C	S	0.3	0.5	50	69.5	25.67	

- Comparison of different RL algorithms across **four different set ups**: A, B, C, D
- A, C: random input
- B, C: seasonal input + action penalty applied
- All models trained until performance plateau reached (~100k steps)

Results: overview

PPO

RBA

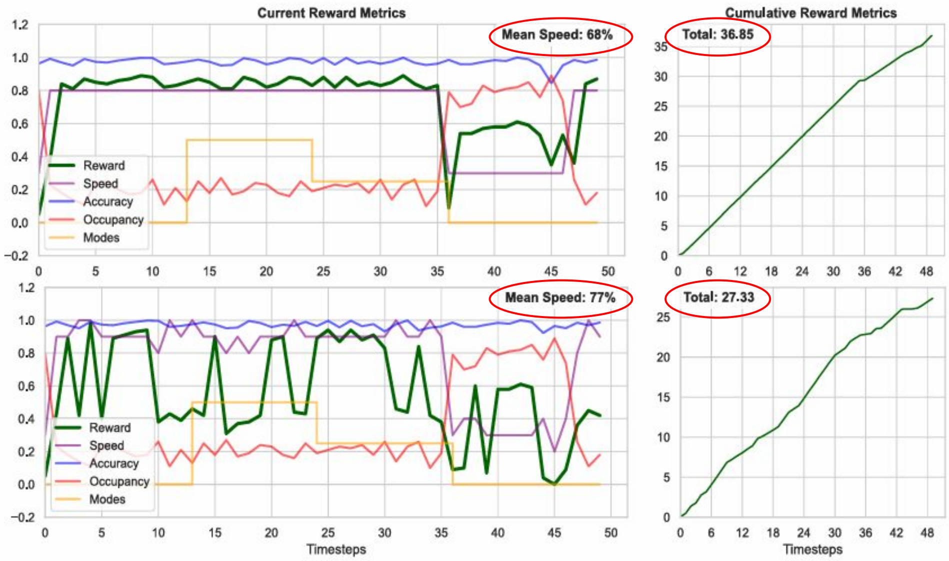


FIGURE 5-6. Immediate and cumulative reward metrics of the advanced sorting environment with seasonal input (for pattern, see Fig. 4, right). The top panel (Fig. 5) shows PPO Agent actions, and the bottom panel (Fig. 6) shows Rule-Based Agent actions. The left panels show the current reward metrics over 50 timesteps, including reward (green), speed (blue), accuracy (purple), occupancy (red), and sorting mode (yellow), coded as basic (0), positive sorting (0.5), and negative sorting (1.0). The right panel illustrates the cumulative reward metrics over the same timesteps.

- In setups with **seasonal input** (B, D), the RL agents maintained **higher purity** and rewards, especially under noisy conditions
- The **RBA** did not learn patterns, treating each input individually, resulting in **lower total rewards** due to frequent action penalties

Results: overview

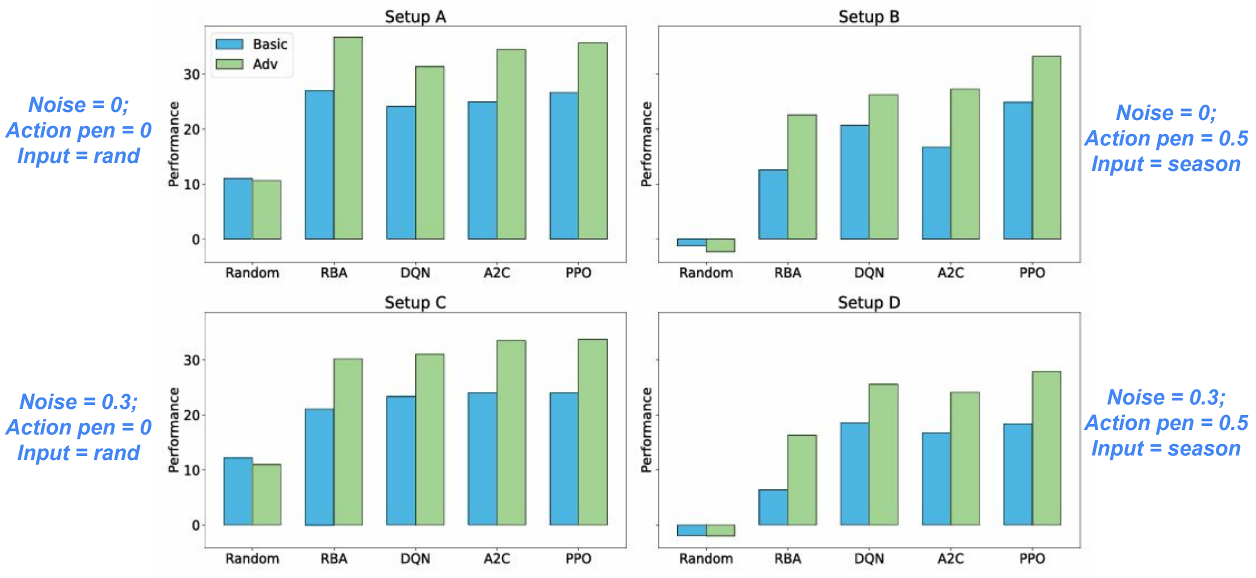


FIGURE 7. Comparison of benchmarking performance ("reward") of multiple RL algorithms in different setups (A, B, C, D) of the sorting environment (see Table 1). Each value depicts the mean of evaluations in ten distinct environments.

- The agents in **advanced environments** consistently outperformed the agents in basic environments
- RL agents **outperformed** the RBA baseline
- Introducing **noise** (C, D) generally led to a **decrease in performance** metrics across all models.
- **DQN** and **PPO** displayed better **robustness** under **noisy** conditions
- The **learning** behavior of RL agents was significantly influenced by **hyperparameters**
- In B & D set-ups (seasonal input) RL agents showed superior adaptability than RBA

Results: overview

Index	Env	Algorithm	Input	Noise	Action Penalty	Speed (Mean)	Purity (Mean)	Reward	Notes
A1	Basic	RBA	R	0.0	0.0	55	85	**26.56**	
A2	Basic	DQN	R	0.0	0.0	44	85	23.9	Fig. 2
A3	Basic	PPO	R	0.0	0.0	53	85	26.32	
A4	Basic	A2C	R	0.0	0.0	45	85	24.39	
A5	Adv	RBA	R	0.0	0.0	59	94	**36.48**	
A6	Adv	DQN	R	0.0	0.0	40	93.5	31.01	
A7	Adv	PPO	R	0.0	0.0	55	94	35.29	
A8	Adv	A2C	R	0.0	0.0	50	94	34.12	
B1	Basic	RBA	S	0.0	0.5	72	83.5	11.95	
B2	Basic	DQN	S	0.0	0.5	53	80.5	23.46	
B3	Basic	PPO	S	0.0	0.5	66	83.5	**28.06**	
B4	Basic	A2C	S	0.0	0.5	50	53	18.33	static
B5	Adv	RBA	S	0.0	0.5	77	91.5	27.33	Fig. 6
B6	Adv	DQN	S	0.0	0.5	48	91.5	30.23	
B7	Adv	PPO	S	0.0	0.5	68	91.5	**36.85**	Fig. 5
B8	Adv	A2C	S	0.0	0.5	44	91.5	30.61	
C1	Basic	RBA	R	0.3	0.0	55	73.5	19.77	
C2	Basic	DQN	R	0.3	0.0	42	85	23.34	
C3	Basic	PPO	R	0.3	0.0	47	84	**23.79**	
C4	Basic	A2C	R	0.3	0.0	48	83	23.1	
C5	Adv	RBA	R	0.3	0.0	61	85	29.5	
C6	Adv	DQN	R	0.3	0.0	41	93	31	
C7	Adv	PPO	R	0.3	0.0	49	93.5	**33.62**	
C8	Adv	A2C	R	0.3	0.0	50	92	32.74	
D1	Basic	RBA	S	0.3	0.5	73	76	10.21	
D2	Basic	DQN	S	0.3	0.5	53	77	**22.98**	
D3	Basic	PPO	S	0.3	0.5	64	64	21.62	
D4	Basic	A2C	S	0.3	0.5	50	53	18.33	Static
D5	Adv	RBA	S	0.3	0.5	78	83	22.83	
D6	Adv	DQN	S	0.3	0.5	49	89.5	27.96	
D7	Adv	PPO	S	0.3	0.5	69	91.5	**35.53**	
D8	Adv	A2C	S	0.3	0.5	50	69.5	25.67	

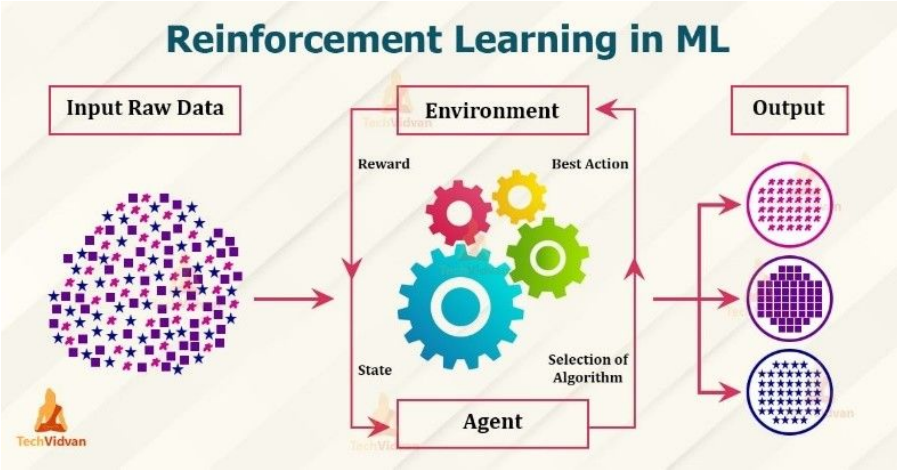
- A2C algorithm tended to select a static belt speed for the entire period, which negatively affected its performance

- Although a static speed might be optimal for specific occupancy levels, it proved to be suboptimal overall, leading to lower rewards compared to more dynamic strategies.

Conclusion & Next Steps

Comments

- The inclusion of **sensor noise, action penalties, and seasonal input patterns** makes the learning problem better for real world problems.
- Basic vs. Advanced makes the environment good to test for both standard benchmarking and **transfer learning**.
- Modeling real-world constraints like **sensor noise and penalties** adds depth.
- Reward function balanced speed and accuracy, but there could be more reward structures such as energy usage, wear-and-tear to reflect industrial trade-offs.



Project progress updates

Completed:

1. Chose our Research Paper: SortingEnv: A RL environment for Industrial Sorting
2. Delivered a detailed Simulation Report
3. Project Paper Presentation

To DO:

1. Final Demo
2. Write and Submit Final Project Report