

Open-World Reinforcement Learning

Reinforcement Learning in offenen Umgebungen

Bachelor thesis in the department of Computer Science by Bora Büyükbas

Date of submission: September 29, 2026

1. Review: M.Sc. Lucas Schulze
2. Review: Dr.-Ing. Oleg Arenz
3. Review: Prof. Dr. Jan Peters
Darmstadt



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Bora Büyükbaz, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Darmstadt, 29. September 2026



Bora Büyükbaz

Abstract

Open-world environments evolve continuously, with shifting dynamics and structures, resulting in changes to the dimensions of the observation and action spaces. For problems formulated in such environments, reinforcement learning enables autonomous control with the training process through trial and error. However, traditional approaches rely on fixed observation and action sizes with the policy networks, making the trained policy obsolete with any structural change. It is not always feasible or desirable to retrain a policy for the agents operating in such dynamic environments with potentially infinitely many possible configurations.

This thesis studies a single policy which can control open-world environments without retraining. As an example of such environments, Extract-Transform-Load pipelines are used with a directed acyclic graph model, having a varying pipeline topology and dynamic number of cores connecting the data pools. We propose a policy architecture based on the Unified Robot Morphology Architecture (URMA). This architecture uses a single core policy for producing general action the pipeline should take, a decoder with topological prior information to calculate action values for each core, and an encoder component for producing a fixed-size observation of pipeline. Together with three baseline approaches, a graph neural network and Transformer-based approaches, the policy networks are trained under the same conditions and compared against each other.

The trained policies are evaluated with both the training and an unseen environment configuration. Experimental results on both configurations show that the usage of a core-wise decoder and topological prior information provides the best results. URMA-based approaches using prior information provide 8% higher mean reward value than the model trained only on the specific dimension sizes, which can only be evaluated in training environment configuration. With unseen action and observation dimensions, URMA-based approaches reach 8% higher mean reward value than the second best approach. While only allocating up to 9% more mean resources, URMA-based approaches change the actions by 50% less compared to the second best approach, resulting in smoother controlling.

Contents

1	Introduction	2
2	Related Work	4
2.1	Policies for Open and Evolving Environments	4
2.2	Variable-Size and Multi-Agent Network Architectures	5
2.3	Resource Allocation Systems with Reinforcement Learning	5
3	Foundations	7
3.1	Extract-Transform-Load Processes	7
3.2	Reinforcement Learning	9
3.3	Neural Architectures	11
3.4	Unified Robot Morphology Architecture	15
3.5	AgentFormer	17
4	Methodology	19
4.1	Problem Formulation	19
4.2	Environment Setup	20
4.3	Reward Formulation	23
4.4	Policy Networks	24
4.5	Training Setup	32
4.6	Evaluation Criteria	33
5	Experiments & Results	36
5.1	Experimental Setup	36
5.2	Baselines	37
5.3	Policy Evaluations	38
6	Discussion	53
6.1	Task Performance	53

6.2	Generalization	55
6.3	Stability	57
7	Outlook	59
7.1	Conclusion	59
7.2	Limitations	60
7.3	Future Work	61

1 Introduction

Over time, the system characteristics progress continuously, with the environmental boundaries being neither fixed nor precisely known in advance. Not only the random fluctuations, but also the structural changes introduce fundamental dynamism for the systems operating in an *open-world* environment. For instance, robots need to make sure that they can perform similarly across changing environmental conditions and states, autonomous vehicles must consider traffic congestion and varying driving styles, drones must take fluctuating winds and weather into account. Therefore, agents in such settings cannot rely on predefined representations, and must also adapt to this evolution.

Reinforcement learning (RL) is a key approach for autonomously addressing the problems inside such environments. Core idea of RL is to explore the environment with interactions for collecting the highest reward. Instead of expert demonstrations or an existing policy, it explores the environment and learns through trial and error to maximize a given reward [1]. In traditional RL methods, it is assumed that the observation and action sizes are usually fixed during the training and inference. Changing their sizes would eventually require learning a new policy, and therefore new weights for an updated network. As components of the real-world systems could evolve over time, it is not always feasible to train different policies that could work on wide-range situations due to the computational cost and varying number of possibilities. While these challenges are apparent in the robotics, similar issues arise in data engineering pipelines.

As an example for such open-world environment, Extract-Transform-Load (ETL) processes are used in today's data processing solutions. Between the sources and the targets, ETL processes fetch new documents, apply transformations and lastly output the processed data into target. In addition to the dynamic resource allocation problem that is affected by the data flow, systems could contain multiple ETL cores as a pipeline for more complicated processing, possibly also affecting each other [2]. Furthermore, with removal or addition of components, the pipeline topology, observations, and action space sizes change as well.

Thus, the learned policy must adapt to such conditions, which is not easy to do with traditional RL methods.

In this environment setup, although every core acts separately, it is important to both capture core-specific information, and the overall pipeline information during the decision-making process. Because every core contributes to the pipeline state, varying number of cores introduces additional complexity for the policy. Also, not all cores contribute similarly, as some of them can be located in between, while others could have more responsibility to fetch data from the source or commit data to the target. To ensure the result is robust against potential changes of the environment dynamics, it is also important that the changing number of cores, therefore changing amount of observation and action sizes does not break the policy.

This thesis explores various possibilities to handle the dynamic nature of ETL processes, while paying attention to the concerns described in the previous paragraph. The environment is constructed with a graph structure, where every core connects the data pools, including the source and target pools of pipeline with the internal pools between the cores. In addition to using only core-specific information, various modifications are presented to exploit this graph structure inside the policy. Keeping the RL training process and simulator the same, different policy architectures will be trained to obtain a policy, which could be applicable on open-world and dynamic example of ETL processes. For evaluation, the results of the presented architectures are compared to various baselines, and the approaches are assessed with their strengths and weaknesses.

The remaining parts of the thesis are organized into six chapters. Chapter 2 reviews relevant literature to form a foundation for the proposed methodology. Then, Chapter 3 introduces background information and the technical details for ETL processes, RL and neural networks. After that, Chapter 4 provides the experimental methodology for testing the learning capabilities of each configuration. Chapter 5 presents and evaluates the performance of each architecture. These results are interpreted in Chapter 6 to understand the characteristics of the approaches, evaluating their suitability for this task. Finally, Chapter 7 concludes this thesis with a summary and discusses future research directions.

2 Related Work

Learning a single reusable RL policy introduces several challenges, especially in open-world environments like the ETL systems with highly stochastic characteristics and multi-agent setup. This chapter reviews three research topics that represent the major components of the studied problem: (i) policies for open and evolving environments, (ii) variable-size and multi-agent network architectures, and (iii) control of resource allocation systems with RL.

2.1 Policies for Open and Evolving Environments

For learning a policy that stays effective when the environment's parameters are changing during the deployment, [3] introduces the Universal Policy Network (UPN). This network consists of a single policy that is trained on large amounts of dynamics simulations with randomized physical parameters, and an online system identification module for estimation of the actual parameters, which is supplied into the policy network. With the single policy learning the relationship of each physical parameter with the behavior and online system identification providing information about the current environment, UPN can still work well with unknown dynamics without retraining.

A practical implementation of RL agent in real-world situations is demonstrated by [4]. In this work, a quadrupedal locomotion controller is trained in simulation with extremely randomized environmental settings, such as the terrain, friction, delayed control, while utilizing an encoder that combines the sensory information. The result in the real-world conditions verifies that the sufficiently randomized domain during the training process could help policy to generalize to unseen settings, which is also discussed more generally for deep neural networks in [5].

2.2 Variable-Size and Multi-Agent Network Architectures

Approaching the cooperative multi-agent RL as a sequence modeling problem, [6] presents an encoder-decoder Transformer architecture called Multi-Agent Transformer. Initially, encoder processes the agent observation histories with the attention mechanism to produce a general latent representation of the team state, which is used by the decoder to generate each agent's action. Because the number of agents is interpreted as the sequence length, the architecture allows usage of variable number of agents.

Transforming the general knowledge into specialized knowledge, [7] introduces the policy distillation for multiple agents. Initially, a central policy with full global state is trained, which is used to compress this knowledge into small standalone policies for every agent operating on local observations. Forming a teacher-student relationship, the outputs of distilled student policies match the teacher's behavior.

For problems consisting of multiple tasks, [8] shows that sharing the representations across the tasks enables more effective use of multi-task deep RL approaches. By learning the common properties of tasks, the networks could generalize more effectively by using the knowledge of other tasks, providing improved performance. Using a core policy with the task-specific output heads could perform better than both the shared and completely separate architectures, especially with the similar tasks that have various dynamics.

Operating in an open environment where both the environment dynamics and the number of agents change, [9] presents a new approach to control one agent among distinct dynamic agents with separate individual tasks. They introduce a new problem formulation called Double Open Stochastic Bayesian Games, enabling the agents to join or leave episodes. Using this extensible model, modular interaction policies are collected for each task type, which get combined with a blending mechanism to stabilize and maintain high performance. As a result, the agent can coordinate with dynamic number of agents without need of retraining.

2.3 Resource Allocation Systems with Reinforcement Learning

In the scope of increasing network traffic problem, [10] studies an RL agent for controlling the job scheduling weights. Using the class-specific metrics, the RL agent adjusts the system configuration to meet the demands of throughput and delay.



Handling the dynamic resource allocation problems with RL, [11] proposes an incremental RL approach to control the baggage claim systems at the airport. As state and action spaces grow with the new tasks, it extends the learned policy incrementally to explore the new possibilities without full retraining.

3 Foundations

This chapter provides the technical framework used throughout the methodology and in the subsequent chapters. ETL processes are defined as a fundamental data processing pipeline with a mathematical modeling of ETL dynamics, followed by a brief introduction of RL from formalizing the environment interactions to algorithms used for the training in this thesis. Later section introduces the foundational neural network architectures, which act as building blocks for the proposed policy networks. Finally, Section 3.4 and Section 3.5 examine the inspiring approaches on which the architectures proposed in the methodology are based.

3.1 Extract-Transform-Load Processes

As a way to model the data processing pipeline, ETL provides three fundamental phases and a way to utilize available resources on each of them. Extract describes that the data is extracted from a source. Any change on data is handled in Transform. In the end, Load manages the transfer of data into the target [12]. While the naming of phases also defines the abbreviation *ETL*, they can also be called as fetch, enrich, and commit phases respectively. These phases are also visualized in Figure 3.1. Although this visualization considers only one ETL process, multiple processes can be combined for more complex processing in the pipeline as visualized in Figure 3.2.

In the lifecycle of a data processing setup, the volume of processed data increases. To deal with such large amount of data, throughput of the ETL pipelines can be increased by configuring every phase to process multiple records of data in parallel. Nonetheless, with more parallelization, more computing resources must be allocated for each phase. Therefore, finding a way for an efficient resource allocation is challenging, because the solution should consider not only the increasing amount of data, but also possible fluctuations due to the environmental interactions.

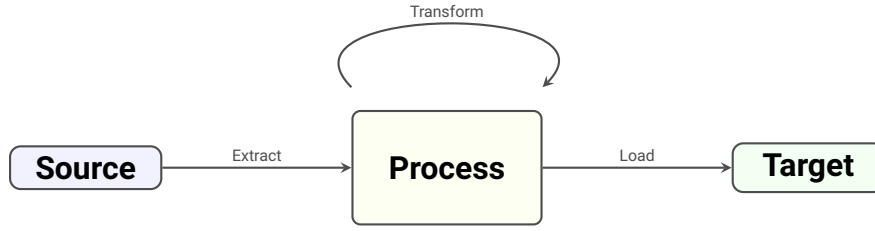


Figure 3.1: Visualization of processing phases for a single ETL process. Source contains the data to be processed, which gets extracted by the ETL process. Then, the data gets transformed inside the ETL process. In the end, the processed data gets loaded into the target.

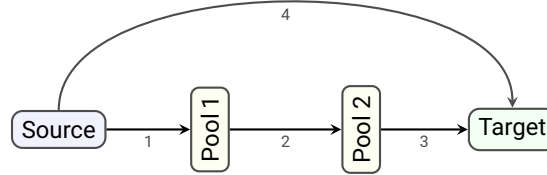


Figure 3.2: An example ETL pipeline with 4 cores and 2 internal pools. The network processes the data from source pool to target pool, where rotated nodes indicate the internal pools. Edges indicate the ETL cores that provide the connection between each pool, where the numbers indicate the core orders. First cores are used to form a chain structured path from source to target node, and the remaining cores are sampled from the possible paths.

[2] provides a mathematical formulation to model ETL dynamics for simulating the characteristics of real ETL system behavior. It models complete chain of ETL processes, including the throughput formulations that also represent the resource allocation problem with parallelism.

We briefly present the mathematical formulation proposed in [2]. The relationship of parallel threads a with mean throughput $T(a)$ and mean processing time $R(a)$ is introduced in Equation 3.1.

$$a = T(a) \cdot R(a) \quad \forall a \in [0, \infty) \quad (3.1)$$

Additionally, it is expected that the mean throughput increases with more threads available,

but not infinitely. Therefore, a bound $T_{\max}$ is defined in Equation 3.2.

$$\lim_{a \rightarrow \infty} T(a) = T_{\max} < \infty \quad (3.2)$$

To control the saturation of the mean throughput with the amount of parallel threads, the mean throughput is modeled as a curve in exponential family with a scaling constant k , holding the property defined in Equation 3.3.

$$f : [0, \infty) \longrightarrow [0, \infty), \quad a \mapsto T_{\max} \left(1 - e^{-a/k}\right), \quad T_{\max}, k \in (0, \infty) \quad (3.3)$$

For modeling the mean processing time $R(a)$ while capturing stochastic noise, gamma distribution Γ with distribution shape α is presented as a possible method using the formulations in Equations 3.4 and 3.5.

$$\hat{R}(a) \sim \Gamma(\alpha(a), \theta(a)), \quad a > 0, \alpha(a) > 0, \theta(a) = \frac{R(a)}{\alpha(a)} \quad (3.4)$$

$$\mathbb{E}[\hat{R}(a)] = R(a) \quad \forall a \in (0, \infty) \quad (3.5)$$

These formulations define the distribution shape α to control the variance of the response times. As a result, the variance of the response times is tied to the value of α , where lower values indicate higher variance and therefore higher stochastic noise.

Capturing the various possibilities of ETL processes, the hyperparameters $T_{\max}$, k and α are sampled from uniform distributions with specified lower and upper bounds to model the system completely.

3.2 Reinforcement Learning

RL describes the machine learning paradigm, where an autonomous agent tries to make decisions to maximize a cumulative scalar reward signal. Unlike supervised learning, the learning is not performed through already existing demonstrations or examples, but rather through trial-and-error in the environment while also accounting for delayed rewards [1]. The interaction with the environment is formalized with a Markov Decision Process (MDP), including states S_t , actions A_t and rewards R_t . A simple interaction between an agent and an environment is visualized in Figure 3.3. As a result of the optimization, the

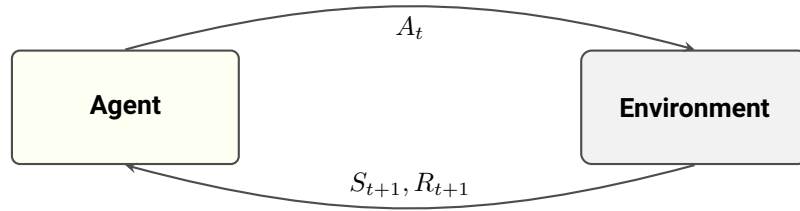


Figure 3.3: Interaction between agent and environment with MDP components. Agent chooses the action A_t to act in the environment. The environment provides the next state S_{t+1} and the reward received R_{t+1} as a result.

agent aims to find an optimal policy, defining the optimal action to take given the current state.

Generally, RL algorithms can be separated into three categories, value-based, policy-based, and actor-critic methods. Value-based methods involve learning of optimal policy by approximating the estimated return value while being in a specific state, or taking a certain action, whereas policy-based methods directly adjust the policy instead of deriving a policy from value estimates. Actor-critic methods combine the approaches of value-based and policy-based methods to evaluate current policy using the value function and supply this value to update the policy estimation. Usually, algorithms making use of policy estimation provide advantage in environments requiring stochastic policies or involving continuous action spaces, as approximation of value function might be computationally complex [1].

As a state-of-the-art policy gradient method, Proximal Policy Optimization (PPO) [13] is widely adopted in current research. It builds on the principle introduced with Trust Region Policy Optimization (TRPO) [14], which ensures that policy updates remain within a trust region of previous policy. Instead of formulating a constrained optimization problem, PPO achieves this with a simple loss function that can be more efficiently optimized. Additionally, PPO implements a clipped surrogate objective function to limit the step sizes of the policy updates, making it more stable during the training process compared to TRPO. As PPO is successful to reliably converge with stable policy updates, it is used as the core algorithm for this work.

3.3 Neural Architectures

3.3.1 Long Short-Term Memory

In most real-world applications, data is not completely independent and identically distributed (i.i.d.) as often assumed in theoretical works. Usually, the agent's current observation highly depends on the history of preceding observations, making the data inherently sequential. Just like in natural language processing (NLP), existence of some words in sentence could change the overall meaning drastically, possibly affecting the subsequent words' meaning. In both cases, the context information needs to be processed from the sequential data to be able to capture the dynamics of the environment, which can be done with an architecture with ability to recognize patterns across sequences.

To address the limitation of vanilla feedforward neural networks, recurrent connections inside the network structures are introduced. Making use of these connections, Recurrent Neural Network (RNN) allows the information from previous step in a sequence to be preserved initially as a hidden state for modeling sequences [15]. By doing so, the network can treat every element in context and maintain an internal memory. Figure 3.4 shows an example how RNNs unroll with a sequence of inputs. However, it is a challenge to train RNNs with long sequences due to the vanishing and exploding gradient problems [16]. Gradients are repeatedly multiplied with the weights in each timestep during the backpropagation, and with small weights, the gradient declines drastically, making the model fail to learn long-term dependencies.

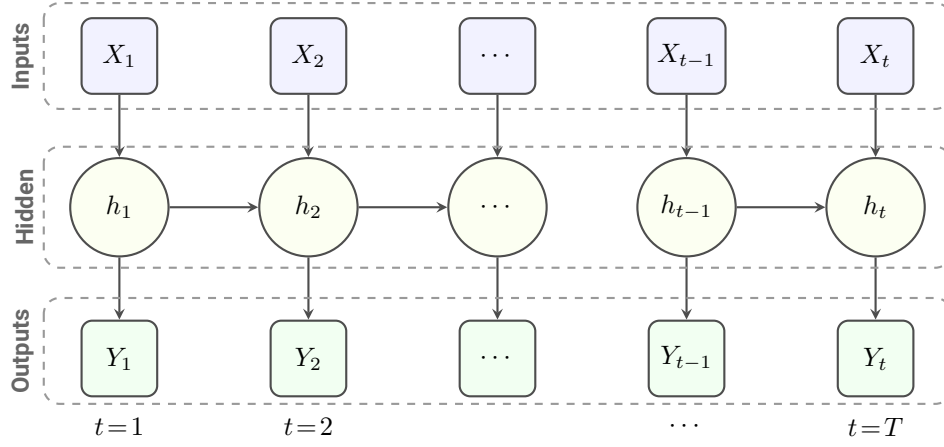


Figure 3.4: Visualization of an unrolled RNN structure for a sequence with timesteps $t = 1, \dots, T$. In each timestep, network receives input X_t , updates the hidden state h_t and produces an output Y_t . To capture the context over the sequence, network preserves its previous hidden state h_{t-1} on each timestep.

For effectively remembering the earlier information in the sequences, [17] introduces Long Short-Term Memory (LSTM) networks. In this architecture, the inner state of each cell is reinterpreted to ensure gradients are stable even after many timesteps. This stability is enabled by gating mechanisms, controlling whether the information should be discarded from memory, added to the memory or passed to the next hidden state. As the cell states are propagated with additive operators instead of multiplicative, LSTMs can maintain the context of observations over longer sequences without vanishing gradients.

3.3.2 Transformers

Although LSTMs can capture the context of sequential data using a hidden state, their architecture is sequential by design. The final form of hidden state needs to be processed step by step while depending on the previous hidden state, making the processing hard to parallelize and inefficient. Moreover, all the sequence information gets compressed in a fixed-size hidden state, which can limit the expressiveness while dealing with long sequences. [18] reconsiders this hidden state approach and proposes Transformers. This new design enables processing all elements of sequences simultaneously and in parallel, and can still capture the general dynamics of the inputs.

As the core of Transformer architecture, attention mechanism allows calculation of the importance of elements inside the sequence. More specifically, it produces a matrix to indicate pairwise importance weights, connecting the elements of sequences that are jointly relevant. In the original implementation of the attention mechanism *scaled dot-product attention* as formulated in Equation 3.6, three major components are used: queries Q , keys K and values V , where d_k indicates the dimensions of key vector.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \quad (3.6)$$

With the multi-head attention, the model is able to capture complex data structures using multiple attention mechanisms in parallel. Each head of the attention can specialize to capture different semantics within the sequence, enabling to build the context in more detail. As the attention mechanism considers the inputs without a sense of ordering, the inputs must be processed to incorporate their positional information. This process is done with the positional encoding just before the encoder and decoder blocks of Transformers. All these components and the network structure of Transformers are visualized in Figure 3.5. Ultimately, Transformers are considered as more efficient and parallelizable context-aware neural architectures.

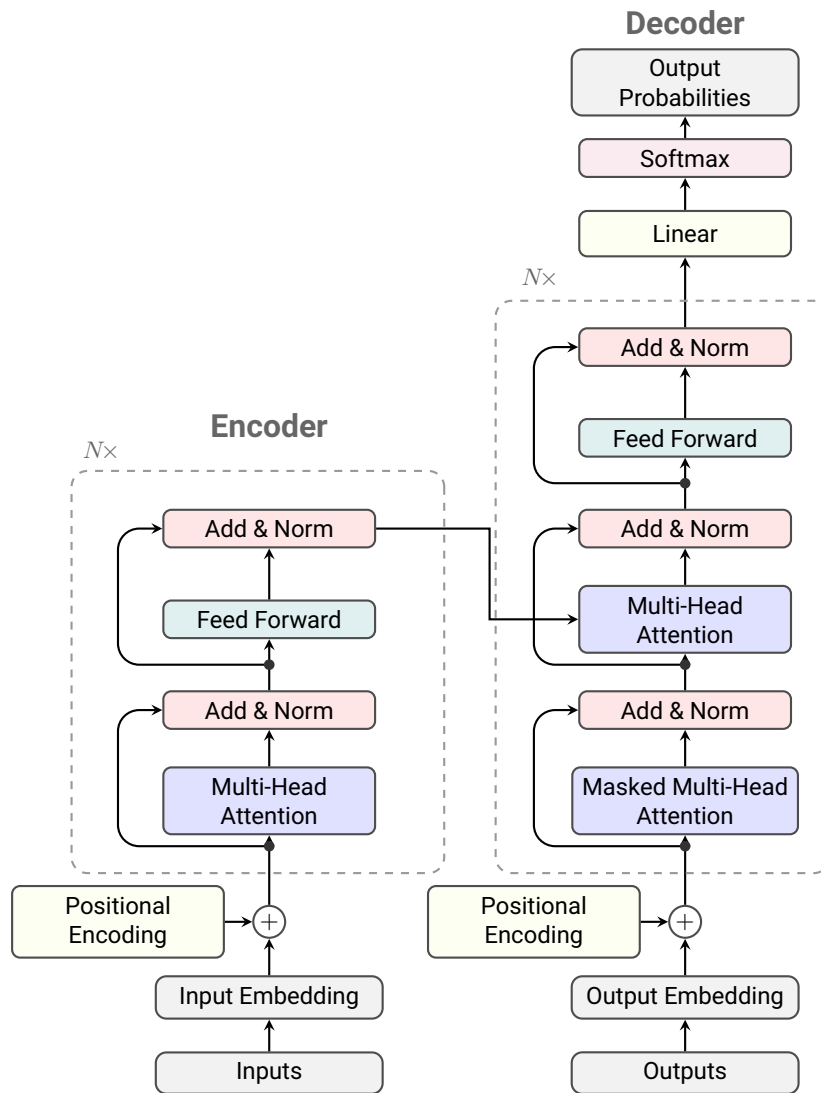


Figure 3.5: Transformer architecture with its two components: Encoder and Decoder. Encoder propagates the overall context across the sequences with attention, which then gets used in the second attention layer of Decoder.

3.3.3 Graph Neural Networks

In contrast to the sequential data discussed in the previous neural architectures, many real-world elements can be in a complex relationship with each other, like biological models or social networks representable with graphs. When using these objects directly with traditional machine learning approaches, most common approach is to represent the relationships into lists of numbers. Although this process can preserve each entity's information, the topological dependencies and relational context between entities disappear. As a way to directly encode a graph's connections, [19] introduces the Graph Neural Network (GNN) architecture.

This architecture employs a mechanism called *information diffusion* inside the encoding network for modeling the irregular dependencies inside the data. In the graph, each node is considered as a standalone unit maintaining an internal state, and allowed to exchange information with neighbors. Repeating this process iteratively across all edges, the state on each node gets updated with the feature and position information coming from its neighbors. By propagating the neighbor messages across multiple layers, the network can represent the dependencies between nodes in the graph.

Although this theoretical concept is a rigid foundation to realize GNNs, modern approaches improve this into more efficient approach called *message passing* introduced by [20]. Here, the diffusion is realized with two steps in loop. First step is called *aggregation*, where each node gathers information from its neighbors. Then, with the *update* step, the summary of the information provided by the neighbors gets propagated into the node's own state. With this advancement, model can capture the relational patterns just like the information diffusion, while maintaining an efficiency to suit better for deep neural networks trained on large datasets.

Using this scheme, the instantiations of GNN such as graph convolutional networks [21] and graph attention networks [22] have become standards to learn from data with graph-structured representations.

3.4 Unified Robot Morphology Architecture

The characteristics of an open-world environment inherently exist in robotics, as the different types of robot embodiments present different number of joints, resulting in different dimensions of observation and action spaces. The variability of embodiments

even requires recalibration of policy parameters to be applied in multiple robots with similar embodiment type, let alone using the same policy across different types of robots. Existing methods to handle the multitask RL problem consider padding the observation and action spaces with zeros until a maximum length, or require training a separate head per task. Although it is efficient to realize training in such ways, transferring the policy into unseen robots performs poorly, as the policy would need to learn the new robot's kinematics in relation to the observation and action spaces. For learning a general robot locomotion policy, [23] introduces Unified Robot Morphology Architecture (URMA).

Instead of having the architecture depending on predefined morphologies or joints, it learns a common locomotion representation that is shared across various robot types with learnable encoders and universal decoder. The network structure contains learned representations for latent observation and latent action vectors, enabling the core policy to work even with variable number of joints and observation sizes. The latent observation is constructed with an encoder using an architecture similar to Transformer, which gets provided into the core policy together with the general observation. After that, output of core policy gets supplied to the decoder network to retrieve actions for each joint separately. This process is visualized by the architecture diagram in Figure 3.6.

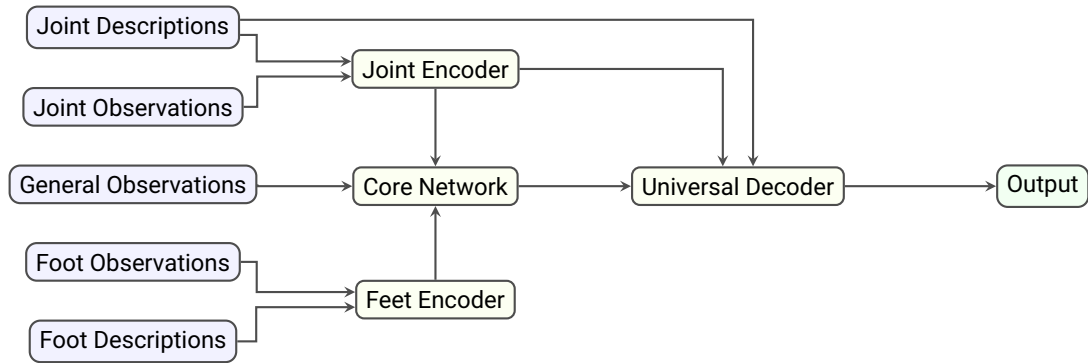


Figure 3.6: The overview of URMA architecture. Robot joints and feet information are processed in their separate encoder networks. The results from these two encoders are provided into core network with the general observations, producing the latent action vector. The decoder uses this latent action vector, the weighted core observation from joint encoder and joint descriptions to output the action values.

Taking a more detailed look into the input structure, the observation vector consists

of two separate observation parts: robot-specific and general observations. The robot-specific observation gets further split into joint-specific and feet-specific observations. These observations contain joint-specific locomotion information like position, velocity, previous actions and other properties. More importantly, each observation is paired with a fixed-size description vector, characterizing each joint using the joint-related properties such as rotation axis, number of child joints, nominal position. As the descriptions are independent of the morphology and fixed-size, the network utilizes them to learn a stable representation that is invariant to the differences of robots.

For robots, the description vectors define distinctive characteristics for each joint of robot. With these characteristics, the policy network can perform more efficient and more successful learning to generalize over unseen embodiments as well. In contrast to robotics, the ETL pipelines consist of multiple ETL cores, which are assumed to be identical. Therefore, the description vectors using only the properties of a core might not be distinctive enough to observe similar performance improvements while using an approach based on URMA to control ETL systems. Additionally, the generalization still depends on the training distribution, as the policy evaluations on embodiments that are significantly different from the training data do not reliably transfer.

3.5 AgentFormer

In problems with multiple agents, the relationship between agents, and uncertainty in the future make the trajectory modeling difficult. Such systems usually model both the time dimension, capturing the effects of past states on future states, and the social dimension, which captures the relationships between each agent. It is possible to initially process one dimension, and use a latent representation to process the other one, but this might cause information loss.

For preserving most of the information, [24] proposes a special Transformer architecture called AgentFormer. The main idea is to formalize a way to model both dimensions at the same time, allowing an agent's state at a time directly affect another agent's state in the future. Usually, Transformer architectures flatten all information in a sequence, losing the order of items passed to the attention. The time information is dealt with adding a separate positional encoding related to the time into the inputs. However, the identity of agents is not always easy to order and might cause additional dependencies like agent indices.

$$\text{AgentAwareAttention}(Q, K, V) = \text{softmax}\left(\frac{A}{\sqrt{d_k}}\right)V \quad (3.7)$$

$$A = M \odot (Q_{\text{self}}K_{\text{self}}^T) + (1 - M) \odot (Q_{\text{other}}K_{\text{other}}^T) \quad (3.8)$$

$$Q_{\text{self}} = QW_{\text{self}}^Q \quad K_{\text{self}} = KW_{\text{self}}^K \quad (3.9)$$

$$Q_{\text{other}} = QW_{\text{other}}^Q \quad K_{\text{other}} = KW_{\text{other}}^K \quad (3.10)$$

Therefore, AgentFormer architecture uses a special attention mechanism called *agent-aware attention*, which is formulated in Equation 3.7. Attention weights used in this attention mechanism are calculated according to Equation 3.8, which utilizes a mask M to switch between two sub-components of the attention weights. The mask matrix results in 1 for the cells which have query and key vectors belonging to the same agent and 0 otherwise. This attention mechanism uses two sets of key and query vectors using various transformations to model two separate attentions: agent-to-agent and agent-to-itself. Agent-to-agent attention describes the relationship between different agents as formulated in Equation 3.10, whereas agent-to-itself captures the informational dependency on the same agent as stated in Equation 3.9. With this definition, the structure of AgentFormer transformer follows the same construction as Transformer illustrated in Figure 3.5, with the exception of the agent-aware attention mechanism.

4 Methodology

In this chapter, the methodology is described for exploring possible solutions for the open-world RL problem. First, the problem will be defined using ETL systems as an example of an open-world RL environment, followed by the exact environmental formulation and properties. Then, the experimental approaches for the policy architectures are presented in detail. Lastly, the evaluation criteria are defined for assessing the properties of the expected solution.

4.1 Problem Formulation

As an example of open-world RL problem, this thesis studies the learning of a single reusable RL policy to control ETL pipelines. The learning task can be formulated as a Markov Decision Process (MDP) given the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma)$. In this tuple, $\mathcal{S}$ corresponds to the state space, $\mathcal{A}$ the action space, $\mathcal{R}$ the reward function, $\mathcal{P}$ the transition dynamics, and $\gamma \in [0, 1)$ the discount factor. Using this formulation, the learned policy $\pi(A_t | S_t)$ provides the action A_t given the current state S_t to maximize the expected discounted return.

State space $\mathcal{S}$ is constructed from information of each core; therefore, the dimension of state space changes with the number of cores N . This information contains the last actions; the average fetch, enrich and commit cycles, as well as the delta of documents committed, enriched and fetched. The size of action space $\mathcal{A}$ also depends on the number of cores, as each core tries to balance the thread utilization for enrich and commit phases. Therefore, the dimensions of both state and action spaces can be realized as functions of the number of cores, as described in Equation 4.1.

$$\dim(\mathcal{S}_t) = f_{\mathcal{S}}(N) \quad \dim(\mathcal{A}_t) = f_{\mathcal{A}}(N) \quad (4.1)$$

Although traditional RL approaches consider the dimensions of state and action spaces fixed with the usage of the policy network, this is not always the case with open-world problems like the current problem formulation about ETL pipeline. Therefore, this thesis will study the methods compatible with various configurations of the policy networks, so that the learned policy can be used with any number of cores without requiring retraining. The goal of the trained policy is described as expectation over the pipeline distribution $\mathcal{G}$ in Equation 4.2, where R_t represents the reward received at timestep t .

$$\pi^* = \arg \max_{\pi} \mathbb{E}_{G \sim \mathcal{G}} \left[\mathbb{E} \left[\sum_{t=0}^T \gamma^t R_t \mid G, \pi \right] \right] \quad (4.2)$$

This formulation with distribution of pipelines ensures the generalization of the learned policy across changing topologies of ETL systems. To achieve this objective, the environment setup, reward formulation and policy architectures are described in the following sections.

4.2 Environment Setup

The environment is represented as an ETL pipeline with the modeling of a directed acyclic graph, where the nodes V of graph are the data pools and the edges E are ETL cores that transfer the data between the pools. In the pools, there are always a source pool providing data for processing, and a target pool receiving the resulting documents from the pipeline. In addition to these pools, the pipeline might contain internal pools located between the cores. The structure of pipeline guarantees the unidirectional flow of data, making the data proceed towards the target on every processing step. The construction of graph is done with pre-defined number of cores and a number of internal pools, where the cores initially connect the pools as a chain structure. Then, the remaining cores are sampled from valid edge possibilities in the pipeline model, without having the same connection between pools.

In the simulation environment, the source pool provides an infinite amount of data, which can be fetched using the ETL cores. Similarly, the target pool can have infinite number of documents committed. However, each internal data pool can only hold a finite amount of data, which is configured as a hyperparameter to hold 100 documents. In the initialization of the pipeline, the number of documents occupying these internal pool slots is randomly assigned.

Cores are considered as agents, and they represent the edges in the ETL pipeline graph model. Every core has an internal queue with size of 100 documents just like the internal pools to hold documents after the enrich action. The fetching action happens automatically on each core in batches with size of 100 documents, when the number of documents that were fetched but not processed drops down to a configured value called *fetch limit*, which is configured to 200 documents. Additionally, another precondition for the fetch process is determined by the *fetching threshold*, calculated with number of queued documents divided by the queue size which needs to be lower than 70%. Number of cycles for the fetch processes is also bounded between 1 and 2. Lastly, the cores are initialized with 10% of threads allocated for enriching and committing in the beginning.

The dynamics of the environment follow the mathematical formulation of [2] as described in Section 3.1. This formulation provides three hyperparameters $T_{\max}$, k and α to model the throughput of every core using the response times, while considering the resource allocation problem. These hyperparameters are sampled independently within their uniform bounds in Equation 4.3, providing a heterogeneity for cores. Because the real deployment could consist of not only fast and low-variance, but also slow and noisy cores, the randomness can represent a nominal pipeline and challenging environments to deal with. All environment hyperparameters can be found in Table 4.1.

$$T_{\max} \sim \mathcal{U}(0.001, 10000) \quad k \sim \mathcal{U}(0.001, 0.01) \quad \alpha \sim \mathcal{U}(1, 5) \quad (4.3)$$

The cores are the edges of the original graph formulation of problem, but the optimization is performed for each core individually. Therefore, the influence of each core is captured using the *line graph* of pipeline model, where every core becomes a node, and the edges encode the direct connections between the cores. As the original problem’s graph structure is a directed acyclic graph, a core is considered to have a direct connection to another core, when both cores share a common pool such that it is the target of first core, and source of the second core. With this line graph formulation, the adjacency matrix of the cores is encoded and provided to the policies as prior information.

Similar to the joint description vectors used in [23] as described in Section 3.4, the core description vectors are defined by the specific features of the cores. Normally, the system needs to assume that every core has the same process, and their response times can already be inferred through the core-based observations. Therefore, the core description vectors encode graph-related information about the cores. They include the normalized distance between the source pool and the target pool, as well as the reachability statistics of each core. The reachability statistics provide the information about all possible paths

Hyperparameter	Value
N (number of cores)	4
P (number of internal pools)	2
action_scale	50
action_offset	1
queue_size	100
internal_pool_size	100
fetch_batch_size	100
fetch_limit	200
fetch_threshold	0.7
fetch_min_wait	1
fetch_max_wait	2
init_enrich_threads	10
init_commit_threads	10
num_sim_steps	1000
randomize_internal_pool	True

Table 4.1: Hyperparameters of the ETL simulation environment.

to reach from one core to another while considering the branches on the path. With this formulation, the core description vectors always result in the same size while still capturing the relevant distinctive information about each core, which is important to properly incorporate this information in the policy networks.

This environment is used on *Brax* [25] simulation framework, providing a convenient way to parallelize rollouts and use a custom environment interface. Multiple environments and batches are constructed in parallel with the domain randomization strategy for generalization [5], which is also further described in Section 4.5. As a further modification, the graph topology information and the description vectors are also made available for each call of the policy function, ensuring that the policy network implementations can use these prior values easily.

4.3 Reward Formulation

In the ideal situation with the optimal performance, the cores should maximize the overall processed documents of the pipeline while allocating fewer threads for the actions. This behavior ensures both increased efficiency of the cores and higher throughput of the pipeline. Additionally, the policy should keep the thread allocation stable enough to avoid unnecessary allocation changes of the cores. Combining all these objectives, the reward R at each timestep t is formulated in Equation 4.4.

$$R_t = w_C \cdot \Delta C_t - w_A \cdot \bar{A}_t - w_S \cdot \overline{\Delta A}_t \quad (4.4)$$

Equation 4.4 uses three different components of reward, while each being weighted with the coefficients w_C for number of newly committed documents of pipeline ΔC_t , w_A for the mean action value of all cores $\bar{A}_t$ and w_S indicating the weight of stability calculated with the mean change of actions of all cores $\overline{\Delta A}_t$. The number of committed documents by pipeline ΔC_t is calculated with summing up all the newly committed documents from the cores which put the processed documents to the target pool.

The first term ΔC_t encourages the policy network to output high number of documents into the target pool. This term is the only positive signal of the reward process; therefore, it is the main objective of every agent. The second term $\bar{A}_t$ penalizes the excessive amount of thread utilization on each core by using a negative weighting coefficient $-w_A$. As the last reward component, $\overline{\Delta A}_t$ penalizes the constantly changing thread utilization values to

obtain a smooth policy output, contributing to the stability. The weights for these reward terms are chosen to keep the resource allocation problem balanced with the throughput of pipeline, as represented in Table 4.2.

Coefficient	Value
w_C	1×10^{-5}
w_A	5×10^{-4}
w_S	1×10^{-2}

Table 4.2: Weight coefficients of the reward function in Equation 4.4.

4.4 Policy Networks

4.4.1 URMA-based Policies

For realizing dynamic number of cores of an ETL pipeline using a single policy network, this policy architecture uses the ideas of URMA as described in Section 3.4 by reinterpreting variable robot morphologies as variable ETL pipelines. With this consideration, the policy network must handle the varying number of cores instead of handling locomotion task across different embodiments. This realization keeps the three main parts of the policy network from original approach: encoder, core policy and decoder.

Although URMA-based policy contains 3 components in complete form, the architecture can be applied to current problem formulation in two ways, differing only in whether the encoder is present. Decoder allows for both variants to produce per-core action, which relies on a latent observation processed with the temporal encoding component. The hyperparameters for both approaches are listed in Table 4.3.

Decoder-only Policy

Decoder-only URMA network considers the fixed-size observations about the pipeline state as the latent observation input for the temporal encoding component. These observations

Hyperparameter	Value
encoder_mask_hidden	(16, 16)
encoder_core_hidden	(8, 8)
action_latent_hidden	(8, 8)
latent_encoder	Transformer
latent_size	48
transformer_feature_size	8
transformer_ffn	(32,)
transformer_num_layers	4
transformer_num_heads	4
decoder_hidden	(8,)

Table 4.3: Hyperparameters of the URMA-based policy network.

contain the average fetch, enrich and commit cycles of the pipeline with the delta number of committed documents.

In the *core policy*, the latent observation input gets processed with respect to the history to produce a latent action vector. This temporal processing allows capturing the memory of previous observations, helping to increase overall performance. To handle such temporal relationships, the architecture can be configured to use LSTM with internal memory as described in Section 3.3.1, or Transformer with an attention-based alternative as described in Section 3.3.2. As a result, the output of this temporal processing architecture summarizes the history of observations and provides a single latent vector describing the general action that the pipeline should take.

The latent action description is then processed with the *decoder*, concatenated with the core observation. Additionally, the decoder can be configured to use core description vectors during the concatenation. This option creates two different policy approaches: decoder-only URMA policy without prior (URMA-D) and decoder-only URMA policy with prior (URMA-D-Prior). With the resulting vector, decoder applies a feedforward network to compute core-based resource allocations separately. The architectures of URMA-D and

URMA-D-Prior are visualized in Figure 4.1, and in Figure 4.2 respectively.

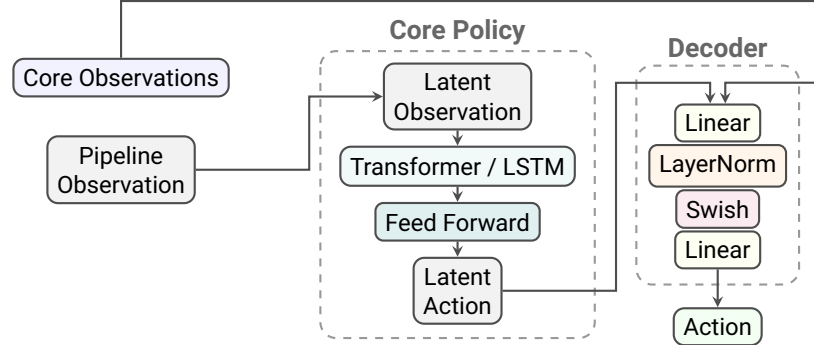


Figure 4.1: Overview of URMA-D. The pipeline observation is provided into the core policy to generate a latent action vector. Decoder takes each core's observation combined with the latent action vector and processes them through a feed-forward network to produce action for each core.

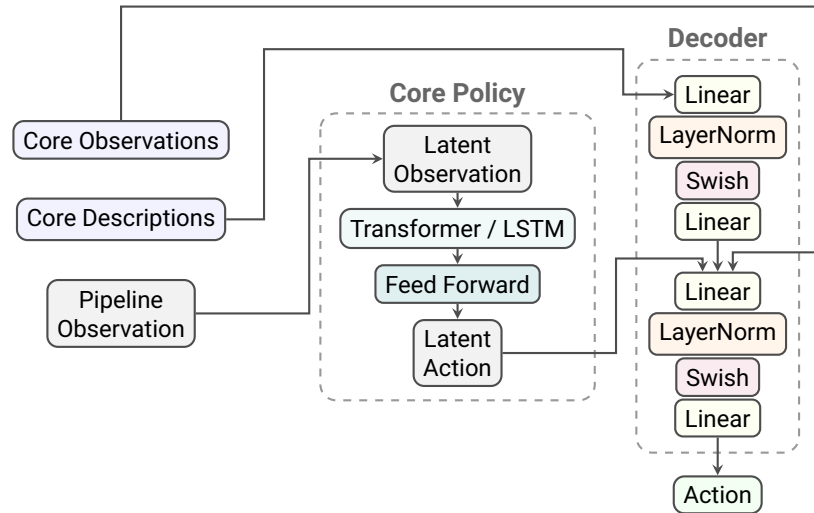


Figure 4.2: Overview of URMA-D-Prior. Core policy processes the pipeline observation to generate a latent action vector. Decoder transforms the core description vectors into an action description vector using a feedforward network. Combining the action description vector with latent action vector and core observations, a feedforward network produces action for each core.

Encoder and Decoder Policy

Using all main components of the decoder-only URMA-based policy, URMA policy with encoder and decoder (URMA-ED) contains the *encoder* in the beginning of the network, allowing the construction of fixed-length latent observation using attention-like mechanism. Here, every core is processed with its local observations and description vector, encoding the topological information of each core. Because of fixed size nature of description vectors, the policy can learn a network to weigh each core's contribution to the general state. This approach produces a latent observation with fixed-size elements that does not depend on the number of cores or network topology.

As a key difference in decoder, the latent action description gets concatenated with the weighted core observations instead of the normal core observations, as well as the core description vectors to construct the final action for each core. Each component of the resulting concatenation provides key information about the pipeline state and core-specific local information to produce more informed per-core decisions and generalize across topologies. The resulting policy architecture URMA-ED is visualized in Figure 4.3.

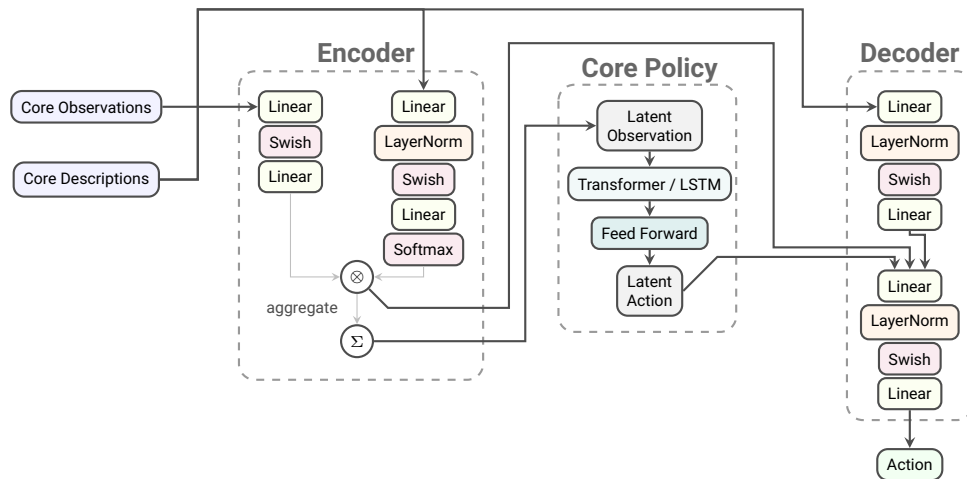


Figure 4.3: Overview of URMA-ED. Encoder summarizes core observations together with core descriptions to mask important fields and aggregate core observations, producing the latent observation. Core policy processes latent observation across the temporal axis to produce the latent action. Combining latent action with weighted core observations and core descriptions, decoder produces action for each core.

4.4.2 Graph Neural Network Policy

To exploit the graph structure of the ETL pipeline, GNN policy uses the theory provided in Section 3.3.3 and builds on the message passing method, as visualized in Figure 4.4. With the encoding done as a graph keeping the same topological relationship, the information gets propagated across the edges. Since the message passing logic works with a variable number of cores and different graph topology, this policy network can also be evaluated with different number of cores and topology than the trained pipeline configuration.

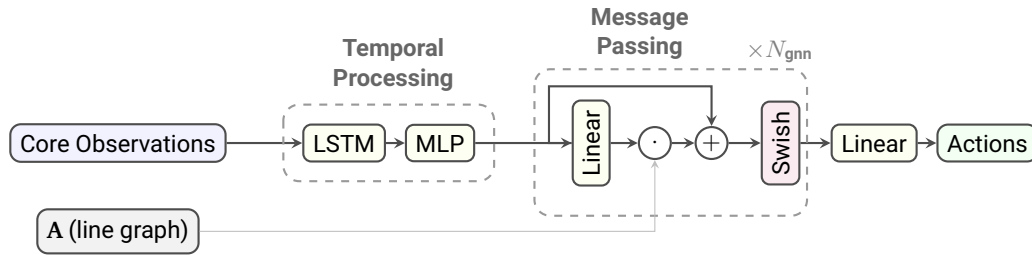


Figure 4.4: Overview of GNN policy. Observation history of each core gets processed through the same temporal processing component. The last history information is provided into the message passing section, which aggregates and updates each core’s information. Results of the message passing are mapped to the action dimension separately for each core.

As the vanilla GNN does not provide a way to deal with sequential data, the observation history needs to be processed separately before utilizing message passing. Therefore, temporal encoding with LSTM gets initially applied to capture the context of observations on each timestep for each core, which summarizes the core-based history. Then, this summarization is used as the fixed-size input for each core inside the graph.

Using the temporal encoding from the initial step, a graph is constructed by taking the line graph of original ETL pipeline. This graph encodes each core as a node, taking the temporally encoded observations initially as their first information. For capturing topological relationship from the ETL pipeline, the edges of line graph represent the direct connections between the cores, which only exist when a core’s target is another core’s source in the ETL pipeline. With this network formulation, the dependencies of each core become structured, making it ready to use on a GNN approach.

The network formulation is used with the message passing mechanism for exchanging information across the cores. The process follows the aggregation and update steps of the message passing mechanism as described in Section 3.3.3, where each node gathers the information projection of its neighboring nodes in the aggregation step, and merges them with its own information in the update step. Repeating this process for several layers makes the information flow across several hops in the network, allowing each node in the path to consider the information of neighbors.

After the information flow with the message passing mechanism, the resulting core-based information from message passing gets mapped into the desired thread allocation separately. As the information size is fixed through the whole process, this part is also compatible across variable number of cores.

The hyperparameters for GNN policy are shown in Table 4.4.

Hyperparameter	Value
<code>lstm_layers</code>	$5 \times (20,)$
<code>mlp_layers</code>	$4 \times (32,)$
N_{gnn}	2
<code>hidden_dim</code>	64
<code>normalize_adj</code>	True

Table 4.4: Hyperparameters of the Graph Neural Network policy.

4.4.3 Transformer-based Policy

Building on top of the attention mechanism and encoder structure as explained in Section 3.3.2, this policy processes the cores and their observation history with an encoder-only Transformer architecture. Attention mechanism enables calculating dependency and interactions of each sequence item in a single layer as a quadratic table in number of cores. The overall architecture of Transformer policy is illustrated in Figure 4.5.

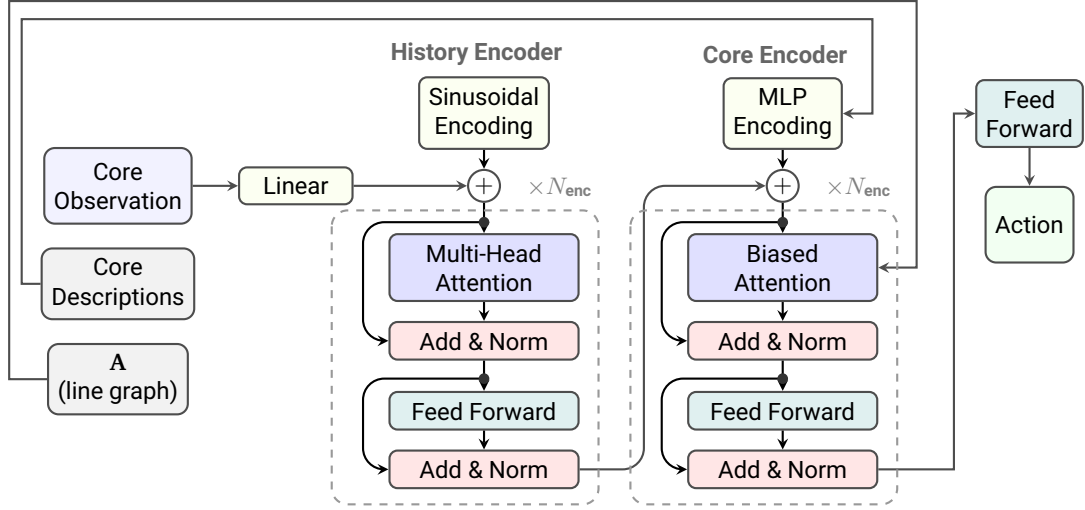


Figure 4.5: Overview of Transformer-based policy. Two different transformer encoders, history and core encoders, process each axis of the core information after the linear projection into the feature dimensions. History encoder uses sinusoidal positional encoding and the original Transformer structure. Core encoder calculates the positional encoding using a feedforward neural network from the core description, and uses the core adjacency matrix to add a bias to the attention. Then, another feedforward neural network calculates the final action values for each core.

The network is structured to have two separate Transformer encoders to handle two temporal dimensions in the core observations separately. The first temporal dimension is the observation history of each core, while the second one is the between-core dependencies of the ETL pipeline. In these both dimensions, attention mechanism allows capturing both the observation history and the connections between the cores.

This policy architecture only includes Transformer encoders and not decoders, as it is expected to produce a fixed-size action for each core instead of generating an output sequentially. During the processing of first temporal dimension, the observation history, the network gets sinusoidal positional encoding which was also mentioned in the original implementation by [18]. As the ordering of observation history is always from past to future, this positional encoding can be helpful for Transformer network to understand each history step separately. However, for second temporal dimension, the ordering of cores cannot be explained as simply as the observation history due to the graphical relationship.

Therefore, the position embeddings are calculated from the core description vectors using a feedforward neural network with learned weights.

Additionally, the pipeline structure is exploited in another way by modifying the attention calculation on the second temporal dimension. Following the similar formula as the original implementation, a learned bias variable is added on the connected cores according to the pipeline topology in the attention, just before calculation of softmax values. This bias provides a structural prior for the connections that are actually present in the ETL pipeline, while reducing the scores of independent cores without any hard-masking procedures. With the core adjacency matrix A and learned bias values b_c and b_n for directly connected and not connected cores, attention is calculated as per Equation 4.5 with the elements of bias matrix B defined in Equation 4.6.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + B\right)V \quad (4.5)$$

$$B_{ij} = \begin{cases} b_c & \text{if } A_{ij} = 1 \\ b_n & \text{otherwise} \end{cases} \quad (4.6)$$

After both encoders process the observations, the resulting information is passed through a network to calculate per-core throughput allocations. Transformer architecture already projects the core observations into a vector with size of features, which also works without any changes on different number of cores than the trained pipeline configuration and various ETL pipeline structures.

All hyperparameters for Transformer-based policy are provided in Table 4.5.

4.4.4 AgentFormer-based Policy

Using the agent-aware attention mechanism introduced by [24], this policy acts with a single Transformer encoder that can capture both the contextual and agent-wise dependencies as described in Section 3.5. Compared to a normal Transformer encoder, this custom attention uses an additional logic to not only attend each core's observations individually, but also affect the calculated attention score between each agent.

The observation initially gets projected into the feature space and flattened for each timestep, which can be viewed as the whole ETL pipeline's observation at a certain

timestep. Subsequently, the positional embeddings of observation history get appended to this projection, and forwarded to the encoder blocks for further processing by using agent-aware attention. Unlike the former Transformer-based approach, the current approach does not consider any structural prior on attention logits or the positional encoding; therefore, it heavily relies on the training examples for the structural understanding.

After the encoding process, the result gets reshaped to separate each core’s information and passed through a feedforward neural network to calculate action values. All hyperparameters for AgentFormer-based policy are the same as Transformer-based approach, which is given in Table 4.5.

Hyperparameter	Value
num_layers_encoder	4
num_heads	4
features	8
pos_ffn_hidden	(32, 32)
policy_ffn_hidden	(32,)
value_ffn_hidden	(32,)
learned_positional	False (sinusoidal)

Table 4.5: Hyperparameters of AgentFormer-based and Transformer-based policy networks.

4.5 Training Setup

For an actor-critic learning process, the training happens with the optimization of two separate networks: policy and value network. The value network for each type of policy follows the same architecture as the policy network, only having an extra linear layer in the end to predict a single value for the network performance. Additionally, the differences for the intermediate layer sizes are stated in each policy network’s hyperparameters table.

All training process is done using PPO algorithm as described in Section 3.2, which is already provided with *Brax* [25] simulation framework. The hyperparameters used for the PPO algorithm are listed in Table 4.6. The existing implementation provides the possibility to normalize the observations, which stores running statistics of every observation element as a Gaussian distribution while updating the mean and variance values through the observations on training process. To handle variable observation sizes, this approach is modified to only store the mean and variance values with respect to single core's observation, whereas every core separately updates the mean and variance values of single core's observation.

Generalization of the policy is based on randomized topologies on every reset of the environment. On each reset, the pipeline gets reconstructed with a different topology, also randomizing the core and pool based properties as described in Section 4.2. As the agent observes multiple ETL pipeline topologies, it potentially generalizes to previously untrained topologies.

4.6 Evaluation Criteria

For comparison of different policy architectures, three different criteria are investigated: task performance, generalization across different pipelines and stability. They represent foundational components for ensuring smooth and effective control of an arbitrary ETL pipeline. Therefore, it is important that the proposed architectures can optimally control the cores while not heavily compromising these criteria.

The task performance is mainly evaluated with three metrics: reward, resource allocation and the delta action. The reward heavily consists of the pipeline data throughput, while slightly penalizing the high allocation of resources and instant changes of actions. Additionally, the number of pipeline's committed documents can be further investigated to compare the units of documents received by the target pool for a direct comparison between the approaches.

As important as the primary measure about the task performance, the policy should be able to generalize across unseen ETL pipeline configurations and keep the task performance optimal with the changing topological structure. To verify generalization, the results of every architecture are also evaluated with different number of cores, which also means different pipeline topologies. Using the results from these evaluations, the architectures

Hyperparameter	Value
num_timesteps	50,000
episode_length	250
num_envs	128
num_eval_envs	128
batch_size	48
unroll_length	4
num_minibatches	48
num_updates_per_batch	4
action_repeat	1
hist_length	5
learning_rate	1×10^{-4}
discounting (γ)	0.9
entropy_cost	1×10^{-4}
reward_scaling	1.0
normalize_observations	True
normalize_with_single_core	True
deterministic_eval	True
num_evals	10
seed	50

Table 4.6: Hyperparameters of the PPO training process.

are compared against both the baseline and the model trained with the same number of cores.

Lastly, as described in Section 4.2, the core configurations and the response times might vary a lot in the real world system. Therefore, an optimal policy should be able to stay consistent during the noisy processing of cores while allocating optimal amount of threads for each core to ensure the best throughput of pipeline. The stability can be evaluated in two ways: resource allocation stability and throughput stability. As the reward already consists of these two terms, it is relatively more apparent if the noisy nature of ETL processes affects the decision-making of policy networks. One more possibility is also to review the committed documents of pipeline and the resource allocation to check sharp changes for the values.

5 Experiments & Results

This section presents the empirical results of the experiments performed to verify the evaluation metrics. Initially, the experimental setup is presented, followed by the baseline policy definitions to define reference points for comparison. Finally, the evaluation results of trained policy architectures are reported under both the training configuration and an unseen configuration both quantitatively and qualitatively.

5.1 Experimental Setup

All experiments are performed with the environment, reward, and training specifications described in Chapter 4. As for the specific ETL pipeline during the training process, the number of cores and internal pools are configured as 4 cores and 2 internal pools. This pipeline configuration is small enough to still capture all local structure that each core appears in. As a result, the observation and action sizes were kept fixed to a constant value during the training process, while the evaluation can still be done with different number of cores.

The training configuration follows the explanations and hyperparameter definitions in Section 4.5. For a common ground of comparison, three policies are provided as baselines in the next section. These approaches are important to verify the architectural improvements on the evaluation metrics as described in Section 4.6.

5.2 Baselines

5.2.1 Delta Policy Heuristics

Delta Policy Heuristics (DPH) is a simple form of a non-learned rule that preserves the last number of documents committed by the pipeline and compares it with the current value. If the pipeline was able to commit more documents at current timestep, it applies the same delta action Δa of previous timestep. Otherwise, if fewer documents are committed, the actions are changed by $-\Delta a$. Following this definition, the policy $\pi^{\Delta\pi}$ is formulated in Equation 5.1, where a_t indicates the action value at timestep t .

$$\pi^{\Delta\pi} = \begin{cases} a_{t-1} + \Delta a_{t-1} & \text{if } \Delta C_t > \Delta C_{t-1} \\ a_{t-1} - \Delta a_{t-1} & \text{otherwise} \end{cases} \quad \Delta a_{t-1} = a_{t-1} - a_{t-2} \quad (5.1)$$

There is no separate control of actions for each core; therefore, same amount of resources are allocated for every core regardless of the amount needed for the best result. Using the same amount of action for all cores eventually works with low number of cores, although it might cause problems with larger pipelines that require more complex and independent resource allocation policy.

Because of the actions getting increased or decreased according to the documents committed, each timestep has an action difference bigger than 0. Therefore, the action allocation cannot stay fixed even for a short period; instead, it has a jitter pattern while approaching the highest number of committed documents. Δa is also an important value for this policy to get close to the best values, as it acts as a step size value of a numerical optimization strategy. With higher step size, there is risk to overshoot the best values, whereas smaller step size could take much longer for convergence.

5.2.2 Fixed Temporal Network

Fixed Temporal Network (FTN) directly processes the observations of all cores together to produce action vectors at once, which have a size determined during the training process. It includes an LSTM network to process the observation history of the pipeline, followed by a feedforward neural network to map final hidden state into the fixed-size action vector. Equivalently, the LSTM can be replaced with Transformer encoder for a comparable processing, as both can effectively summarize the temporal context.

As the observation and action vectors have fixed size, this policy cannot be used to evaluate pipelines that have different number of cores, making it quite limited for generalization. However, it is expected to reach a relatively good performance, because fixed size observation contains the information of all cores at the same time, allowing the neural network to learn relationships across the cores. The ordering of the cores might not be constant across the episodes except the first cores forming a chain structure in the pipeline. Therefore, the ordering limitation might slightly affect the performance.

5.2.3 Core Temporal Network

To allow evaluation across various pipelines with different number of cores than the training process, Core Temporal Network (CTN) processes each core one-by-one. It considers the core observations as a batch dimension instead of capturing the context of whole pipeline. Therefore, the temporal information of cores gets processed independently, and the final hidden state gets processed to get action vectors for each core separately, allowing evaluation of environments with variable number of cores.

Allowing the usage of variable number of cores, this baseline establishes a lower bound of generalization that a simple core-based neural network can achieve. It also makes possible to interpret the performance gaps between the relational policies due to the absence of core relationship modeling.

5.3 Policy Evaluations

This section evaluates all policy implementations discussed before in the same environments with training configuration 4 cores and 2 internal pools, also in the same unseen environments with 6 cores and 3 internal pools. Evaluation results are provided both quantitatively in Table 5.1, and qualitatively with the figures showing each policy's behavior as plots at every timestep during the evaluation.

5.3.1 Baselines

Looking at the graphs for the baseline policies in Figure 5.1, it is visible that DPH baseline can reach the highest peak reward values among the baselines. However, as shown in Table 5.1, DPH fails to keep the peak reward consistently during the evaluation due to the

Method (Mean)	4 Cores, 2 Internal Pools			6 Cores, 3 Internal Pools		
	Reward	Action	Δ Action	Reward	Action	Δ Action
DPH	0.56958	56.5672	1.20176	0.844295	57.3512	1.20336
FTN	0.638435	64.7906	0.27512	—	—	—
CTN	0.398836	45.7764	0.409528	0.650045	41.1727	0.298038
URMA-D	0.588621	59.3902	1.40102	0.858016	55.2295	0.7085
URMA-D-Prior	0.684002	67.6324	0.29964	0.995273	66.9085	0.258473
URMA-ED	0.690324	66.1539	0.20056	1.00035	65.0368	0.224446
GNN	0.642044	66.7772	0.509107	0.897375	66.6157	0.629637
Transformer	0.659384	62.4287	0.396817	0.929922	59.9369	0.406992
AgentFormer	0.609561	59.3421	0.602082	0.696681	50.5202	1.09187

Table 5.1: The main results of baselines and represented approaches in the methodology. The table includes two environment settings: the training configuration (4 cores and 2 internal pools) and the unseen environment configuration (6 cores and 3 internal pools). Results are represented as groups to separate baselines from other approaches. Bold values indicate the best among each group for each metric, favoring the highest mean reward value, lowest mean core’s action and the lowest mean core’s delta action values.

slow convergence to best values and high mean delta action values, reducing the overall mean reward. The slow convergence also lowers the policy’s mean action value, because the policy starts from a low resource allocation and increases it gradually as visible in Figure 5.1.

According to Table 5.1, FTN achieves higher mean reward compared to DPH, while maintaining more stable resource allocation shown with the lower delta action values. This result is supported with the curves in Figure 5.1, where FTN follows a trajectory close to the peak points of DPH without the oscillations or the slow convergence step in the beginning. The mean action value of FTN is therefore higher than DPH, because the policy instantly utilizes enough resources to maximize the throughput in the whole evaluation. However, as this approach is trained for fixed size of observations and actions, the trained network with the configuration of 4 cores and 2 internal pools cannot be evaluated on 6

cores and 3 internal pools.

Interpreting the core dimension as batch dimension to solve the shortcomings of FTN, CTN achieves the lowest mean reward compared to other two baselines which can interpret the overall pipeline observation according to Table 5.1. This behavior is also illustrated in Figure 5.1, where CTN could not reach the peak reward values of other two baselines. Additionally, it also fails to stabilize the reward during the evaluation of some environments.

Evaluating the resulting policies on the environment configuration with 6 cores and 3 internal pools, results from Table 5.1 indicate a notable mean reward gap for CTN compared to DPH. This behavior could also be confirmed with the observations in Figure 5.2, where DPH could reach higher reward values consistently through multiple environments.

5.3.2 URMA-based Policy

Decoder-only Policy

According to the values in Table 5.1, URMA-D reaches a higher mean reward than CTN, but there is still a notable gap from the highest performer baseline FTN. The reward plots in Figure 5.3 also support the reported values and the performance ordering of policies. Additionally, URMA-D suffers from the stability issues similar to CTN, which is both visible in the reward and action trajectories, and indicated with notably high mean delta action values in Table 5.1. During the evaluation on the 6 cores and 3 internal pools settings, URMA-D could generalize and perform similarly, outperforming the results of DPH from Table 5.1. This result is supported by Figure 5.4, where DPH could reach higher peak values without maintaining them consistently, causing URMA-D to perform better overall.

Utilizing the core description vectors, URMA-D-Prior can not only significantly improve the mean reward values compared to the baseline approaches, but also achieve lower mean delta action value over URMA-D as shown in Table 5.1. Therefore, the resulting action and reward curves are highly stable in Figure 5.3, providing smoother control of the whole pipeline. It also generalizes quite well according to Table 5.1, as URMA-D-Prior could both reach higher mean reward and lower mean delta action values during the evaluation on environments with 6 cores and 3 internal pools. The reward and action curves in Figure 5.4 support this generalization with URMA-D-Prior outperforming the baseline approaches on environments that were not present during the training process.

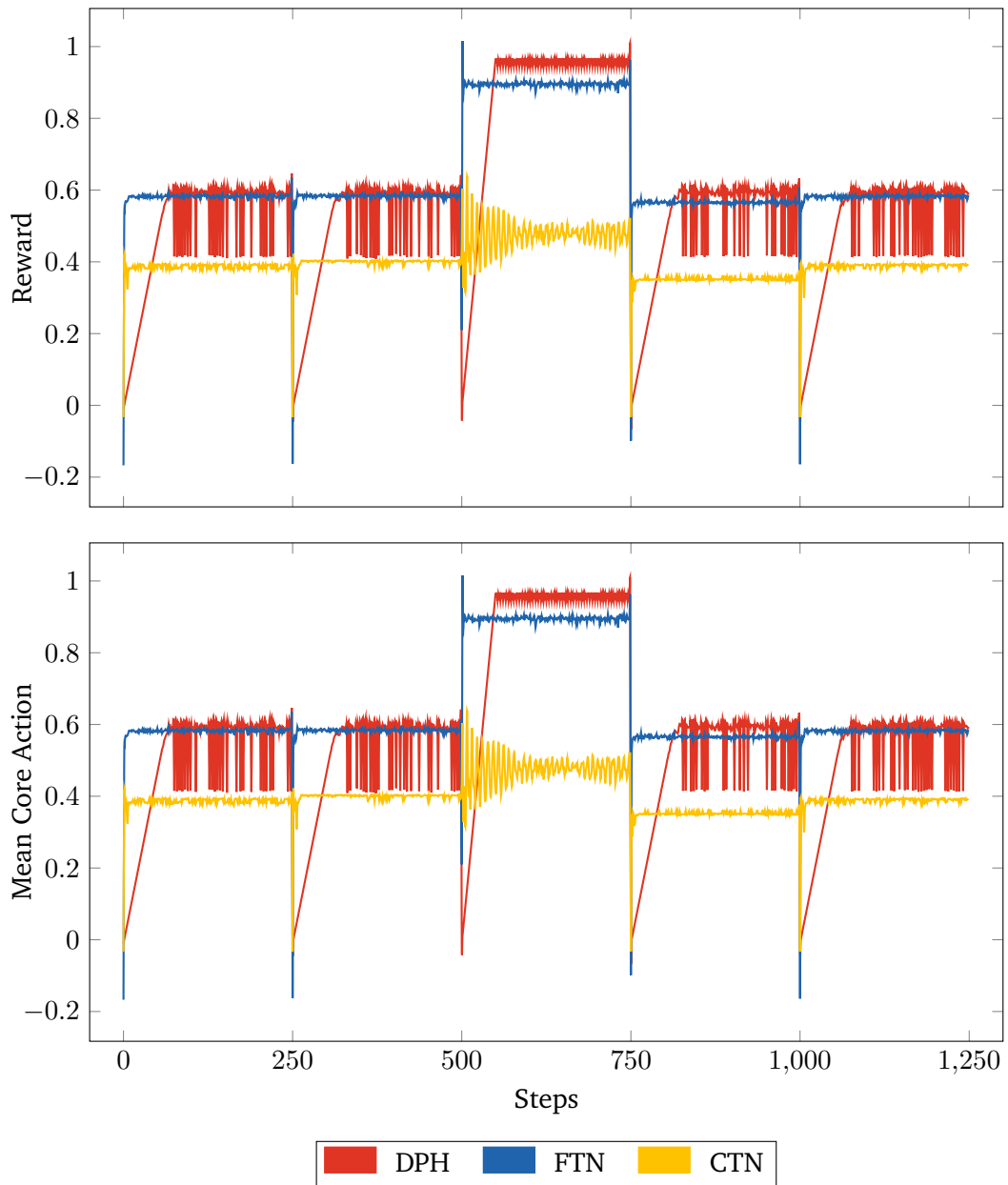


Figure 5.1: Evaluation results of baseline approaches after the training process using the same environment settings during the training (4 cores, 2 internal pools).

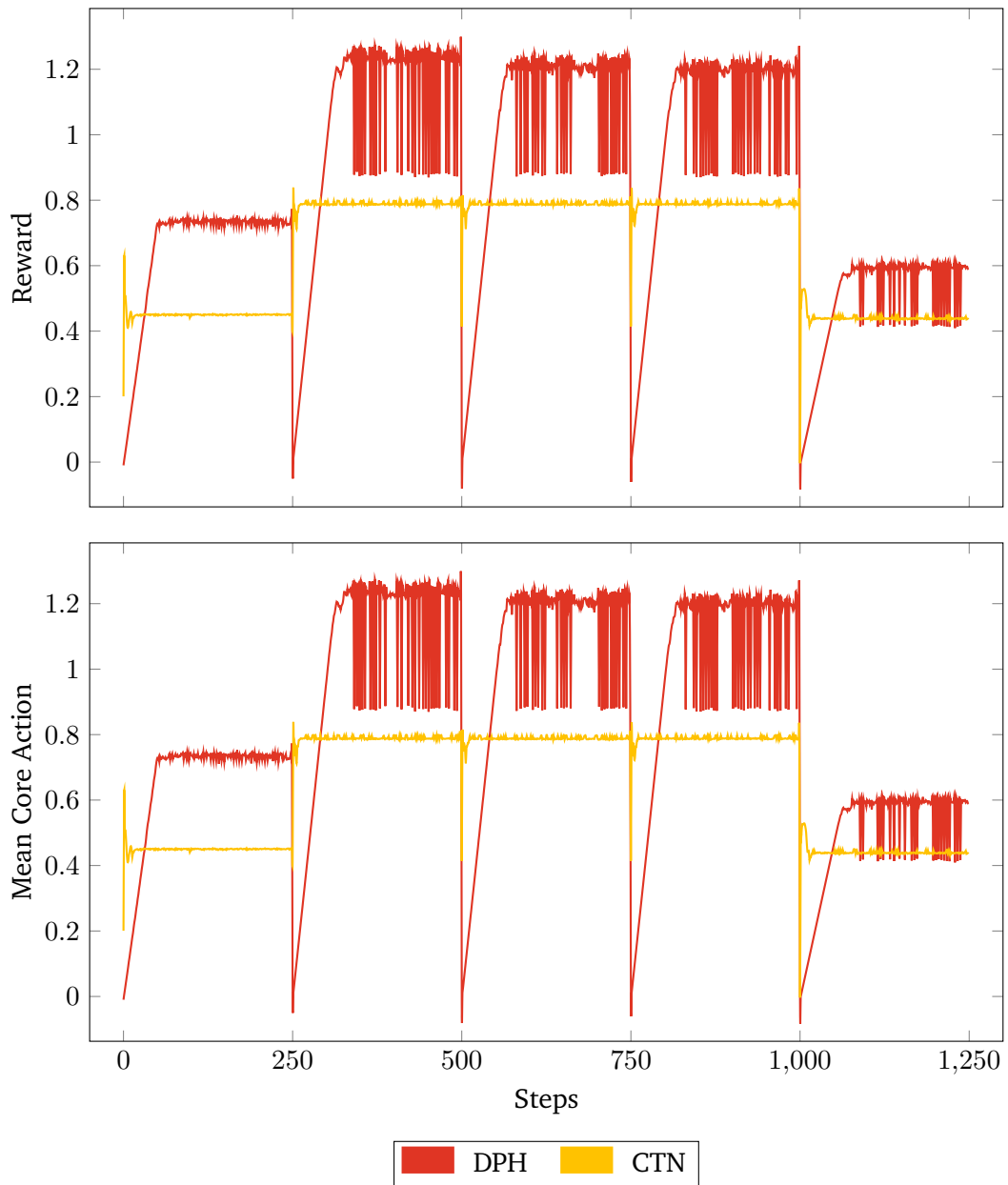


Figure 5.2: Evaluation results of baseline approaches after the training process using unseen environment settings (6 cores, 3 internal pools).

Encoder and Decoder Policy

Adding an extra component to the policy network, URMA-ED could reach a comparable mean reward as URMA-D-Prior according to Table 5.1 in both the training and unseen environment configurations, which is also further illustrated in Figures 5.3 and 5.4. However, the mean core's action and delta action values of URMA-ED are noticeably lower than URMA-D-Prior in both configurations. Therefore, the inclusion of encoder component resulted in more efficient and stable policy that can reach the same reward values.

5.3.3 Graph Neural Network Policy

Considering the values in Table 5.1, GNN policy achieves lower mean reward values than URMA-ED and URMA-D-Prior, although remaining in a comparable range. Figure 5.5 illustrates that GNN keeps the reward value stable while matching the best performing policy most of the time for the environments using the similar configuration as the training. GNN policy also provides medium mean delta action values, indicating a lower stability compared to URMA-ED and URMA-D-Prior, which is visible with sudden changes on the trajectory in Figure 5.5.

Observing GNN policy during the evaluation on the 6 cores and 3 internal pools settings, the values in Table 5.1 confirm that the policy could transform its knowledge for different environments with a comparable mean reward value. However, the reward plots in Figure 5.6 illustrate that the situation of lower performance is more frequent during the evaluation. Additionally, the change of environment has increased the mean delta action values of GNN policy, resulting in degraded stability visible in the reward curve. In contrast to this instability, the mean core's action is quite stable similar to the original environment configuration. Overall, the lower mean core's action values on both evaluation configurations indicate a struggle to understand the uncertainty of the core dynamics.

5.3.4 Transformer-based Policy

Transformer-based policy favors the efficient control of the pipeline, while maintaining close to the best throughput values. According to the results in Table 5.1, Transformer-based policy could reach a mean reward value close to the value of URMA-ED in the training environment configuration, which is also visualized with the reward curve in Figure 5.7. Compared to URMA-ED, it utilizes fewer threads on average, which allows

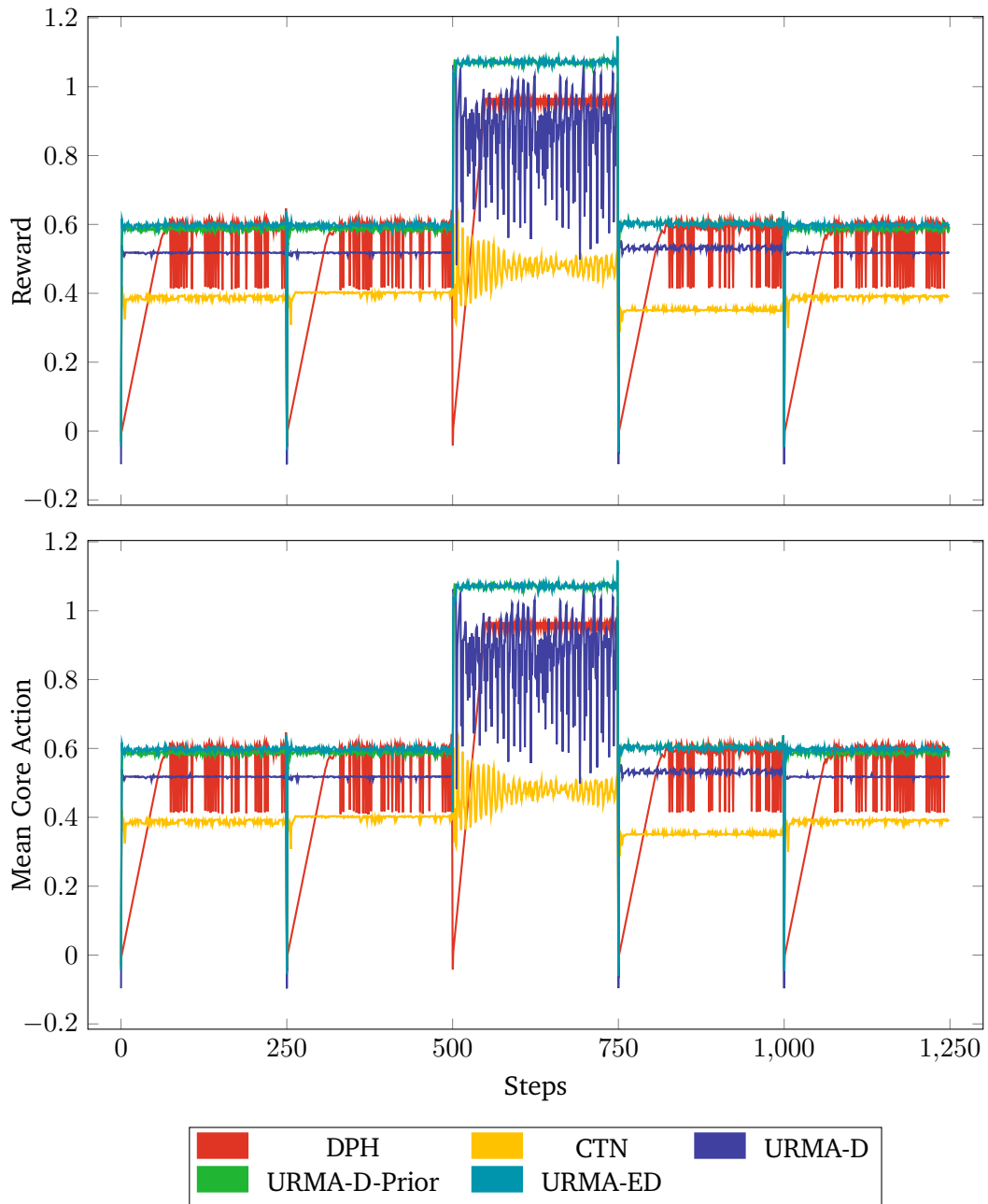


Figure 5.3: Evaluation results of URMA-based policies after the training process using the same environment settings during the training (4 cores, 2 internal pools).

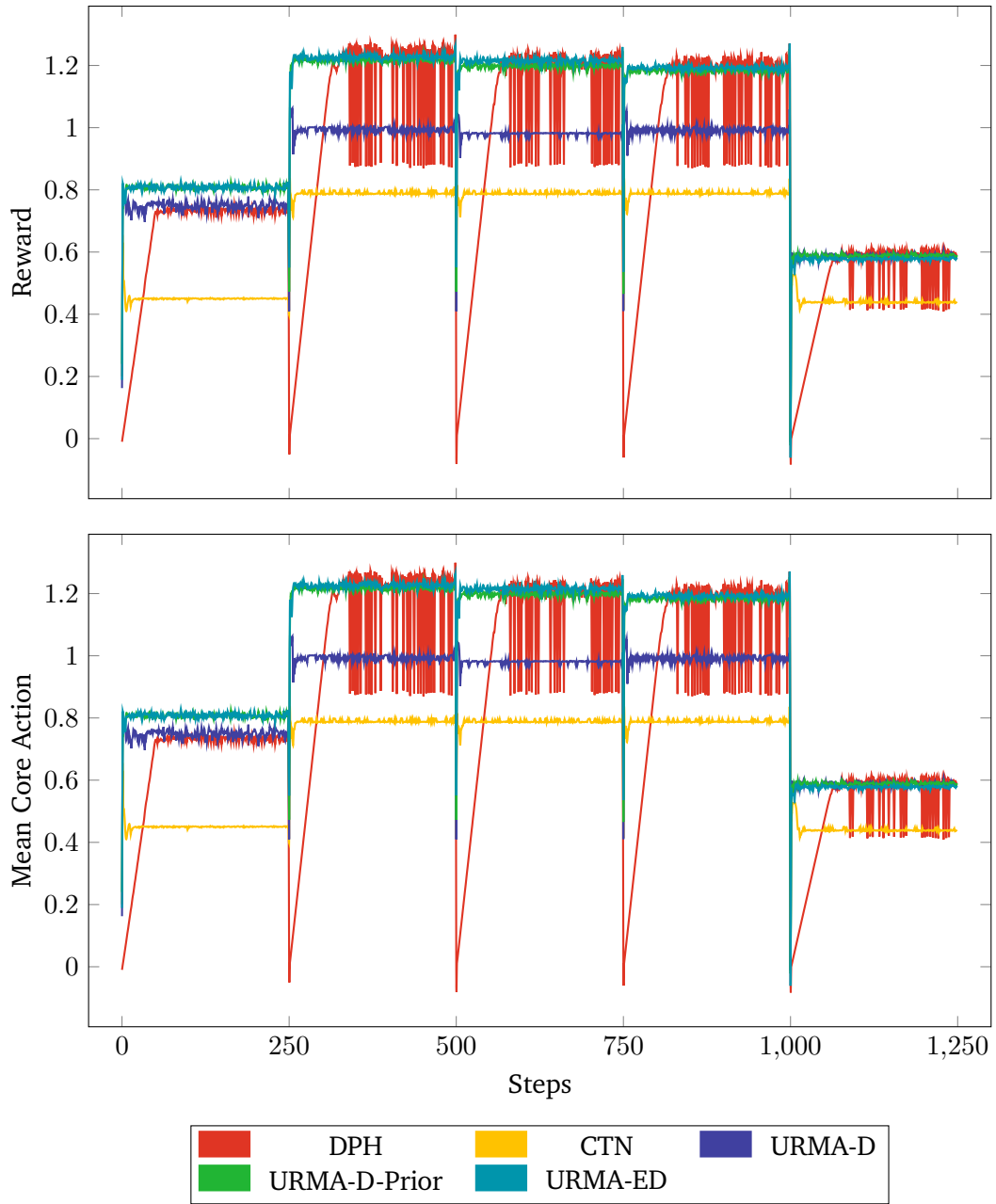


Figure 5.4: Evaluation results of URMA-based policies after the training process using unseen environment settings (6 cores, 3 internal pools).

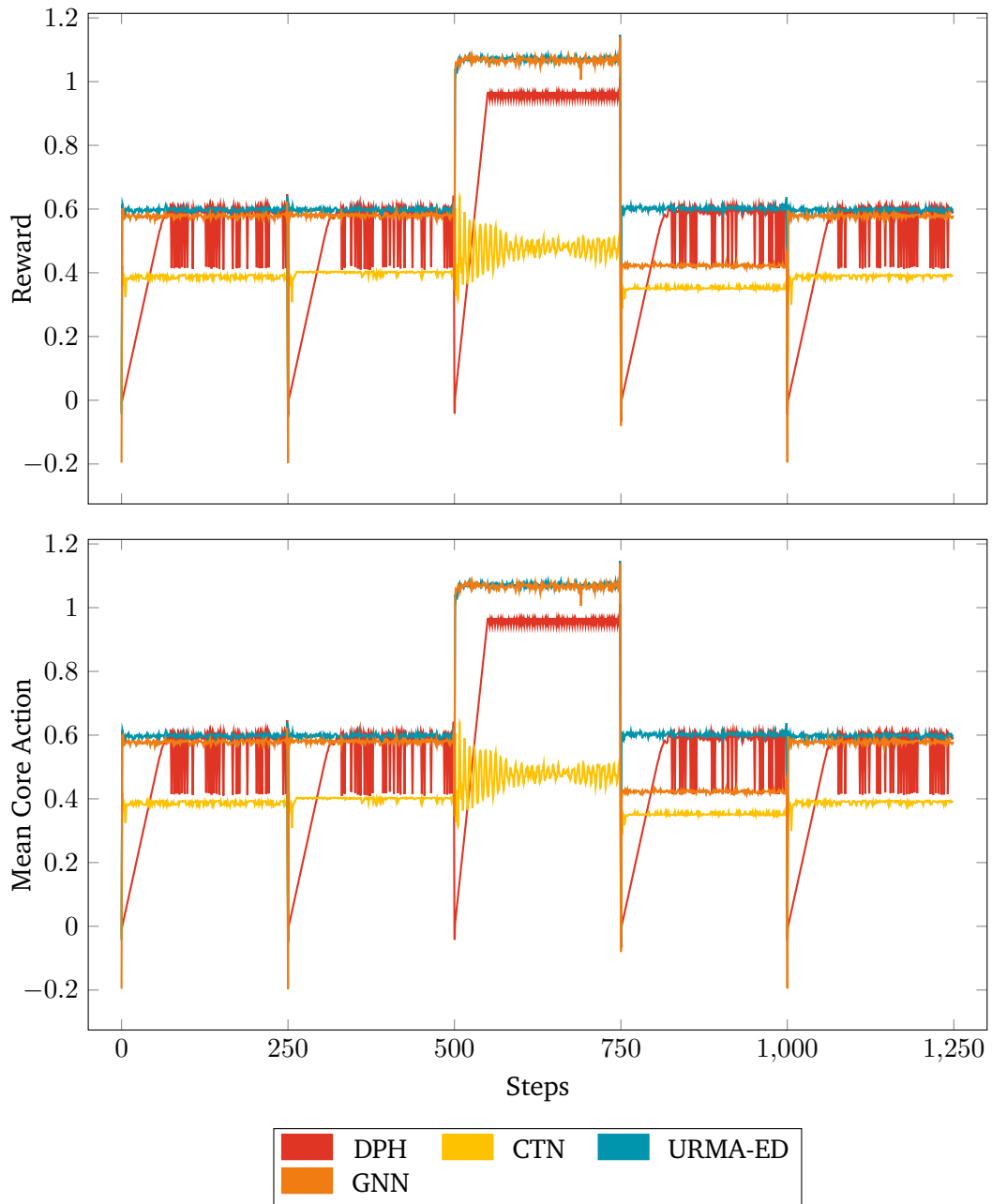


Figure 5.5: Evaluation results of GNN policy after the training process using the same environment settings during the training (4 cores, 2 internal pools).

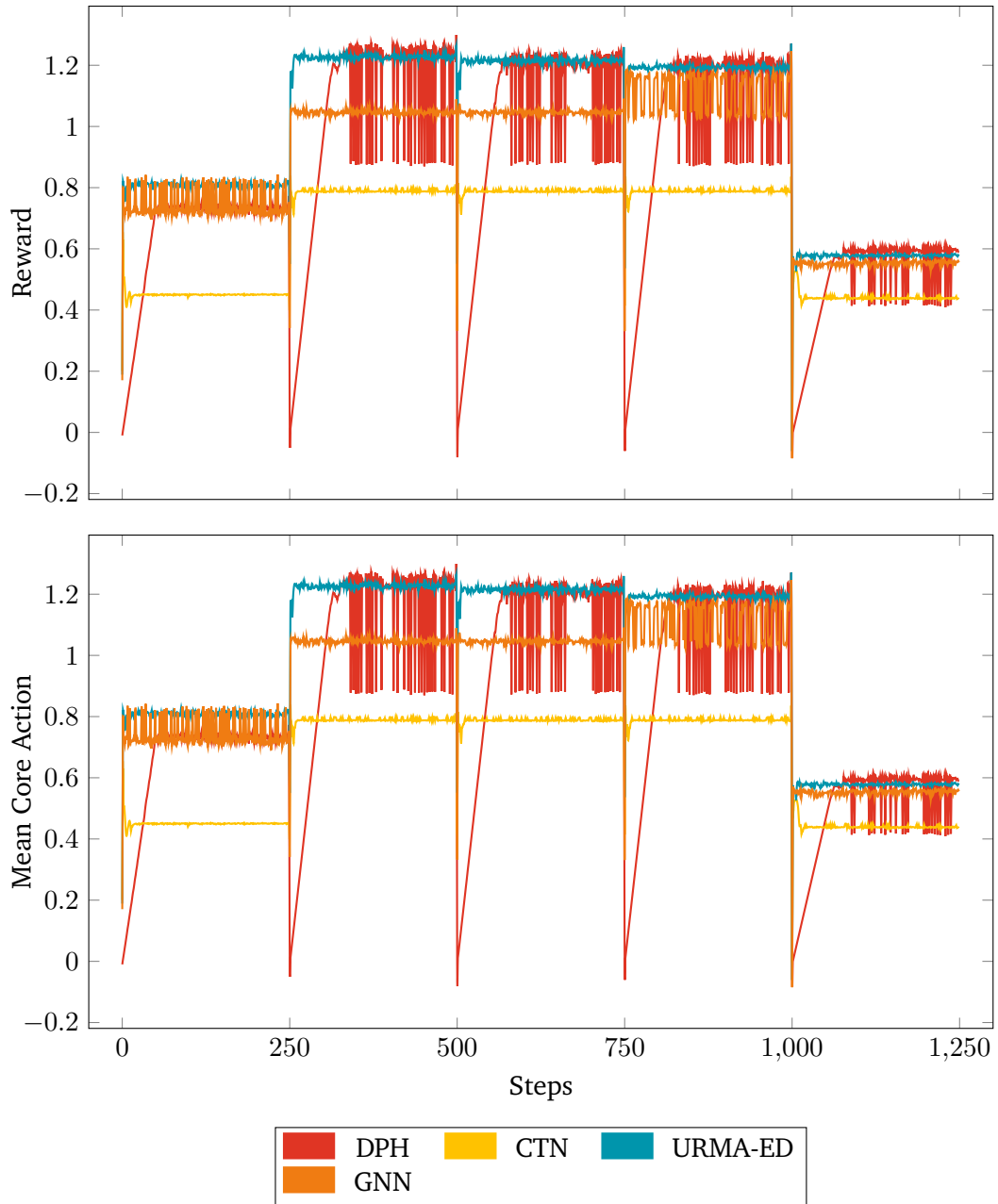


Figure 5.6: Evaluation results of GNN policy after the training process using unseen environment settings (6 cores, 3 internal pools).

more efficient processing as a whole pipeline. The mean delta action is slightly higher than URMA-ED and lower than GNN, which is visible with the reward and action curves following relatively stable trajectories during the evaluation, allowing for the smooth control of the pipeline.

During the evaluation on unseen configuration, the values in Table 5.1 show slightly degraded performance of Transformer-based policy compared to URMA-ED, but with a higher mean reward than GNN policy as visualized in Figure 5.8. The performance gap could be as a result of the efficiency oriented learning, because the mean core's action values of Transformer-based policy are significantly lower compared to URMA-ED on almost all the evaluated environments. Besides, Transformer-based policy maintains a moderate delta action, which is reflected in the stability of reward and action curves.

5.3.5 AgentFormer-based Policy

As indicated in the evaluation results in Table 5.1, AgentFormer-based policy achieves a lower mean reward compared to Transformer-based policy on the training environment settings. It favors utilization of even fewer threads and reaches slightly lower mean reward without any prior information about the pipeline topology, which is also visible in Figure 5.9. With the relatively higher delta action value, AgentFormer-based policy is visibly more unstable compared to the other approaches in some environments, but not resulting in oscillations like DPH or GNN policy.

Observing the evaluation results of the unseen environment configuration, Table 5.1 shows the poor generalization of AgentFormer-based policy compared to Transformer-based and URMA-ED due to the substantially lower mean reward value. Figure 5.10 also illustrates that the reward reached in most of the environments still has a notable gap from the best performing policies. For the stability aspect, both the reward and mean core's action values look largely similar to the evaluation results with original pipeline configuration, which has a moderate instability. The mean core's action values are also notably lower compared to the policy architectures that were introduced before.

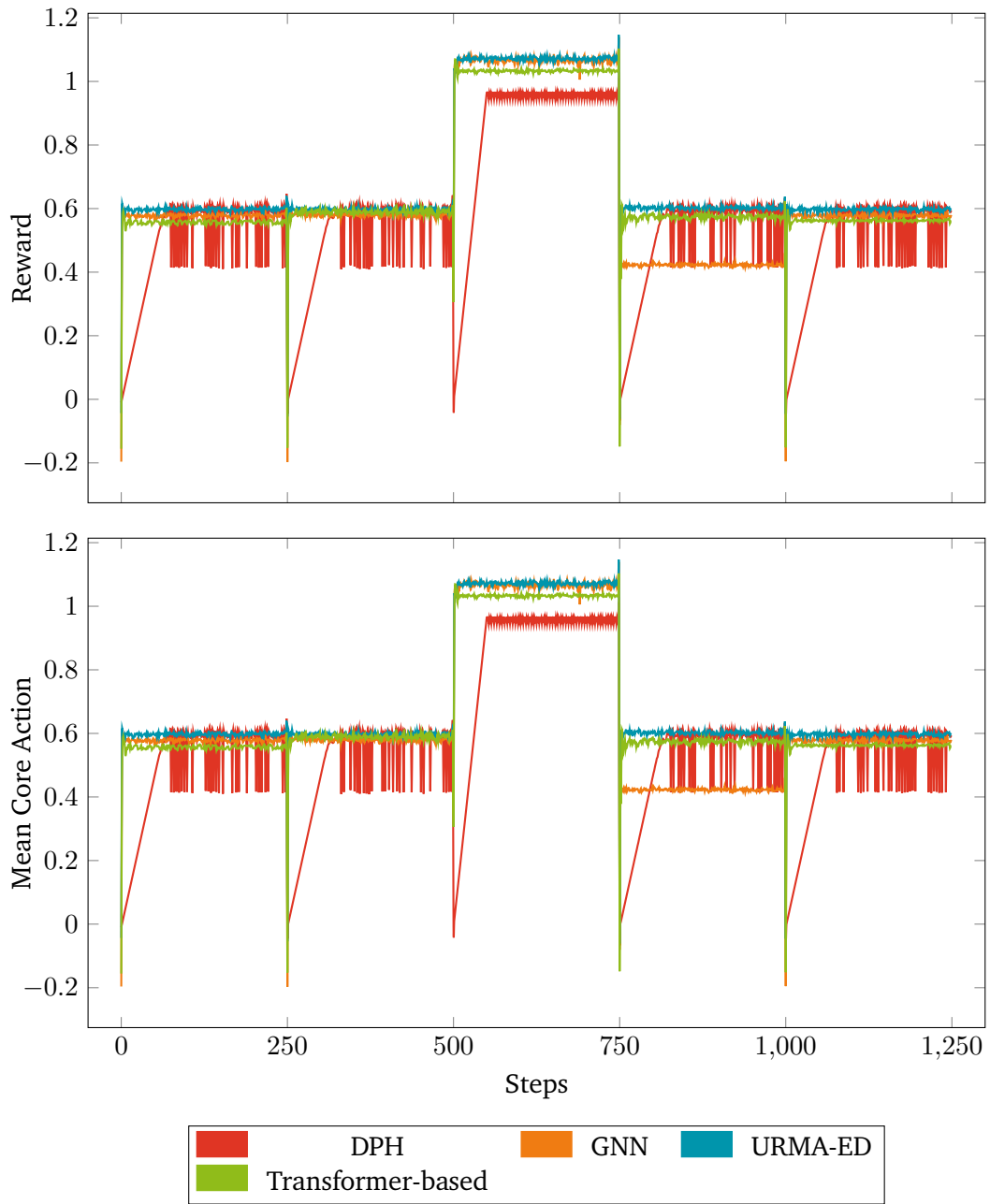


Figure 5.7: Evaluation results of Transformer-based policy after the training process using the same environment settings during the training (4 cores, 2 internal pools).

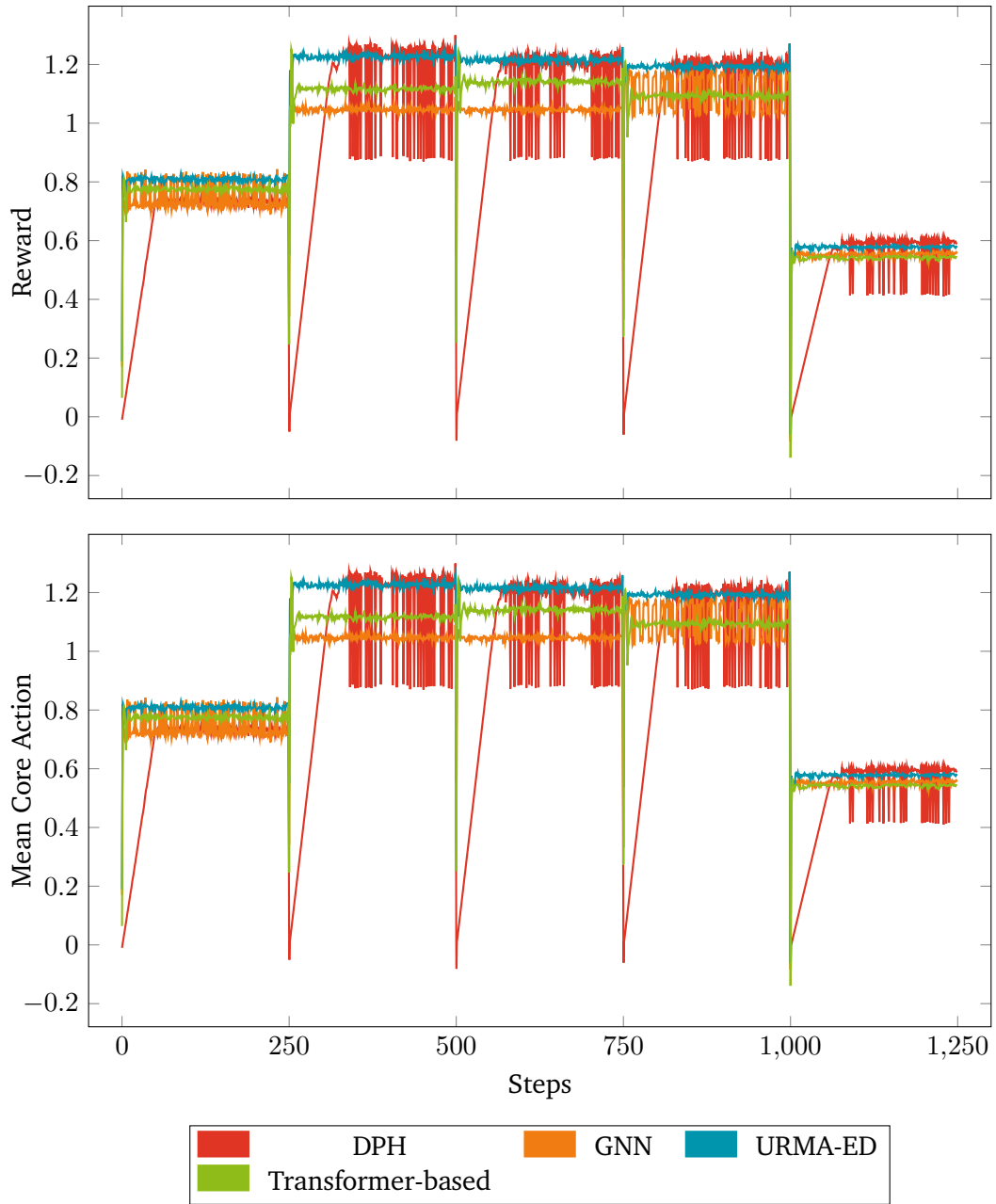


Figure 5.8: Evaluation results of Transformer-based policy after the training process using unseen environment settings (6 cores, 3 internal pools).

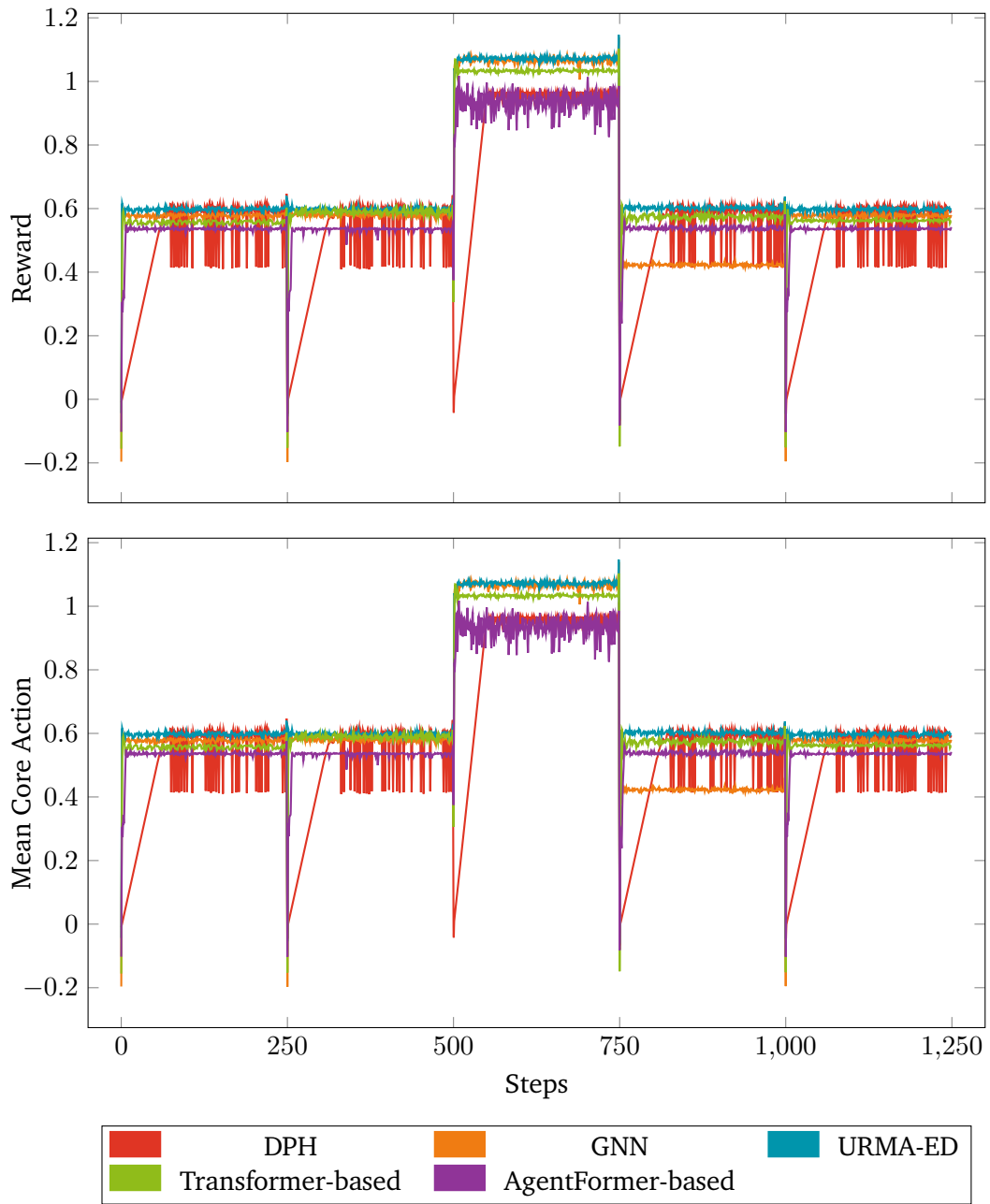


Figure 5.9: Evaluation results of AgentFormer-based policy after the training process using the same environment settings during the training (4 cores, 2 internal pools).

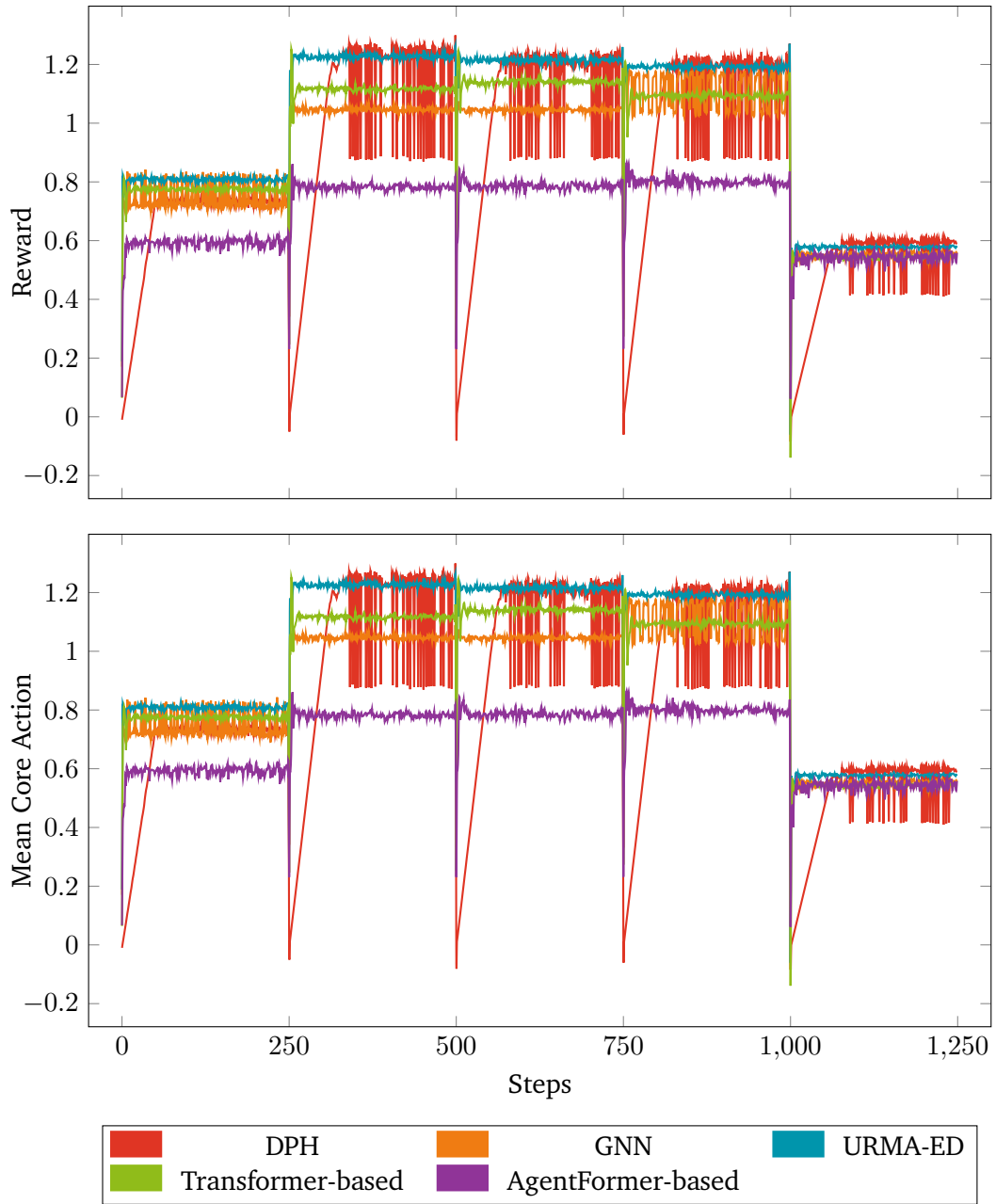


Figure 5.10: Evaluation results of AgentFormer-based policy after the training process using unseen environment settings (6 cores, 3 internal pools).

6 Discussion

Following the experiments, this chapter interprets the experimental results reported in previous chapter with respect to the evaluation metrics defined in the methodology. At first, the task performance is analyzed to understand the characteristics of each policy in terms of the network throughput and the resource efficiency in the environments following the training configuration. Then, the policies are examined in the unseen environment configurations to understand the generalization capabilities of the trained policies. As the last metric, the stability of each policy is reviewed to ensure smooth resource allocation and stable trajectory for the received reward.

6.1 Task Performance

Task performance of each policy is assessed with the cumulative reward, combining the pipeline throughput, resource utilization cost and allocation stability as mentioned in Section 4.3. This reward is mostly composed of the delta committed documents of the pipeline with a high contribution, resembling a similar graph as the actual delta committed documents graph. Using the distinct findings for every approach from the results presented in Chapter 5, the results will be interpreted in following paragraphs.

Although being a simple strategy to control resource allocation by increasing and decreasing thread allocation according to the committed document count, DPH achieves the best peak reward values among all approaches. However, this performance with peak reward values doesn't translate into better cumulative reward. As the controlling logic resembles an online numerical optimization that controls the resource utilization until the highest number of committed documents reached, the resource allocation always changes in each timestep. This behavior results in a constant penalty due to the stability term of reward, and therefore a decreased cumulative reward in some challenging environments.

FTN reaches better reward values than DPH, while maintaining a stable resource allocation. With the usage of a temporal processing, the network can understand the observation history and produce smooth outputs without the oscillations. Also, the similar amount of peak reward value as DPH can be reached with less resource utilization, indicating a more efficient policy compared to DPH.

Due to the independent processing of each core as a batch dimension, CTN struggles to capture the relationships between cores, which are important to optimally allocate the threads for each core in the complete ETL pipeline. Especially, inability to model dependencies between cores might affect the scenarios where multiple cores share the same internal pool. Thus, the cumulative reward in different environment evaluations fails to reach the same level as DPH and FTN baselines.

URMA-D can reach higher cumulative rewards compared to CTN by using fixed-size pipeline observation as the latent observation representation of the whole pipeline. Therefore, the actions get generated with the consideration of overall pipeline state, which is missing in CTN. However, this policy has a notable gap in cumulative reward compared to the best performing baselines like FTN, also it experiences instabilities similar to CTN.

Using the core description vectors as prior information, URMA-D-Prior outperforms the mentioned baselines in the cumulative reward. The description vectors include topological information of each core, which gets used by the decoder to differentiate cores that are close to source or target, or in the middle of the pipeline. With this information, the policy can make more specific decisions for each core, which allows higher throughput and smoother thread allocation.

Additionally utilizing encoder on top of URMA-D-Prior, URMA-ED observes the same reward level during the evaluation of different environments. However, the efficiency of the policy improves significantly by achieving the same performance while utilizing fewer resources on average. Therefore, the encoder with similar logic as attention can provide a more informative latent representation of the pipeline by aggregating each core's observation. Additionally, the core-based attention logits are also used in the decoder, making the most important components of the observation more apparent during the action generation.

GNN policy can reach cumulative reward values that are comparable to URMA-based architectures using the training environment configuration, while mostly having a stable reward signal through different environments. Message passing allows GNN to aggregate the information of neighboring cores, capturing dependencies between the cores without an extra core description vector definition.

Learning an efficient control policy, Transformer-based policy performs close to the best performing policies, while allocating fewer threads in general for each core. Attention mechanism processes both the observation history and the core relationships to capture both dimensions of the problem. Additionally, utilization of custom positional encoding derived from the core description vectors and learning a bias value for the directly connected cores in attention scores allows the network to build a topology- and history-aware representation.

AgentFormer-based policy provides a similar control mechanism as Transformer-based policy by having a single attention process that both handles the observation history and core relationships at the same time. As a result, it performs comparable but somewhat lower level than Transformer-based policy, while utilizing even smaller amount of threads. In contrast to Transformer-based policy, this policy does not receive any prior information about the pipeline topology, so the learned attention weights can handle the dependency information.

6.2 Generalization

This section discusses the evaluation results of the policies in unseen environments with configuration of 6 cores and 3 internal pools, which were not present during the training. The reached cumulative reward provides a direct measure for each policy’s generalization ability from the learned knowledge to new pipeline topologies.

DPH can trivially be used with any pipeline configuration, as its logic does not depend on the number of cores and allocates the same resources for every core. In the tested environments, the heuristics could perform well enough in most of the cases, but is still suffering from the same problems observed in the original environment settings. However, this resource allocation strategy is not always the best, as more complex pipeline configurations require more advanced resource allocations. Also, the step size becomes more and more sensitive to tune for increased number of cores.

FTN is trained on 4 cores and 2 internal pools similar to the other approaches, but it assumes a fixed observation and action size. Therefore, this policy cannot be evaluated in this configuration at all. Although the architecture was able to perform relatively well in the training configuration, it is not suitable to use in the environments with variable topologies due to the lack of generalization.

Handling the core dimension as batch dimension, CTN can be evaluated in different pipeline configurations, providing a baseline for what a single learned policy can achieve with variable number of cores. In the evaluation of 6 cores and 3 internal pools configuration, CTN still performs worse compared to DPH, picturing the limited generalization ability due to the lack of relational information.

URMA-D can generalize beyond CTN with a higher mean reward compared to both CTN and DPH baselines. However, it still leaves a noticeable gap from the peak reward value of DPH on unseen configuration. Using the pipeline observations as the latent observation provides a transferable context for the network, but does not allow decoder to adjust the per-core decisions accordingly for the changed pipeline structure.

Adding the prior information, URMA-D-Prior generalizes significantly better, performing very similar or even better than the peaks of DPH. The prior information encoded by the core description vectors provides the decoder with topological information that is invariant to the number of cores, resulting in more informed resource allocations based on the structural position of cores inside the pipeline.

With inclusion of encoder, URMA-ED can generalize comparably to URMA-D-Prior, achieving similar rewards on the unseen environment configuration. Because the encoder applies an attention-based aggregation to calculate a fixed-size latent representation of pipeline state, it enables the network to be more aware of the topological details. Therefore, for even larger unseen pipeline configurations, URMA-ED might generalize better compared to URMA-D-Prior.

Using the message passing mechanism, GNN policy operates on the graph structure directly to allow generalization to unseen topologies, without depending on fixed number of cores. During the evaluation on this unseen configuration, it could successfully maintain comparable mean core’s action values to the training configuration, displaying the transfer of the resource allocation strategy. However, the number of environments where GNN performs worse increases and reward stability degrades. From these results, it can be concluded that even though GNN can accommodate new topologies, the stochastic dynamics of additional cores might introduce uncertainty that the policy is not capable to handle.

Transformer-based policy can perform quite well on unseen configuration, with slightly less cumulative reward compared to URMA-ED. Using the core description vectors and learned attention bias from the structural information can guide attention mechanism for the new topologies. As the training environment evaluations already favor the most

efficient resource allocation, it is possible that the policy aggressively assigns action values just around the limit, leaving slightly larger performance gap on unseen configurations.

Without any prior information provided, AgentFormer-based policy performs significantly poor generalization compared to Transformer and URMA-ED. The cumulative reward is substantially lower compared to the best performing policies in most of the evaluated environments. This limitation can be caused by several factors, like the policy not being trained with examples from different pipeline configurations than the setup with 4 cores, so the agent-aware attention was not able to handle varying number of agents, or due to not having any structural prior information like URMA-ED and Transformer-based policies.

6.3 Stability

The thread allocation decisions should ideally remain stable for a fixed environment, as frequent changes can cause overhead or unpredictability during the operations. Resource allocation stability is therefore captured in the reward formulation in Section 4.3, and assessed both through the reward signal, and by examining the action allocation trajectories.

The stability of DPH policy heavily depends on the step size value. As this policy constantly modifies the current resource allocation even with lower value of step size, the trajectories are never stable even while staying close to the best values. When the current action values are not the best, the policy needs to adapt the action value quickly, which is possible with higher step size. However, when the policy is close to the best values, it should preferably perform zero changes, meaning a lower step size. Therefore, the choice of step size introduces a trade-off due to the online numerical optimization behavior.

Observing the evaluation results on environments following the training configuration, CTN often produces an oscillating reward and mean action trajectory which starts with high intensity, reducing as the timesteps increase in the same environment. Such challenging environments also affect URMA-D, as the evaluation of this policy resulted in highly oscillating trajectory. Transformer-based policy and AgentFormer-based policy show high peaks and dips in initial timesteps of environment, which becomes more evident with evaluation results on unseen environment configuration. Additionally, AgentFormer-based policy shows slightly less stable trajectory compared to the other remaining policies, which were equally stable.

Looking at the environment evaluations following the unseen configurations, GNN policy becomes visibly less stable in some environments together with AgentFormer-based policy which follows a similar noisy trajectory as the evaluations on environments on training configuration. Apart from these policies, the other policies remain equally stable with their generalized form. It is possible that the instabilities seen on the environments with training configuration were not present in this environment configuration, because the evaluated environments might not be challenging enough as the environments of previous evaluations.

7 Outlook

Referring to the main findings and their implications, this chapter concludes the thesis by providing a brief summary of the problem definition, environmental assumptions, experimental setup and the policy architectures offering the best balance with respect to the evaluation criteria. Last sections state the limitations of current study in detail, followed by the future work directions.

7.1 Conclusion

This thesis addresses the open-world problem where observation and action sizes of the environment are not constant. As an example for such open-world environment, the proposed architectures were demonstrated on ETL pipelines, which might have varying number of cores and different topological structures. The main challenge is to explore policy architectures to control these pipelines without requiring to retrain it, by capturing both core-specific information and pipeline context while remaining consistent for the changes of pipeline topology and number of cores. Using the ETL environment formulation by [2], the problem has been formalized with the domain randomization strategy of topology reconstruction and sampled core dynamics at each environment reset.

Several policy architectures were presented and trained using PPO under identical conditions on the same environments with 4 cores and 2 internal pools configuration. After the training, they were evaluated on both seen and unseen conditions with 6 cores and 3 internal pools in three metrics: task performance, generalization, resource allocation stability.

In summary, the usage of decoder and the prior information inside the policy architecture allows reaching significantly higher reward values. URMA-ED provides the maximum performance while prioritizing generalization and stability, whereas URMA-D-Prior provides

the best balance with a relatively simple implementation and quite similar reward values as URMA-ED. In environments where the resource efficiency is important, Transformer-based policy could reach similar performance as URMA-ED while allocating noticeably fewer threads in general. URMA-ED introduces additional complexity for the implementation, but it provides improved efficiency while performing the same as URMA-D-Prior.

7.2 Limitations

As a major limitation, the training was done exclusively on ETL pipelines with 4 cores and 2 internal pools, utilizing various topologies with the stochastic core dynamics at each environment reset to ensure domain randomization. Although the results look promising, introduction of different pipeline configurations might help to generalize even better for most of the architectures, especially the policies without exploiting the structural prior information like AgentFormer-based policy.

Furthermore, the modeled ETL system considers a directed acyclic graph where each core connects only one source pool to one target pool. Real-world ETL systems might have more complex topologies like having parallel connections between two pools, and possibly include cycles. Additionally, while the information about the core adjacency matrix is usually available for ETL pipelines, some systems require this information to be estimated. These special cases are not captured with the current environment model, and it is unclear how the introduced architectures would perform in such environments.

The stochasticity in environment dynamics is integrated with gamma-distributed response times, which is controlled with hyperparameters to limit ranges of maximum throughput, saturation rate and variance. Although the operational noise can be captured with this formulation, ETL environments can also suffer from system latencies, inconsistent data arrival rates, correlated node failures sharing the same infrastructure. Also, the stochasticity of the cores is sampled once through the environment reset and does not vary over time. Therefore, the modeling of these factors of additional uncertainty is required for more advanced simulation of the ETL processes.

7.3 Future Work

The first possible future work is to deploy the learned policies on actual ETL systems operating regularly in production environments. This process requires providing the RL agent an interface to actually control the existing pipelines, and creating stable monitoring solutions to supply policy with these assumptions.

Additionally, the current experiments use small pipelines that have either 4 cores or 6 cores, whereas the production ETL systems can consist of larger number of processing stages. The increasing number of cores might present computational challenge for attention-based architectures, as both computation and space complexities increase quadratically. Message passing mechanism of GNN should be sufficient to capture longer dependencies in deep graphs.

Lastly, the contributions of thesis might also be applied in other dynamic graph-based problems, as most of the architectures utilize structural priors and topology-aware networks to handle variable size of state and action space. Because learning a single policy using the topological information can generalize structurally well even for the unseen environments, the resulting policy could provide a robust solution for domains where the systems continuously evolve.

References

- [1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series, Cambridge, Massachusetts: The MIT Press, second ed., 2018.
- [2] L. Maier, L. Schulze, R. Lilow, L. Hahn, N. Krasowski, A. Barth, S. Gaebel, F. Gran, O. Hanau, G. Wagner, F. Borgmann, O. Arenz, and J. Peters, “Mathematical Foundations of Modeling ETL Process Chains,” No. arXiv:2603.29877, arXiv, Feb. 2026.
- [3] W. Yu, J. Tan, C. K. Liu, and G. Turk, “Preparing for the Unknown: Learning a Universal Policy with Online System Identification,” in *Robotics: Science and Systems Conference*, arXiv, Feb. 2017.
- [4] T. Miki, J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, “Learning robust perceptive locomotion for quadrupedal robots in the wild,” *Science Robotics*, vol. 7, p. eabk2822, Jan. 2022.
- [5] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, “Domain randomization for transferring deep neural networks from simulation to the real world,” in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (Vancouver, BC), pp. 23–30, IEEE, Sept. 2017.
- [6] M. Wen, J. G. Kuba, R. Lin, W. Zhang, Y. Wen, J. Wang, and Y. Yang, “Multi-Agent Reinforcement Learning is a Sequence Modeling Problem,” in *Neural Information Processing Systems*, (New Orleans, Louisiana, USA), pp. 16509–16521, arXiv, Oct. 2022.
- [7] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, “Policy Distillation,” in *International Conference on Learning Representations*, arXiv, Oct. 2015.

-
-
- [8] C. D'Eramo, D. Tateo, A. Bonarini, M. Restelli, and J. Peters, "Sharing Knowledge in Multi-Task Deep Reinforcement Learning," in *International Conference on Learning Representations*, arXiv, Apr. 2020.
- [9] D. Rother, J. Pajarinen, J. Peters, and T. H. Weisswange, "Open-ended coordination for multi-agent systems using modular open policies," *Autonomous Agents and Multi-Agent Systems*, vol. 39, p. 40, Oct. 2025.
- [10] H. Fawaz, D. Zeghlache, T. A. Pham, J. Leguay, and P. Medagliani, "Deep Reinforcement Learning for Smart Queue Management," *Electronic Communications of the EASST*, vol. 80, Sept. 2021.
- [11] J. Wang, J. Cao, S. Wang, Z. Yao, and W. Li, "IRDA: Incremental Reinforcement Learning for Dynamic Resource Allocation," *IEEE Transactions on Big Data*, vol. 8, pp. 770–783, Apr. 2020.
- [12] W. H. Inmon, *Building the Data Warehouse*. Wiley Technology Publishing Timely, Practical, Reliable, Indianapolis, Indiana: Wiley, fourth edition ed., 2005.
- [13] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," No. arXiv:1707.06347, arXiv, Aug. 2017.
- [14] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," in *International Conference on Machine Learning*, no. arXiv:1502.05477, PMLR, Feb. 2015.
- [15] J. L. Elman, "Finding Structure in Time," *Cognitive Science*, vol. 14, pp. 179–211, June 1990.
- [16] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, pp. 157–166, Mar. 1994.
- [17] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, pp. 1735–1780, Nov. 1997.
- [18] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," in *Neural Information Processing Systems*, arXiv, 2017.
- [19] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The Graph Neural Network Model," *IEEE Transactions on Neural Networks*, vol. 20, pp. 61–80, Jan. 2009.

-
-
- [20] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural Message Passing for Quantum Chemistry,” in *Proceedings of the 34th International Conference on Machine Learning* (D. Precup and Y. W. Teh, eds.), vol. 70 of *Proceedings of Machine Learning Research*, pp. 1263–1272, PMLR, Aug. 2017.
- [21] T. N. Kipf and M. Welling, “Semi-Supervised Classification with Graph Convolutional Networks,” in *International Conference on Learning Representations*, no. arXiv:1609.02907, arXiv, Sept. 2016.
- [22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph Attention Networks,” in *International Conference on Learning Representations*, no. arXiv:1710.10903, arXiv, Feb. 2018.
- [23] N. Bohlinger, G. Czechmanowski, M. Krupka, P. Kicki, K. Walas, J. Peters, and D. Tateo, “One Policy to Run Them All: An End-to-end Learning Approach to Multi-Embodiment Locomotion,” in *Conference on Robot Learning*, 2024.
- [24] Y. Yuan, X. Weng, Y. Ou, and K. Kitani, “AgentFormer: Agent-Aware Transformers for Socio-Temporal Multi-Agent Forecasting,” in *International Conference on Computer Vision*, (Montreal, QC, Canada), pp. 9793–9803, IEEE, Oct. 2021.
- [25] C. D. Freeman, E. Frey, A. Raichuk, S. Girgin, I. Mordatch, and O. Bachem, “Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation,” 2021.