University of Stuttgart
Institute of Industrial Automation and
Software Engineering

Adversarial Multi-Agent Transfer Learning

Forschungsarbeit 3613

Submitted at the University of Stuttgart by
Amal Chulliyat Jose

M.Sc. Electrical Engineering

Examiner: Jun.-Prof. Dr.-Ing. Andrey Morozov

Supervisor: Joachim Grimstad, M.Sc.Ing

16th April 2024

Abstract

Fault Tree Analysis (FTA) is an important technique for assessing and mitigating risks inherent in complex systems. It provides a graphical representation of the logical interconnections among system components and the potential events culminating in system failure. While static fault trees delineate fixed relationships, Dynamic Fault Trees (DFTs) can accommodate dynamic systems where conditions change over time, such as repairs, maintenance, or system reconfiguration. DFTs for complex systems can be modeled as a game environment in which two adversarial agents play. This study explores the feasibility of leveraging Transfer Learning (TL) to train neural networks representing adversarial agents within a simulated gaming environment, to craft fault-tolerant systems capable of adaptations in dynamic settings.

The proposed methodology involves Multi-Agent Reinforcement Learning (MARL), where adversarial agents, labeled "red" and "blue," collaborate within a gaming environment to induce or avert system failure, respectively. Multiple games are played on different versions of the environment, reflecting changes introduced by DFT adjustments. However, the extensive interactions required between the agents and the environment pose challenges for complex large-scale systems. To mitigate this challenge, TL is introduced to enhance agent learning performance by transferring knowledge acquired from previous game environment versions.

Q-learning is a well-known Reinforcement Learning (RL) algorithm in which the Q-value function is estimated using the Bellman equation iteratively until it converges to the optimal value. In Deep Q-Networks (DQN), Convolutional Neural Networks (CNNs) are employed to estimate the Q-value. The initial layers of the CNNs capture low-level data from the current game, enabling generalization for subsequent game versions. Through TL, the parameters of these initial layers are transferred and fine-tuned, facilitating knowledge transfer and improving efficiency across environments.

Proximal Policy Optimization (PPO) stands as another prominent algorithm in the realm of RL, renowned for its stability and effectiveness in optimizing policy functions. Unlike Q-learning, which focuses on estimating Q-values, PPO directly learns the policy by using a clipped surrogate objective function, which constrains policy updates to avoid making significant modifications that could upset the learning process. Similar to DQN, CNNs are employed here also to approximate the policy function.

This study contributes to the advancement of fault-tolerant system design by demonstrating the potential of TL in enhancing agent learning and adaptation within dynamic environments. The findings demonstrate that agents learn more rapidly and effectively with fewer iterations when TL is employed.

Preface

This research work explores the application of TL in enhancing fault-tolerant system design within dynamic environments. As technology continues to advance, the complexity of systems grows, necessitating innovative approaches to risk assessment and mitigation. FTA has long been a cornerstone in this domain, providing a structured methodology for understanding system failures. However, the evolving nature of systems demands adaptive solutions capable of learning and evolving.

The journey of this research has been both challenging and rewarding, filled with moments of insight and collaboration. First and foremost, I thank God almighty for this successful completion. Secondly, I thank Jun.-Prof. Dr.-Ing. Andrey Morozov, for giving me this opportunity. I also extend my sincere gratitude to my supervisor, Joachim Grimstad, M.Sc.Ing, whose guidance has been invaluable throughout this endeavor. His support and encouragement have shaped the trajectory of this research and enriched its outcomes.

Furthermore, I wish to acknowledge the contributions of my colleagues and peers who have provided valuable insights and encouragement along the way. Their perspectives have broadened my understanding and inspired new avenues of inquiry.

Lastly, I am grateful to my family and friends for their unwavering support and encouragement. Their belief in me has been a source of strength and motivation, propelling me forward even in moments of doubt.

This report represents the culmination of the past few months of dedication. I hope that the insights presented herein will contribute to the ongoing discourse and inspire further research in this area.

Table of Contents

List of Figures

List of Tables

Nomenclature

Game

A game is a formal description of a strategic situation (Stengel and Turocy, 2003).

Markov models

Markov models are stochastic models used for randomly changing systems (Gagniuc, 2017).

Petri nets

A Petri net is a mathematical model used to represent and analyze concurrent systems, comprising places, transitions, and arcs to model state changes and transitions (Murata, 1989).

Policy

It is the conversion of states into actions, dictating what the agent ought to do in a given condition. (Zhu, Lin, Jain et al., 2023).

Source task

The original task or learning problem that the agent has already been trained on or has some prior experience with (Sutton and Barto, 2018).

Target task

A new task or learning problem that the agent aims to solve or improve its performance on (Sutton and Barto, 2018).

Abbreviations

AEC Agent Environment Cycle.

API Application Programming Interface.

BDD Binary Decision Diagram.

CNN Convolutional Neural Netwok.

DDQN Double Deep Q-learning Network.

DFT Dynamic Fault Tree.

DL Deep Learning.

DNN Deep Neural Network.

DPB Dynamic Potential Based approach.

DPBA Dynamic Potential-based value function Advice.

DPID Direct Policy Iteration with Demonstrations.

DQfD Deep Q-learning from Demonstrations.

DQN Deep Q-Network.

FDEP Functional dependency.

FMEA Failure Mode and Effects Analysis.

FTA Fault Tree Analysis.

GAE Generalized Advantage Estimation.

GAIL Generative Adversarial Imitation Learning.

IRL Inverse Reinforcement Learning.

LfD Learning from demonstrations.

MARL Multi-Agent Reinforcement Learning.

MATL Multi-Agent Transfer Learning.

MBSE Model Based Systems Engineering.

MDP Markov's Decision Process.

ML Machine Learning.

MPE Multi Particle Environment.

MSE Mean Squared Error.

PAND Priority-AND.

PBA Potential-Based state-action Advice.

PBRS Potential Based Reward Shaping.

PDEP Probabilistic dependency.

POMDP Partially Observable Markov Decision Process.

POR Priority-OR.

PPO Proximal Policy Optimization.

ReLU Rectified Linear Unit.

RL Reinforcement Learning.

SAIL Self-Adaptive Imitation Learning.

SEQ Sequence enforce.

SFT Static Fault Tree.

SGA Stochastic Gradient Ascent.

SGD Stochastic Gradient Descent.

SPARE Spare gate.

SR Successor Representation.

SysML Systems Modeling Language.

TL Transfer Learning.

TRPO Trust Region Policy Optimization.

UML Unified Modeling Language.

UVFA Universal Function Approximation.

1 Introduction

Advancements in science and technology, driven by evolving consumer preferences and various other factors, are leading to increasingly complex systems across diverse fields. Whether it's in aeronautics, nuclear power plants, or even simple home appliances, every system is becoming more intricate. However, this rise in complexity poses significant challenges for risk analysis and mitigation. Traditional methods such as FTA and Failure Modes and Effects Analysis (FMEA), once effective in managing risks, are now struggling to cope with the intricacies of modern systems. These systems exhibit dynamic interactions and unpredictable behaviors, which traditional methods often fail to address adequately. Hence, there is a mounting need for more sophisticated approaches to fault analysis and the development of fault-tolerant systems. As complexity continues to rise, it becomes increasingly evident that relying solely on conventional techniques is insufficient to ensure the reliability and safety of these intricate systems. By embracing innovative fault analysis techniques like DFTs, and leveraging the support of various Machine Learning (ML) tools, it is possible to tackle these challenges and develop safer and more resilient systems.

DFT is an extension of the Static Fault Tree (SFT) that allows for the modeling of sequence dependencies in the system. Unlike SFTs, DFTs can model dynamic systems where conditions change over time, such as repairs, maintenance, or system reconfiguration. Since DFTs consider failure sequences instead of boolean combinations, we cannot use a Binary Decision Diagram (BDD) to represent them (Junges et al., 2016). Instead, they are represented by Markov modelss or Petri netss.

In fault analysis, the initial step involves modeling the system. Model-Based Systems Engineering (MBSE) offers a structured approach to system engineering, utilizing modeling to simulate system behavior, assess performance, and evaluate the repercussions of system modifications (Shevchenko, 2020). In the context of this research, the system model is represented as a DFT which is then translated into a game. This game involves two adversarial agents, one attempting to cause a system failure and the other striving to prevent it. The outcomes of their interactions are used to refine the model, resulting in a new version of the game. Through iterative refinement, this process aims to develop a more fault-tolerant system.

Each of the agents is trained within the created game environment using RL techniques. RL, a branch of ML, focuses on training agents to perform tasks within unknown environments (Mnih et al., 2013). A typical RL problem involves an environment that follows Markov's Decision Process (MDP). Here, the agent begins with an initial state and proceeds to subsequent states by executing actions, with each action yielding a reward guiding future actions. The learning objective is to determine the best policy, which determines the agent's likelihood of acting in a given way in a given state. This is accomplished by iteratively updating the policy following each interaction with the environment.

A multi-agent system consists of an environment and multiple agents with individual goals interacting with it (Albrecht, Christianos and Schäfer, 2024). Agents receive the state description of the environment and act to manipulate the system state to their advantage, earning individual rewards based on their actions. However, in some scenarios, agents may only have partial access to the environment's state description, a situation known as a Partially Observable Markov Decision Process (POMDP). In such cases, the optimal policy is derived by selecting actions that maximize the agent's reward, by assuming that the adversary also performed the best action for itself (Da Silva and Costa, 2019). The primary learning objective in such scenarios is to achieve the Nash Equilibrium. While these methods can yield successful solutions, they often suffer from inefficiencies due to the large number of required agent-environment interactions. TL addresses this challenge by enhancing the learning performance of agents in the target domain through the transfer of knowledge acquired from a related source domain. The objective here is to utilise both external information from the source domain and internal information from the target domain, to identify the optimal policy for the target domain (Da Silva and Costa, 2019).

This research delves into the performance evaluation of two widely used RL algorithms, namely Q-learning and Proximal Policy Optimization (PPO) within an adversarial environment utilizing TL. Agents are represented by Convolutional Neural Networks (CNNs), where the initial layers

learn the low-level information from the given task. Upon completion of learning for a particular environment, the parameters of these initial layers are transferred to the corresponding agents' initial layers in the subsequent environment. These parameters are then fine-tuned to adapt to the new task. This approach facilitates a more efficient learning process, enabling agents to effectively navigate complex environments and achieve desired outcomes.

2 Literature Review

2.1 Model-Based Systems Engineering

In modern engineering practices, MBSE stands out as a prominent approach, offering a structured methodology for system design, analysis, and optimization (Shevchenko, 2020). Through the creation of models, MBSE facilitates a comprehensive understanding of the system, enabling engineers to effectively manage its complexity and enhance its performance. System modeling is typically conducted using standardized modeling languages such as Unified Modeling Language (UML) and Systems Modeling Language (SysML). These languages provide standardized notations for expressing system structures, behaviors, and interactions, thereby facilitating clear communication among system components.

MBSE offers valuable capabilities for fault analysis and risk management in complex systems. By capturing system behaviors, interactions, and dependencies through models, engineers can systematically analyze potential failure modes, identify critical failure pathways, and assess the impact of failures on system performance and safety. However, despite its numerous benefits, MBSE also presents challenges in practical implementation. These challenges include the complexity associated with model development, interoperability issues between modeling tools, and the necessity for standardized model libraries and methodologies. Future research directions may focus on addressing these challenges through the development of automated model generation techniques, improved tool interoperability standards, and advanced model-based analysis methods.

2.2 Game Theory

Games theory is the study of interactions and conflicts between multiple players or agents within a given environment (Burguillo, 2018). It offers a mathematical framework for analyzing potential strategies employed by players in both cooperative and competitive settings. In a game, multiple players interact with one another, with each player's outcome influenced by the actions of all the other players. The choices made by the players lead to different rewards, which represent the value of the outcome. By anticipating what the other players will do, each player seeks to maximize their rewards. The game models are mainly classified into the following types:

1. Cooperative Games: These are the games where the agents collaborate and establish coalitions to achieve a common goal.
2. Non-cooperative Games: In this kind of game, each player acts independently and chooses their strategy to maximize their rewards while taking into account the strategies chosen by the other players. This model is ideal for adversarial situations where agents are competing against each other and aiming for the best outcome for themselves.

In adversarial multi-agent systems, non-cooperative game models are often represented as strategic form games, which are usually represented in matrix form. Every row indicates a single player's approach, while every column indicates a different player's plan. The reward for each agent depends on the combination of strategies chosen by all the agents. The optimal solution in such games is known as the Nash equilibrium, wherein no player can unilaterally improve their reward given that other players maintain their strategies fixed (Da Silva and Costa, 2019). However, in adversarial multi-agent systems, multiple Nash equilibria may exist, some of which could be undesirable or unstable. In such instances, alternative solution concepts, such as the minimax solution, are

employed to identify optimal outcomes. The minimax solution aims to minimize the maximum possible loss for an agent, assuming that other agents play optimally.

This concept is particularly relevant in zero-sum games, where the total rewards of all players sum to zero. Zero-sum games are prevalent in adversarial multi-agent systems where the interests of agents are opposed, resulting in one agent's gain equating to another's loss. Here, agents utilize the minimax solution to mitigate maximum losses or maximize minimum gains, optimizing their strategies within the competitive landscape.

2.3 Dynamic Fault Tree

FTA is a crucial technique for assessing and mitigating risks associated with complex systems. It involves constructing a graphical representation, known as a fault tree, which illustrates the logical relationships between the components of a system and the events that can lead to failure. The fault tree resembles a tree structure, with component failures depicted as leaves and gates representing the propagation of these failures. To analyze fault trees, they are typically converted into BDDs (Junges et al., 2016).

SFTs are foundational in FTA and primarily consist of basic gates like AND and OR, alongside Voting OR (k-out-of-n) gates, which fail if a specified number of signals out of a set fail. SFTs are used to model the static behavior of systems, assuming component independence and system steadiness. However, they cannot capture the dynamic system behavior, such as temporal dependencies. DFTs extend the capabilities of SFTs by incorporating sequence dependencies in system modeling. Unlike SFTs, DFTs can represent dynamic systems where conditions evolve due to factors like repairs, maintenance, or reconfiguration. Since DFTs consider failure sequences rather than boolean combinations, they are represented using Markov models or Petri nets.

In addition to gates used in SFTs, DFTs introduce several additional gates to capture dynamic dependencies and behaviors:

- Priority-AND (PAND) gate: Fails only if its children fail from left to right.
- Priority-OR (POR) gate: Fails if the first child fails before any other child does.
- Spare gate (SPARE): Manages shared spare modules, switching to a non-failed child when the primary fails.
- Sequence enforcer (SEQ): Ensures failures occur only from left to right without propagating faults.
- Functional dependency (FDEP): Causes dependent events to fail when its trigger fails, without propagating faults.
- Probabilistic dependency (PDEP): Extends FDEP with a probability factor, determining the likelihood of dependent event failure.

Failure propagation in DFTs involves both upward propagation and propagation through FDEPs. This comprehensive approach allows for a detailed analysis of system failure scenarios, aiding in the development of effective risk mitigation strategies.

2.4 Reinforcement Learning

RL is a branch of ML where an agent is trained to make decisions by interacting with an MDP-modeled environment to accomplish a predetermined objective. Rewards are given to the agent as feedback. Based on the rewards or penalties it receives for its actions, the agent gradually modifies its behavior to maximize cumulative rewards (Sutton and Barto, 2018). RL is commonly used in situations where explicit instructions or labeled data are unavailable, but the agent can learn through trial and error (Zhu, Lin, Jain et al., 2023). An MDP is represented by a tuple, $(\mu_0, S, A, T, \gamma, R)$, where:

- μ_0 is the set of initial states.
- S represents the state space.
- A represents the action space.
- T is the transition probability distribution such that the chances of the state shifting to s' by taking action a at state s is given by $T(s' \mid s, a)$.
- $\gamma \in (0, 1]$ is the symbol for discount factor.
- R is the reward distribution such that, $R(s, a, s')$ gives the reward that the agent gets by taking action a to move from state s to state s'.

The agent in RL follows a Policys π, and the learning objective is to find the optimal policy π^* that maximizes the mean rewards:

$$\pi^* = \max_{\pi} \mathbb{E}_{(s,a)\sim\mu^\pi(s,a)} \left[\sum_{k=0}^{\infty} \gamma^k r_k \right] \tag{1}$$

where $\mu^\pi(s, a)$ is the stationary state-action distribution induced by policy π, and r_k is the reward received after k steps (Zhu, Lin, Jain et al., 2023).

This is done by iteratively updating the Q-function after each interaction with the environment:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{2}$$

where α is the learning rate, γ is the discount factor, and r is the reward for taking action a in state s to transition to the new state s' (Devlin and Kudenko, 2011).

It is also noteworthy to mention the difference between rewards and goals. Each goal can be differentiated by its reward function. Rewards are the messages that the agent gets from the environment to evaluate its actions, while goals are the desired outcomes that the agent aims to achieve. The following three terms are associated with RL to represent the cumulative rewards for the agent being in a particular state and taking certain actions from that state:

- Value function: This function calculates how good it is to be in a specific state s.

  $$V^\pi(s) = E[r_0 + \gamma r_1 + \gamma^2 r_2 + \ldots] \tag{3}$$

  Here, $r_i = R(s_i, a_i, s_{i+1})$ is the reward the agent gets for taking the action a_i to traverse from the state s_i to s_{i+1} (Zhu, Lin, Jain et al., 2023).

- Q-function: This function calculates how good the action taken at a particular state s is. It indicates the anticipated future rewards gained by performing the particular action in the specified state. Equation 4 gives the formula for this fuction (Zhu, Lin, Jain et al., 2023).

  $$Q^\pi(s, a) = E[R(s, a, s') + \gamma V^\pi(s')] \tag{4}$$

- Advantage function: In a given state, this function estimates the relative benefit of a specific action over alternatives, under a given policy (Sutton and Barto, 2018).

  $$A^\pi(s, a) = Q^\pi(s, a) - V^\pi(s) \tag{5}$$

2.4.1 Q-learning

Q-learning is a popular RL algorithm in which the Q-value function is estimated using the Bellman equation, iteratively, till it converges to the optimal value (Mnih et al., 2013).

$$Q_{i+1}(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a \right] \tag{6}$$

Here, the optimal Q-value, denoted as $Q^*(s, a)$, represents the highest Q value achievable for the specific pair of state and action by following a strategy. This value serves as a crucial metric for the agent, aiding in the selection of the most favorable action within a given state. During each iteration, the agent actively engages with the environment, executing actions and receiving corresponding rewards. This iterative learning process enables the agent to continually refine its strategy in pursuit of maximizing its expected future rewards.

In Deep Q-learning, CNNs play a pivotal role in estimating the Q-value function $Q(s, a; \theta)$, which seeks to determine the optimal Q-function $Q^*(s, a)$ by learning the parameters θ. CNNs are particularly well-suited for this task due to their ability to effectively process spatial information. Training of the Q-network involves minimizing the loss function $L_i(\theta_i)$, which evolves with each iteration.

$$L_i(\theta_i) = \mathbb{E}\left[(Q_i(s, a) - Q(s, a; \theta_i))^2\right] \tag{7}$$

This loss function measures the discrepancy between the predicted Q-value $Q(s, a; \theta_i)$ and the target Q-value $Q_i(s, a)$, where Q_i is derived from the Bellman equation. This loss function, optimized through stochastic gradient descent, facilitates the update of network weights based on the gradient of the loss.

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}\left[(Q_i(s, a) - Q(s, a; \theta_i))\, \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{8}$$

where,

$$Q_i(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) \mid s, a\right] \tag{9}$$

In this equation, γ represents the discount factor, and r is the instantaneous reward obtained after taking action a in state s. θ_{i-1} denotes the parameters of the Q-network from the previous iteration (Mnih et al., 2013).

Conventional Q-learning is affected by maximization bias, where the expected maximum can be overestimated (He, Yang and Shen, 2019). This occurs because the same samples are being used to decide the best action, and also to estimate its value. As a result, the agent may become overly optimistic about the rewards associated with certain actions, which can impact its decision-making process and the overall performance of the learning algorithm. To overcome maximization bias and maintain a stable target in Q-learning, a technique called double Q-learning is utilized. In double Q-learning, two separate networks are employed: the main network and the target network. The main network determines the best action in the next state, while the target network helps estimate the target Q-value.

$$\hat{a}'_{\text{main}} = \max_{a'} Q(s', a'; \theta_{\text{main}}) \tag{10}$$

$$Q_{\text{target}}(s, a) = \mathbb{E}[r + \gamma Q(s', \hat{a}'_{\text{main}}; \theta_{\text{target}}) | s, a] \tag{11}$$

where, $\hat{a}'_{\text{main}}$ represents the action chosen by the main network, while $Q_{\text{target}}(s, a)$ denotes the target Q-value calculated. In this case, θ_{main} denotes the parameters of the main network, and θ_{target} denotes the parameters of the target network (He, Yang and Shen, 2019).

Consecutive samples in RL often have strong correlations, which can lead to inefficient learning and unwanted feedback loops (Wang et al., 2016). To avoid that, an experience replay buffer is used. It is a dataset, $D_t = \{e_1, e_2, \ldots, e_t\}$, that the agent maintains while learning, where $e_t = (s_t, a_t, r_t, s_{t+1})$ is the experience. While training the network, instead of using only the current experience, a random minibatch is sampled from the buffer to train on.

Thus the loss function given in equation 7 gets modified to:

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s') \sim U(D)}\left[\left(Q_{\text{target}}(s, a) - Q(s, a; \theta_i)\right)^2\right] \tag{12}$$

In many RL problems, the value of each action is not equally important. In some cases, it may be more important to know the value of a state than the value of each action in that state. To take it into account, a dueling network architecture is considered, in which the last layers of the normal convolutional structure are split into separate streams to predict the value function $V(s)$ and an advantage function $A(s, a)$, from which $Q(s, a)$ can be computed (Wang et al., 2016).

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + A(s, a; \theta, \alpha) - \frac{1}{|A|} \sum_{a'} A(s, a'; \theta, \alpha) \tag{13}$$

where, $|A|$ is the dimension of the vector $A(s, a'; \theta, \alpha)$. θ represents the parameters of the CNN whereas α and β represent the parameters of the two streams of fully connected layers.

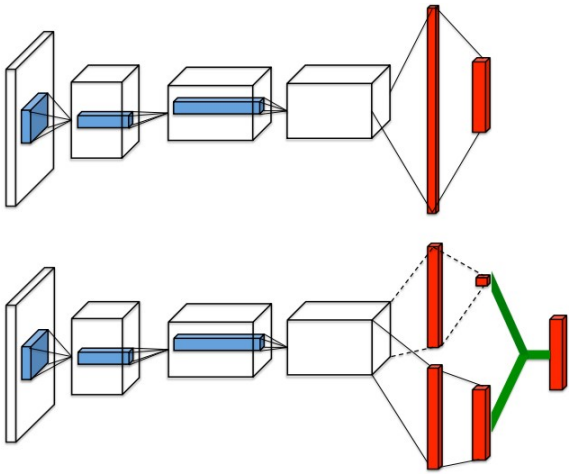

Figure 1: Dueling Network Architecture (Wang et al., 2016)

Figure 1 compares the networks without and with a dueling architecture, with the final layer giving out the Q-values in both cases.

2.4.2 Proximal Policy Optimization

PPO is another popular RL technique, leveraging advantages from the Trust Region Policy Optimization (TRPO) algorithm (Schulman et al., 2017). It offers simplicity in implementation and enhanced sample efficiency. PPO, categorized as a policy gradient method, directly optimizes the policy function to maximize the expected cumulative rewards. The loss function considered here is:

$$L_{\text{PG}}(\theta) = \hat{E}_t \left[\log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{14}$$

where $\hat{A}_t$ is the projection of the advantage function at timestep t and π_θ is the probabilistic policy. The advantage is computed using the Generalized Advantage Estimation (GAE) formula:

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{15}$$

Here, $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$, γ is the discount factor, and λ is the GAE parameter (Schulman et al., 2017).

The agent learns by using a stochastic gradient ascent algorithm where the gradient is estimated from the loss function:

$$\hat{g} = \hat{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right] \tag{16}$$

This expectation is computed over a finite batch of samples, allowing for efficient memory management and effective utilization of parallel processing capabilities. Although it seems desirable to do several optimization steps on this loss L_{PG}, this often results in overly large policy updates. To mitigate the risks associated with excessively large policy updates, PPO adopts a clipped loss function, denoted as $L_{\text{CLIP}}(\theta)$. It ensures stability during training by constraining the probability ratio $r_t(\theta)$. Specifically, the clipped loss function is defined as:

$$L_{\text{CLIP}}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{17}$$

where, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ denote the probability ratio. The clipping is done at either $1-\epsilon$ or $1+\epsilon$ based on whether the advantage is negative or positive (Schulman et al., 2017).

2.5 Multi-Agent Reinforcement Learning

MARL extends RL to situations in which several agents, each with their own goals or aims, interact within the same environment. Every agent in MARL develops the ability to decide on its own while taking other agents' actions and behaviors into account. This introduces additional complexity, as the environment is no longer stationary and the actions of one agent may impact the outcomes experienced by others. MARL is applied in various domains, including robotics, game theory, and decentralized control systems, where multiple autonomous entities need to coordinate their actions to achieve collective objectives. If we extend RL to multi-agent systems, then MDP gets replaced by Stochastic Game, which is also represented by a tuple, $(S, U, T, \gamma, R_{1...n})$, where:

- n is the number of agents.
- S represents the state space.
- U represents the joint action space of all the agents.
- T is the state transition function (in this case, it depends on joint actions rather than individual ones).
- $\gamma \in (0, 1]$ is the discount factor.
- R_i is the reward function of agent i. It relies on joint actions and the state.

In MARL, there are two primary paradigms to consider: one where multiple individual learners employ single-agent RL algorithms, and another where multiple agents act in unison within the environment (Albrecht, Christianos and Schäfer, 2024).

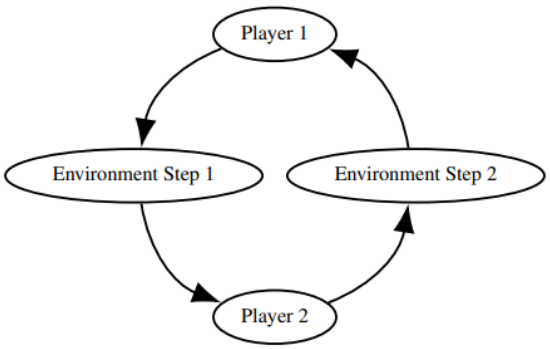

Figure 2: Agent Environment Cycle (Terry et al., 2021)

In the former scenario, each agent operates independently, navigating the environment using its own single-agent RL algorithm. Here, agents make sequential actions, and observations, and receive rewards in isolation, one after another. This dynamic interplay allows the system to adapt to the actions of other agents. An illustrative depiction of this learning paradigm for a two-agent system is outlined in Figure 2.

Conversely, in the latter paradigm, multiple agents collaborate, jointly influencing the environment with their collective actions. At any given state, each agent contributes an action, and their combined efforts dictate the transition to the subsequent state. In this synchronized act, each agent receives individual rewards as a consequence of their actions, reflecting the unique contributions of each participant to the collective endeavor. Figure 3 illustrates this collaborative process.

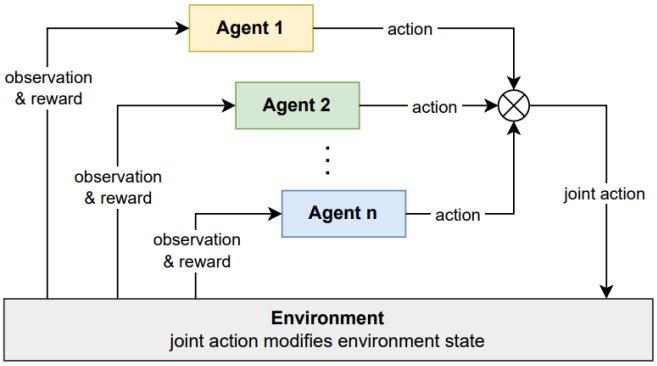

Figure 3: Schematic of MARL (Albrecht, Christianos and Schäfer, 2024)

2.6 Transfer Learning

Learning from scratch can be a daunting task, especially in complex environments. Through the transfer and reuse of knowledge from one related yet distinct source domain to another, TL seeks to enhance the agent's learning performance in a target domain. The fundamental objective of TL is to discover the optimal policy for the target domain by harnessing both exterior insights from the source domain and internal data from the target domain (Da Silva and Costa, 2019).

In TL, the learning process takes place in a knowledge space represented by K, from which a policy π is formed. This knowledge space is made up of many different elements, such as interactions with other agents and samples taken from the source and target domains. Decisions regarding the type, timing, and procedure of incorporating data into and extracting data from this knowledge space are of paramount importance. Consequently, one of the central challenges in TL lies in judiciously managing this process to avoid negative transfer, wherein unrelated knowledge adversely impacts the learning trajectory.

Negative transfer poses a significant obstacle in TL, potentially derailing the learning process. Negative transfer occurs when irrelevant or incompatible knowledge from the source domain impedes progress in the target domain. To counteract this phenomenon, careful attention must be paid to the selection and integration of knowledge from the source domain. Strategies such as feature adaptation, domain alignment, and selective knowledge transfer can mitigate the risk of negative transfer, ensuring that the transferred knowledge complements rather than hinders learning in the target domain.

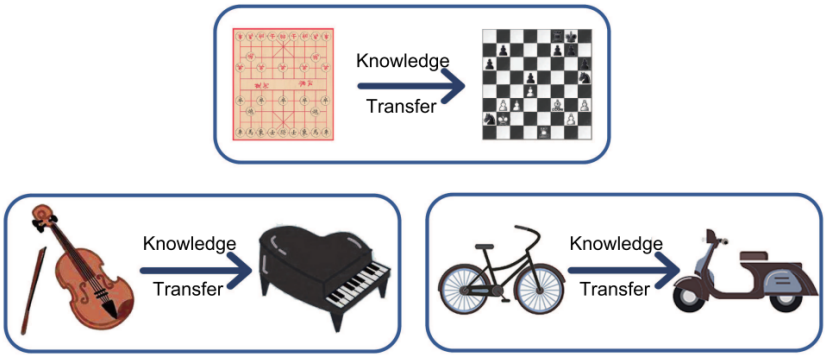

Figure 4: Some intuitive examples for TL (Zhuang et al., 2020)

The traditional MDP is insufficient for TL because it is unable to adequately capture the similarities between the source and target domains (Sherstov and Stone, 2005). To capture various tasks carried out in the same domain, domain (D) and task (τ) are defined separately from this MDP, where $D = (S, A, T)$ and $\tau = (D, R)$.

2.7 Evaluation Metrics

In assessing the efficacy of TL methodologies, a range of metrics are employed to gauge the agent's adaptability to the target domain and its overall performance. These metrics serve as vital benchmarks for evaluating the effectiveness of TL strategies. The following metrics are commonly utilized in academic research to provide comprehensive insights into the transfer learning process:

1. Jumpstart Performance: This metric delineates the initial performance of the agent upon commencing training in the target domain. It offers a snapshot of the agent's early adaptability and readiness to engage with the new environment.
2. Asymptotic Performance: Representing the ultimate performance attainable by the agent after extensive training, asymptotic performance encapsulates the agent's capability to fully exploit the resources and nuances of the target domain.
3. Total Rewards: Calculated as the area under the performance curve, total rewards offer a holistic measure of the agent's cumulative performance over the course of training. This metric provides valuable insights into the overall effectiveness and efficiency of the learning process.
4. Transfer Ratio: The transfer ratio quantifies the relative improvement in performance achieved through transfer learning compared to training from scratch. By comparing the asymptotic performance of the agent with and without TL, this metric elucidates the extent to which transfer learning enhances learning outcomes.
5. Time to Threshold: This metric denotes the duration required for the agent to achieve a predefined performance threshold. It offers insights into the speed at which the agent acquires proficiency in the target domain, thereby informing decisions regarding training duration and resource allocation.
6. Performance after fixed epochs: This metric evaluates the agent's performance after a certain amount of training epochs. It makes it easier to continuously evaluate and modify training methods by offering intermediate performance assessments.

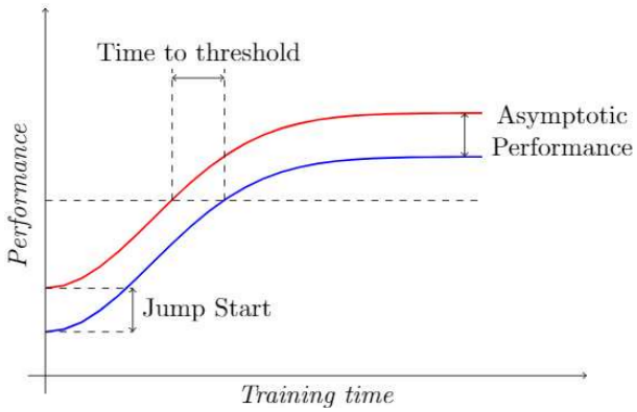

Figure 5: Performance curve of an agent with (red) and without (blue) TL (Da Silva and Costa, 2019)

2.8 Different Approaches for Transfer Learning

As depicted in Figure 6, TL approaches are often classified based on the nature of knowledge transfer. These methods vary depending on the source domain, which provides the knowledge, and the target domain, where the knowledge is applied. By understanding these classifications, researchers can effectively leverage TL techniques to address diverse challenges across various domains.

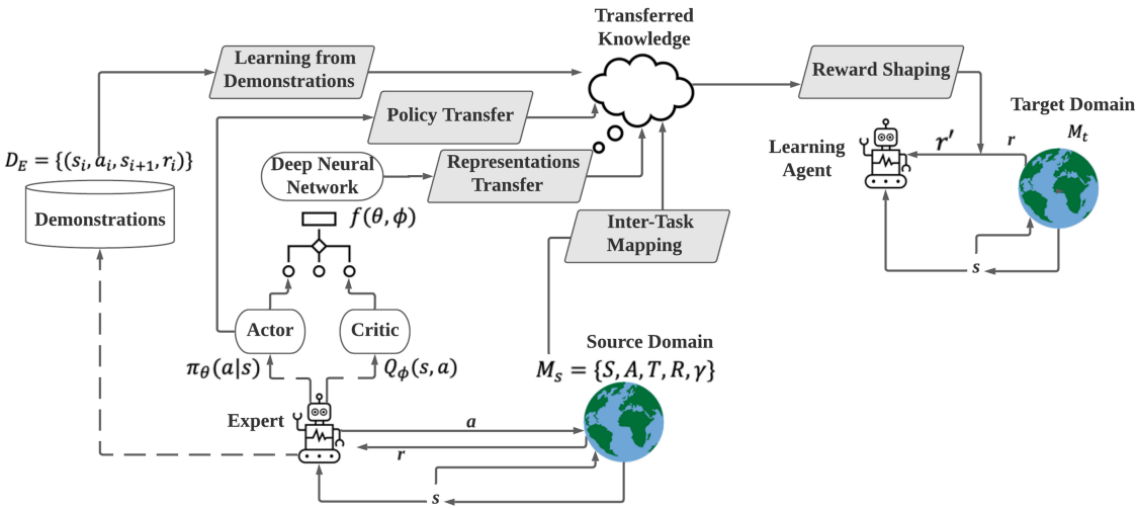

Figure 6: An overview of different TL approaches (Zhu, Lin, Jain et al., 2023)

2.8.1 Reward Shaping

Using an additional reward-shaping function F, this method redistributes the target domain's reward structure based on heuristic information. The reward-shaping function provides extra rewards apart from the already existing rewards, R, in the target domain, to guide the agent to take better actions. Thus, the new reward function that the agent uses for policy learning is: $R' = R+F$. In other words, we can say that the target domain gets altered from $M = (S, A, T, \gamma, R)$ to $M' = (S, A, T, \gamma, R')$.

Reward shaping is often used along with other approaches for better convergence of the learning agent, as it reduces the time needed to learn the optimal policy. There are various definitions for the reward-shaping function:

1. Potential Based Reward Shaping (PBRS): Here, F is the difference between the potential functions $\Phi(\cdot)$, where the potential function evaluates the quality of the given state (Ng, Harada and Russell, 1999).
$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{18}$$
PBRS, when applied to a MAS, increases the probability of convergence to the Nash equilibrium without any information about the reward function or the joint action but can alter the final policy it learns (Devlin and Kudenko, 2011).

2. Potential-Based state-action Advice (PBA): This is an extension of PBRS to overcome its disadvantage of not taking into consideration the quality of actions. Now, $\Phi(\cdot)$ is a function over both state and action space.
$$F(s, a, s', a') = \gamma\Phi(s', a') - \Phi(s, a) \tag{19}$$
Here, a' is the action taken in the next state (s') after the agent transitions from the current state (s). Thus, F is now dependent on both the current transition (s, a, s') and also on the action a' that comes after, which enables a dependence on the policy the agent is currently learning (Wiewiora, Cottrell and Elkan, 2003).

3. Dynamic Potential Based approach (DPB): The earlier two approaches considered the potential function to be static, and the potential function had to converge first for the agent to converge to the optimal policy. But in DPB, $\Phi(\cdot)$ is a function of both state and time, and it was proved that the agent converged to the optimal policy even without the potential function converging. This approach helps agents acquire knowledge that has changed while they were learning and can be seen as the dynamic alternative to PBRS (Devlin and Kudenko, 2012).
$$F(s, t, s', t') = \gamma\Phi(s', t') - \Phi(s, t) \tag{20}$$

Here, t is the time the agent arrived at the state s, and t' is the time it reached the next state s'.

4. Dynamic Potential-based Value Function Advice (DPBA): Similar to DPB, this approach serves as the dynamic alternative to PBA.

$$F(s, a, t, s', a', t') = \gamma\Phi(s', a', t') - \Phi(s, a, t) \tag{21}$$

With this framework, we can shape the reward function of the target domain based on behavioral knowledge, which is not possible with other approaches. Suppose, from expert knowledge, we have an arbitrary reward $R^\dagger$ and wish to achieve $F \approx R^\dagger$ without losing policy invariance. Then, according to PBA, the potential function should satisfy:

$$\gamma\Phi(s', a') - \Phi(s, a) = F(s, a, s', a', t') = R^\dagger(s, a) \tag{22}$$

The additional state-action value function $\Phi(\cdot)$ is introduced to accommodate $R^\dagger(s, a)$. It is a dynamic function learned over time along with the policy. According to Bellman's equation:

$$\Phi^\pi(s, a) = r^\Phi(s, a) + \gamma\Phi^\pi(s', a') \tag{23}$$

which implies that $\Phi(\cdot)$ learns on the negation of $R^\dagger$ (Harutyunyan et al., 2015).

Advantages	**Disadvantages**
• It accelerates the learning process by providing incentives to guide the agent toward the desired behavior. • It allows experts to impart their knowledge into the learning process. • It helps the agent avoid being stuck in undesirable states by penalizing the agent.	• Reward shaping can cause the agent to prioritize short-term gains at the expense of long-term optimal policy. • Incorrect design of reward functions can cause unintended behaviors to get rewarded and ultimately lead to undesired outcomes • If the expert's understanding of the problem is different from the actual one, then the heuristics used could be inaccurate and can lead to bad reward shaping.

Table 1: Advantages and Disadvantages of Reward Shaping

2.8.2 Learning From Demonstrations

Learning from demonstrations (LfD) is an approach where the agent in the target domain is presented with a data set D_E, containing the interactions of an expert with the source domain. The target agent, like in supervised learning, uses this data to learn the policy. In other words, the agent learns to perform a task by imitating the behavior of the expert (Correia and Alexandre, 2023).

There are mainly two ways to acquire demonstrations (Fang et al., 2019):

- Direct demonstration, where the learning agent itself performs the demonstration.
- Indirect demonstration, where an external/expert agent performs the demonstration.

The data acquired from demonstrations usually consists of state representation as feature vectors, actions performed by the expert, and rewards for each state-action transition. LfD approaches are generally classified based on when the demonstrations are used for knowledge transfer:

- Offline approaches: These include offline RL, where the demonstration data set alone is used to train the agent, without any interactions with the environment; and, pretraining of RL components using the demonstration data set before the RL agent starts interacting with the environment.
- Online approaches: In these methods, demonstrations are directly used during the agent's interaction with the environment to help it explore more efficiently.

Popular online LfD approaches are:

1. Direct Policy Iteration with Demonstrations (DPID): This approach combines the samples from the demonstration data set (D_E) with the experiences the learning agent gets from the environment (D_π) to estimate the Q-value ($\hat{Q}$) by using the Monte Carlo method. The policy π is derived from $\hat{Q}$ according to:

$$\pi(s) = \max_a \hat{Q}(s, a) \tag{24}$$

   Further policy improvement is done by comparing $\pi(s)$ with the expert policy π_E to define a loss function $L(s, \pi_E)$. The policy is adjusted to minimize this loss (Chemali and Lazaric, 2015).

2. Deep Q-learning from Demonstrations (DQfD): With this method, temporal difference losses and supervised losses are used to pre-train the learning agent using the demonstration data. The agent gets guidance in mimicking the demonstration by the supervised loss, while the temporal difference loss helps it to develop a reliable value function, that fits the Bellman Equation. These lay the foundation for further learning when the agent starts interacting with the environment.

   During pre-training, the agent takes small mini-batches from the demonstration and uses them to update its network with four types of losses:

   - 1-step double Q-learning loss: $J_{DQ}(Q)$
   - n-step double Q-learning loss: $J_n(Q)$

   Both of the above Q-learning losses together ensure that the Bellman equation is satisfied.

   - Supervised large-margin classification loss:

$$J_E(Q) = \max_{a \in A}[Q(s, a) + l(a_E, a)] - Q(s, a_E) \tag{25}$$

     where a_E is the action taken by the expert in the state s, and $l(a_E, a)$ is the loss function to guide the agent to take action near a_E.
   - L2 regularization loss to prevent overfitting on the small mini-batch: $J_{L2}(Q)$

   Therefore, the total loss can be expressed as:

$$J(Q) = J_{DQ}(Q) + \lambda_1 J_n(Q) + \lambda_2 J_E(Q) + \lambda_3 J_{L2}(Q) \tag{26}$$

   where, λ parameters are weight factors.

   Once pre-training is done, the agent begins operating in the environment using its learned policy and updates its network by combining both demonstrated behaviors and experiences it gathers by interacting with the environment. To facilitate this process, the DQfD algorithm manages two distinct replay buffers and selects samples from each with specified priorities. For self-generated data, the supervised loss is not utilized, even though the same loss function is used for sampled data. (Hester et al., 2017).

3. Generative Adversarial Imitation Learning (GAIL): This method allows the agent to learn the policy directly from the demonstration dataset without interacting with the expert (Ho and Ermon, 2016). This is equivalent to the combination of Inverse Reinforcement Learning (IRL), to recover the expert's cost function, and use that to extract the policy by RL. For

that, it introduces the concept of occupancy measure, ρ_π, which is the probability distribution of state-action pairs under the policy $\pi \in \Pi$.

$$\rho_\pi(s,a) = \pi(a|s) \sum_{t=0}^{\infty} \gamma^t P(s_t = s|\pi) \tag{27}$$

where $P(s_t = s|\pi)$ is the probability of landing in state s at time t when following the policy π (Ho and Ermon, 2016). Based on this formula, the occupancy measure of the current policy (ρ_π) and the expert policy (ρ_E) are found. The divergence between these two is minimized to design a new reward function. However, since π_E is unknown, ρ_E is determined from the demonstration dataset. The reward function is found using adversarial training, where a discriminator, D, learns to distinguish between the state-action pair sampled from the expert policy and from the current policy, using a min-max optimization strategy.

The output of the discriminator is optimized as:

$$J_D = \max_D \mathbb{E}_{d_\pi} \log[1 - D(s,a)] + \mathbb{E}_{d_E} \log[D(s,a)] \tag{28}$$

where d_π and d_E are the normalized stationary state-action distributions of π and π_E respectively, with $d_\pi(s,a) = (1-\gamma)\rho_\pi(s,a)$ and $d_E(s,a) = (1-\gamma)\rho_{\pi_E}(s,a)$.

The discriminator's output is used as a new reward:

$$r' = -\log[1 - D(s,a)] \tag{29}$$

which helps to adjust the agent's policy to match the expert policy distribution. The agent uses this reward to maximize the probability of the discriminator being mistaken about whether the action came from the expert or the agent (Ho and Ermon, 2016). In other words, the learning agent tries to maximize its rewards while simultaneously trying to minimize the discriminator's ability to differentiate between its actions and those of the expert, by generating expert-like actions, i.e.

$$\min_\pi \max_D J(\pi, D) = \mathbb{E}_{d_\pi} \log[1 - D(s,a)] + \mathbb{E}_{d_E} \log[D(s,a)] \tag{30}$$

4. Self-Adaptive Imitation Learning (SAIL): Imperfect demonstrations are a challenge faced by LfD. SAIL is an off-policy approach that can learn sparsely rewarded tasks by using a smaller number of suboptimal demonstrations (Zhu, Lin, Dai et al., 2020). Similar to GAIL, this approach also tries for distribution matching between the source policy (expert policy) and the target policy (current policy), but simultaneously, it also asks the agent to deviate from the previously learned lessons to find better behavior compared to the source demonstrations. Thus, the learning objective is to minimize the divergence between d_π and d_E, while simultaneously maximizing the divergence between d_π and d_B, where d_π, d_E, and d_B are the normalized state-action distributions of π, π_E, and a mixture of previously learned policies, respectively.

$$\max_\pi J(\pi) = -D_{\mathrm{KL}}[d_\pi || d_E] + D_{\mathrm{KL}}[d_\pi || d_B] \tag{31}$$

Like the DQfD algorithm, this one also maintains two separate replay buffers, namely the expert replay buffer R_E, sampled from an unknown expert policy π_E, and the self-replay buffer R_B to store the state-action pairs generated by the current policy π. It trains a discriminator to differentiate between the expert demonstrations and the self-generated samples. The output of this is used to generate the reward, which is used by the learning agent.

$$\max_D J_D = \mathbb{E}_{d_B} \log[1 - D(s,a)] + \mathbb{E}_{d_E} \log[D(s,a)] \tag{32}$$

$$r' = -\log[1 - D(s,a)] \tag{33}$$

The learning agent consists of an actor and a critic. The actor tries to increase the reward by choosing suitable actions, while the critic tries to maximize the expected Q-value over the distribution d_B. During training, if any good quality self-generated trajectory is found, then it is added to the expert replay buffer.

Advantages	Disadvantages
• It can be combined with other approaches. • The agent learns without interacting with the environment. Hence this method is much safer. • Faster convergence as there is no need for the agent to do trial and error fashioned exploration of the environment, to find the optimal policy.	• Performance depends upon the quality of the demonstration set. • Difficulty associated with the creation of the demonstration data set.

Table 2: Advantages and Disadvantages of LfD

2.8.3 Policy Transfer

Policy transfer is the technique of using expert policies from one or more source domains to generate a policy for the target domain. It is generally classified into two categories:

1. Policy distillation: It involves transferring the expert policies by compressing the knowledge from multiple sources into an ensemble for the target domain (Rusu, Colmenarejo et al., 2016). This target policy is learned by minimizing the divergence between the expert policy (π_E) and the current policy (π_θ). There are two types of policy distillation (Czarnecki et al., 2019):

   - Teacher distillation: Here, the divergence is minimized by taking the mean over the trajectories taken from π_E, i.e.

   $$\min_\theta \mathbb{E}_{s\sim\pi_E}\left[\sum_{t=1}^{|s|}\nabla_\theta H^\times(\pi_E(s_t)|\pi_\theta(s_t))\right] \tag{34}$$

   - Student distillation: Here, the divergence is minimized by taking the mean over the trajectories taken from π_θ instead of π_E, i.e.

   $$\min_\theta \mathbb{E}_{s\sim\pi_\theta}\left[\sum_{t=1}^{|s|}\nabla_\theta H^\times(\pi_E(s_t)|\pi_\theta(s_t))\right] \tag{35}$$

   Here, θ represents the parameters of the target policy π_θ.

   From another angle, policy distillation can be classified into the following two approaches:

   - Minimize the cross entropy between π_E and π_θ over the actions. The actor-mimic algorithm is a popular example of this approach. It uses deep reinforcement learning and model compression methods to train an agent using guidance from multiple experts. Given a set of source games $S_1, S_2, ..., S_N$ with corresponding expert networks $E_1, E_2, ..., E_N$, the goal is to train a network using the guidance from these expert networks. This guidance is obtained by transforming the Q-values of each expert using a softmax:

   $$\pi_{E_i}(a|s) = \frac{e^{\tau^{-1}Q_{E_i}(s,a)}}{\sum_{a'\in A_{E_i}} e^{\tau^{-1}Q_{E_i}(s,a')}} \tag{36}$$

   where, A_{E_i} is the action space of the expert E_i and τ is the temperature parameter (Parisotto, Ba and Salakhutdinov, 2016).

The knowledge distillation is done by using the following optimization equation:

$$\min_{\theta} \mathbb{E}_{s\sim\pi_\theta}[H(\pi_E(a|s), \pi_\theta(a|s))] + \lambda\|\theta\|_2^2 \tag{37}$$

where, λ is the coefficient of weight decay, and $H(.)$ is the cross-entropy measure (Parisotto, Ba and Salakhutdinov, 2016).

- Maximize the probability of the expert policy meeting the trajectories generated by the learning agent. The distral (distill and TL) algorithm is a popular method under this category. Here, the common behaviors from multiple expert policies are distilled to form a shared policy. KL divergence is then used to regularise this distilled policy, for the target domain. The following objective function is used to do this:

$$\max_{\theta} \sum_{i=1}^{k} J(\pi_\theta, \pi_{E_i}); \text{ where} \tag{38}$$

$$J(\pi_\theta, \pi_{E_i}) = \mathbb{E}_{s\sim\pi_\theta}\left[\sum_{t\geq 0} \gamma^t R_i(a_t, s_t) + \frac{\gamma^t \alpha}{\beta} \log \pi_\theta(a_t, s_t) - \frac{\gamma^t}{\beta} \log \pi_{E_i}(a_t, s_t)\right] \tag{39}$$

Here, α and β are scalar factors for KL divergence. $-\log \pi_{E_i}(a_t, s_t)$ is the entropy term that encourages exploration. $\log \pi_\theta(a_t, s_t)$ is a reward-shaping term, which encourages the agent to take up actions from the distilled policy (Teh et al., 2017).

2. Policy reuse: Policy reuse is a method in which the expert policies are directly used as a weighted sum to form the target policy. Thus, this method uses different policies, that solve different tasks in a particular domain, to bias the policy learning for a similar task in the same domain. The PRQ Learning algorithm is a popular technique under this category. For this, a library of past policies, $L = \{\pi_{E_1}, \ldots, \pi_{E_n}\}$ is maintained, and the reuse gain (W) of each policy is measured. Reuse gain is the total reward obtained by the agent when that particular policy is used in the target domain. To determine whether a particular policy in this library is to be reused or not, we compute the following probability:

$$P(\pi_j) = \frac{e^{\tau W_j}}{\sum_{p=0}^{n} e^{\tau W_p}} \tag{40}$$

Where W_j is the reuse gain of policy π_{E_j} from L and τ is a temperature parameter. W_0 is the average reward the agent gets by following its policy π_θ. The policy with the maximum probability measure is used (Fernández and Veloso, 2006).

Advantages	**Disadvantages**
• Policy distillation helps in compressing the knowledge from the complex source domains to a simpler one. • Policy reuse helps in reducing the need for large labeled data in the target domain. • The models generated using distillation need less memory.	• Policy reuse cannot be used if the source and target domains are dissimilar. • There are chances for negative transfer. • In the case of distillation, fine details from the source domain are lost due to knowledge compression. • Policy Distillation is sensitive to hyperparameters, which need to be tuned carefully.

Table 3: Advantages and Disadvantages of Policy Transfer

2.8.4 Inter-Task Mapping

The Source tasks and Target tasks can vary in many ways. Inter-task mapping is an approach where mapping functions are used to find the correspondences between the source domain and the target domain (M. E. Taylor, Stone and Liu, 2007). For this, the earliest technique was to find the mapping functions X_S and X_A for the states and actions in the target domain, respectively, to find their corresponding counterpart in the source domain, based on the similarity of transitions. From these functions, another mapping function, ρ, for the Q-values was derived, which transforms the Q-value from the source domain to the target domain.

$$X_S(s_{i,t}) = s_{j,s}$$
$$X_A(a_{i,t}) = a_{j,s}$$
$$\rho(Q_s) = Q_t$$

where, $s_{i,t}$ and $a_{i,t}$ are the state and action, respectively, in the target domain, and $s_{j,s}$ and $a_{j,s}$ are the corresponding state and action, respectively, in the source domain.

$$Q_s = Q_{\text{source}}(s_{j,s}, a_{j,s})$$
$$Q_t = Q_{\text{target}}(s_{i,t}, a_{i,t})$$

When we use the Q-value reuse method, $Q_t = Q_t + Q_s = \rho(Q_s)$ (M. E. Taylor, Stone and Liu, 2007).

Another paper by Ammar and M. Taylor, 2011 proposed to use a common task subspace, S_c, to learn the relationship between the two domains. Here only an interstate mapping is learned. The dimensionality of S_c is lower than that of the state representations in the target or source domain and is described by the control problem's definition or by the user. To learn the mapping function X_S, this method takes n_1 number of state-successor state patterns from the target domain and projects it into the subspace. Similarly, n_2 number of state-successor state patterns from the source domain is also taken and projected. The pairs with the minimum distance in this space are determined, and their corresponding states in the source domain and target domain are found.

Further, for each state s_t in the target domain, the corresponding states in the source domain s_s are found using X_S. It then selects the action a_s in the source domain according to the expert policy to find the next state $s'_{s,\pi}$ in the source domain. For each s'_t, X_S is used again to find $s'_{s,X}$. The $(s'_{s,\pi}, s'_{s,X})$ pair with the least distance is found, and the action a_t that corresponds to this minimum value is taken as the best action for s_t.

Advantages	**Disadvantages**
• Faster training due to the availability of useful features from the source task as a starting point. • Inter-task learning acts as a form of regularization and prevents overfitting. • It requires less amount of labeled data from the target domain.	• It is computationally complex. • If there is no similarity between the source task and the target task, then a negative transfer can occur.

Table 4: Advantages and Disadvantages of Inter-Task Mapping

2.8.5 Representation Transfer

Representation transfer is a kind of TL in which knowledge is captured by Deep Neural Networks (DNNs) and transferred from the source domain to the target domain. The idea is to separate the

input space (which can be the state space, action space, or reward space) into different subspaces that are orthogonal to each other and can be used across different tasks. Thus, it is easier to transfer knowledge between tasks as the knowledge that is relevant to a particular task can be isolated and reused for another task. The approaches under this topic are classified into two:

1. Reuse the representations from the source domain:

   - Progressive Net: Progressive networks retain a pool of pre-trained models during the training and extract useful characteristics from these for a new task, by learning lateral connections (Rusu, Rabinowitz et al., 2022). The knowledge transfer in this manner helps to prevent catastrophic forgetting and improves convergence speed. The network comprises several columns, each of which is a policy network related to a particular task. Initially, it contains only a single column for that first task. Later when more tasks are added, the network expands to include more columns. The weights of the neurons in the previous columns remain frozen while their representations are transferred to the new column to aid knowledge transfer. Here, each column defines a policy $\pi(a|s)$ taking as input a state from the environment and outputting the probability of action, following that policy.

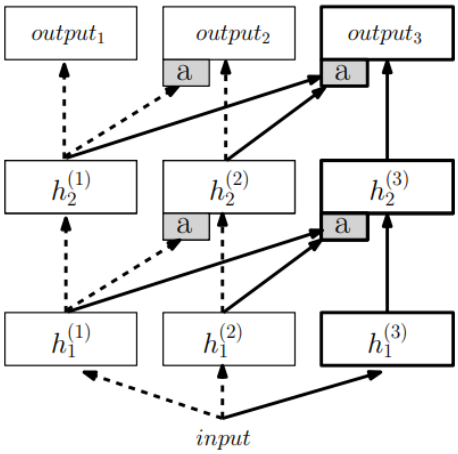

Figure 7: A progressive network (Rusu, Rabinowitz et al., 2022)

   - PathNet: A disadvantage of the Progressive Net is the growth of the network with the number of tasks. This is because it uses a hard-wired design for knowledge transfer. PathNet overcomes this by allowing the relationship between the columns to evolve (Fernando et al., 2017). The knowledge transfer in this network takes place by reusing the optimal pathway in the source task, by fixing its parameters, while allowing the rest of the parameters to change and evolve for the target task. A pathway is a particular route through the neural network that chooses which subset of parameters to use and update in the forward and backward propagation. Each pathway represents a different way of analyzing the input to produce an output prediction. The optimal pathway is selected through an evolutionary process that maximizes the network's performance on a given task.
   - Modular Networks: The policy network is segmented into task-specific and agent-specific modules in this method (Devin et al., 2016). This implies that several agents can do the same task by using a task-specific module. Comparably, many tasks on the same agent may be performed with an agent-specific module. Zhang, Satija and Pineau, 2018 proposed to decouple the learning process into separate modules for the state dynamics model and reward function, where, each module is learned separately.

     The state dynamics module, as shown in figure 8 learns a latent representation of the dynamics of the environment. The encoder and decoder learn a mapping between the state space S and a representation space Z. The forward model predicts the transition probability in Z, and the inverse model takes in the current state and the next state in Z to predict the action to do so.

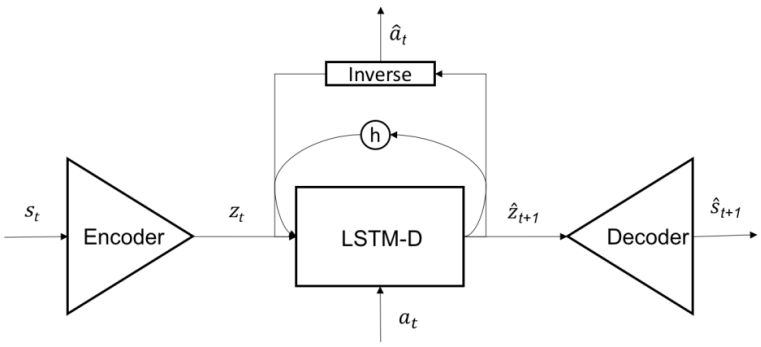

Figure 8: Dynamics Module (Zhang, Satija and Pineau, 2018)

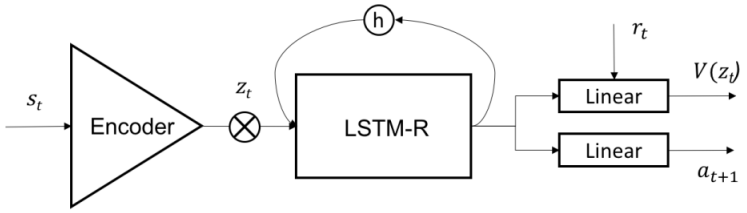

Figure 9: Reward Module (Zhang, Satija and Pineau, 2018)

The actor-critic technique is used by the reward module, shown in figure 9, to learn the value function and the policy in Z. During training, the proposed learning framework is first pre-trained on a set of related tasks to learn a forward dynamics model and an inverse dynamics model. Once the pre-training is complete, the pre-trained model is fine-tuned on a target task with a new reward function. Thus, we can reuse the pre-trained models for task representation and dynamics, which are likely to be relevant to the new task, while only learning the reward function from scratch.

2. Learn to split the source domain representations into independent sub-feature representations:

   - Successor Representation (SR): It is an approach where the state features are decoupled from the reward distribution. Successor Representation is a state representation that captures the underlying structure of the environment without being biased toward any specific task or goal. This type of state representation is useful for TL because it can be used across multiple tasks that share similar underlying structures, even if their reward functions are different.

     For that, the value function is decomposed into two independent components:

     $$V_M^{\pi}(s) = \sum_{s'} \psi(s, s') \cdot w(s') \tag{41}$$

     Where, $w(s')$ is called the reward mapping function, which maps the states to rewards, and $\psi(s, s')$ is the successor representation. It represents a state as the occupancy measure of the future states.

     $$\psi(s, s') = E_{\pi}\left[\sum_{i=t}^{\infty} \gamma^{(i-t)} l[S_i = s'] \Big| S_t = s\right] \tag{42}$$

     In the equation 42,

     $$l[S = s'] = 1 \text{ is the indicator function.} \tag{43}$$

     This method allows knowledge transfer across domains that have only their reward distribution different (Dayan, 1993). Successor representation was originally proposed for discrete state spaces. We get successor features when we extend it to continuous spaces. In successor features, each state is represented by a vector that describes the expected future occurrence of all states under a fixed policy.

Now, the reward associated with the transition (s, a, s') may be written as:

$$r_i(s, a, s') = \boldsymbol{\phi}(s, a, s')^T \boldsymbol{w}_i \tag{44}$$

Where, $\boldsymbol{\phi}$ is a one-hot coded vector, which is the latent feature of (s, a, s') transition and $\boldsymbol{w}$ is the task-specific reward mapper containing the weights.
From this, we get the Q-function:

$$Q^\pi(s, a) = E_\pi \left[\sum_{i=t}^{\infty} \gamma^{(i-t)} \phi_{(i+1)} \middle| S_t = s, A_t = a \right]^T \boldsymbol{w} = \boldsymbol{\psi}^\pi(s, a)^T \boldsymbol{w} \tag{45}$$

$$\boldsymbol{\psi}^\pi(s, a) = E_\pi \left[\phi_{(t+1)} + \gamma \boldsymbol{\psi}^\pi(s_{(t+1)}, a_{(t+1)}) \middle| S_t = s, A_t = a \right] \tag{46}$$

$\boldsymbol{\psi}^\pi(s, a)$ is the successor features of (s, a) and the elements of $\boldsymbol{w}$ contain the values of $r(s, a, s')$.
If $Q_s^\pi = \boldsymbol{\psi}^\pi(s, a)^T \boldsymbol{w}_s$, represents the Q-value for the expert task in the source domain, then for the agent in the target domain, the corresponding Q-value can be derived as:

$$Q_t^\pi = \boldsymbol{\psi}^\pi(s, a)^T \boldsymbol{w}_t \tag{47}$$

Here, we assumed that the reward function is a linear combination of successor features (Barreto, Dabney et al., 2018). However, the reward functions do not always have this linear behavior.
The solution is to learn a matrix $\boldsymbol{\phi}(s, a, s')$ instead of a vector. This matrix consists of the basis vectors of the latent space. If there are D number of individual expert tasks in the source domain out of which n are linearly independent tasks, then those n tasks form the basis vectors. The latent features of the target task can be developed as a linear combination of these basis vectors along with its reward. Thus the latent features can be represented as:

$$\boldsymbol{\phi}(s, a, s') = \boldsymbol{r}(s, a, s') = [r_1(s, a, s'), r_2(s, a, s'), \ldots, r_D(s, a, s')] \tag{48}$$

where the ith element of $\boldsymbol{r}(s, a, s')$ is $r_i(s, a, s')$ which is the reward for the ith task in the source domain.
Similarly, we get the successor features:

$$\boldsymbol{\psi}^\pi(s, a) = [Q_1^\pi(s, a), Q_2^\pi(s, a), \ldots, Q_D^\pi(s, a)] \tag{49}$$

where, $Q_i^\pi(s, a)$ is the Q-value for the ith task in the source domain.
The learned $\boldsymbol{\psi}^\pi$ and $\boldsymbol{\phi}$ are then used to compute the policy for the target task using the Generalized Policy Improvement algorithm (Barreto, Borsa et al., 2019).

- Universal Function Approximation (UVFA): This approach permits TL only for tasks that differ solely in their reward functions, much like the SR. However, unlike SR which is concerned with learning a state representation that is not reliant on the reward function, the purpose of this approach is to identify a function approximator that is suitable for both states and goals. The method involves learning a matrix of latent features through a matrix factorization process that decomposes the UVFA, $V(s, g; \theta)$, into a set of basis functions, $\phi(s)$ and $\psi(g)$. These functions can be used to represent both states and goals.
For an optimal policy π^*, the corresponding optimal value function for a task g is $V_g^*(s)$. In this method, $V(s, g; \theta)$ is used to approximate the optimal value functions over the state and goal spaces.

$$V(s, g; \theta) \approx V_g^*(s) \tag{50}$$

Here, $V(s, g; \theta) = \phi(s)\psi(g)$, where $\phi(s)$ represents the state embedding function and $\psi(g)$ represents the goal embedding function (Schaul et al., 2015).
The state embedding function $\phi(s)$ is a neural network that maps a state s to a fixed-length feature vector. It captures the relevant information about the state for the given

task. By using the same state embedding function across tasks that differ only in goals, the UVFA can generalize to new tasks with different goals. When a new task with an unseen goal g is encountered, the UVFA can be initialized with the values $V(s, g; \theta)$ learned from a previous task. This is possible because the state embedding function $\phi(s)$ is shared across tasks, and the same state features can be used for both tasks. By initializing the UVFA with the values learned from a previous task, the agent can start with a good estimate of the optimal value function for the new task, which can speed up learning considerably.

Advantages	**Disadvantages**
• These methods reduce the amount of data required to learn a new task, as it leverages knowledge from previously learned tasks. • It can improve the generalization performance of the agent as it encourages the model to learn more abstract and task-invariant representations	• It can be challenging when the source and target tasks are significantly different, as the transferred knowledge may not be relevant or may even be harmful to the target task. • It can introduce bias into the learned model if the transferred knowledge is not properly adapted to the target task. • It can be computationally expensive, as it requires training multiple models on multiple tasks and selecting the best model for the target task.

Table 5: Advantages and Disadvantages of Representation Transfer

3 Methodology

The experiments were conducted within the confines of a baseline environment provided by the PettingZoo library, specifically the "Simple Adversary" environment. This environment served as the foundational arena within which the performance of two distinct RL algorithms, PPO and Q-learning, with TL, was assessed and compared. PettingZoo's "Simple Adversary" environment offered an ideal starting point due to its simplicity and clarity, allowing for a focused examination of the algorithms' capabilities in a controlled setting. This environment, characterized by its straightforward dynamics and clearly defined objectives, provided a conducive backdrop for evaluating the efficacy and adaptability of the chosen algorithms.

Following the initial experiments and analysis of the obtained results, it became evident that the Q-learning technique exhibited considerable promise warranting further exploration within a more intricate environment. Consequently, the focus of the experimentation pivoted towards implementing the Q-learning algorithm within a DFT environment. This transition was motivated by the desire to assess the algorithm's performance and adaptability in a setting characterized by heightened complexity and dynamicity. Within this novel setting, two adversarial agents, "red_agent" and "blue_agent," were introduced. Using the DFT environment offered a unique opportunity to simulate real-world scenarios characterized by complexities and evolving conditions. DFT's capacity to model intricate systems and dynamic fault behaviors provided a robust foundation for the experimental framework. This enriched environment facilitated a thorough evaluation of the Q-learning algorithm's efficacy.

Overall, for all the experiments conducted in this research, the TL process can be summarised as follows:

- Pretrain the neural networks on the given source task to find the acceptable estimation of

the respective Q-values (in case of Q-learning) or policy (for PPO). This helps the agents learn basic strategies.

- Simulating a game between the adversarial agents over this source task to output the final scores. The current task is then modified based on this output and again fed into the neural networks.
- Transfering the knowledge between the agent's networks, i.e. parameters from the corresponding layers of the previous environment (source task) to that of the current environment (target task).
- Tuning the parameters for the target task.

3.1 Experiments with the Simple Adversary Environment

The "Simple Adversary" environment, accessible through the MPE library, is designed for simulating multi-agent scenarios involving adversarial interactions (Mordatch and Abbeel, 2017). Within this environment, there exists one adversary and multiple good agents, as well as multiple landmarks. Notably, the number of landmarks is the same as the total count of good agents, with one among them designated as the target landmark. Both good agents and the adversary have discrete action spaces, with actions allowing movement in four directions (left, right, down, up), as well as a no-action option. The observation space for good agents includes information about their positions and velocities, as well as the relative positions of both landmarks and other agents. Conversely, the observation space for the adversary consists of relative positions of landmarks and other agents. The reward structure within this environment is multifaceted, driven by two main factors: the proximity of good agents to the goal landmark and the distance of the adversary from the goal landmark. Good agents aim to maximize their proximity to the target landmark while thwarting the adversary's attempts to approach it. Conversely, the adversary's objective is to discover the target landmark, which remains unknown to it, and navigate towards it. Consequently, the adversary's reward is computed based on the Euclidean distance between its position and the goal landmark's position.

The versatility of the "Simple Adversary" environment extends beyond traditional usage, thanks to its integration with the Agent Environment Cycle (AEC) and Parallel API capabilities provided by PettingZoo. Through experiments conducted using both APIs, the nuances of agent interactions and system dynamics under different computational paradigms can be explored and compared.

In the Parallel API, agents and the adversary execute actions simultaneously at each time step, creating a synchronized environment where all entities act in concert before receiving their respective rewards. This parallel execution facilitates efficient exploration of system behaviors and interactions, to study strategic decision-making in a concurrent setting. By synchronizing actions and rewards across all agents, the Parallel API offers insights into how simultaneous decision-making impacts system dynamics and performance.

Conversely, the AEC API introduces a sequential execution model where actions are taken in a serialized manner, one after the other. This sequential action resolution fosters interdependence among agents, as each agent's action is influenced by the decisions of its counterparts. Immediate reward feedback upon action execution allows for real-time adaptation and learning, shaping the evolution of system dynamics through iterative decision-making processes. By capturing the temporal dependencies inherent in sequential action resolution, the AEC API helps to delve into the intricacies of agent coordination, competition, and adaptation within dynamic environments.

An important point in an adversarial multi-agent game is the Nash equilibrium, where neither of the agents would have a reward for taking any action over the environment. Nash equilibrium enables us to identify optimal strategies for the agents and to guide them toward stable and balanced states. This is done by using two methods. The first one is reward-shaping where the agents are rewarded positively for desirable actions and negatively for undesirable actions. The second way is to employ exploration strategies in RL to avoid agents getting stuck in an unstable state.

3.1.1 Q-learning

The Q-learning approach employed in these experiments is based on the Dueling Double Deep Q-learning Networks (Dueling DDQN). This framework utilizes CNNs to estimate the Q-values, and later the parameters of the initial layers are transferred to a similar network for the next game. The parameters are then tuned for the new environment, thus enabling better and faster learning from the prior lessons.

To explore system states effectively, an epsilon-greedy method is employed, balancing between exploration and exploitation. Under this strategy, the agent chooses the action with the highest Q-value with a probability of $1 - \epsilon$, while selecting an action at random with a probability of ϵ. Over time, the value of ϵ is gradually reduced, encouraging the agent to exploit learned Q-values more frequently as training progresses. This adaptive exploration strategy lets the agent gradually transition from exploration to exploitation, leading to more effective decision-making.

Furthermore, to enhance learning stability and prevent catastrophic forgetting, an experience replay buffer is utilized. This buffer stores the agent's experiences, such as tuples of state, action, reward, and next state, and randomly samples batches of experiences for training the Q-network. By breaking correlations in sequential experiences and providing diverse training data, experience replay promotes exploration and ensures that valuable past experiences are not forgotten during training.

The core idea behind DDQN is to decouple the action selection from the action evaluation, while the dueling architecture separates the state-value estimation from action-advantage estimation. Both of these together help to mitigate overestimation bias and stabilize learning. The dueling architecture consists of two separate network heads and a merging layer:

- Value stream head: Estimates the value function by providing a single scalar value for each state.
- Advantage stream head: Evaluate the advantage of each action relative to others for a given state.

Thus, the key components of the Dueling DDQN with TL capabilities can be listed as:

- Experience Replay: It stores experiences (tuples of state, action, reward, next state) in a replay buffer during interactions with the environment. Instead of using these experiences immediately for learning, random mini-batch samples are taken from the replay buffer to break correlations in sequential experiences.
- Fixed Q-targets: In standard Q-learning, the same network is used to both select and evaluate an action, which can lead to an overestimation of Q-values. DDQN introduces the concept of fixed Q-targets to mitigate this issue by using two separate neural networks.
  - Main network: The main network takes the state as input and estimates the Q-values for each possible action in that state. The action with the highest Q-value is then selected. Its parameters are updated frequently.
  - Target network: The target network is used to compute the expected maximum future rewards (target Q-values) based on the next state and the action selected by the main network, which are used as the targets during the training of the main network. It is a copy of the main network architecture, but its parameters are frozen for several episodes.
- Loss Function: The loss function guides the training of the main Q-network. It is the Mean Squared Error (MSE) loss between the predicted Q-values and the target Q-values.
- Optimizer: The Adam optimizer is employed for training the Dueling Double Deep Q-learning Network (Dueling DDQN). Adam stands for Adaptive Moment Estimation, an extension of the Stochastic Gradient Descent (SGD) optimizer that adapts the learning rate for each parameter individually based on the past gradients and their magnitudes.

- Action Selection: During training, actions are selected based on an exploration-exploitation trade-off, using an epsilon-greedy strategy. The agent sometimes chooses a random action (exploration) and other times chooses the action with the highest predicted Q-value (exploitation).
- Model Training: The networks are updated iteratively using mini-batch samples from the replay buffer. The main network aims to minimize the temporal difference error between the target Q-values, obtained from the frozen target network, and the predicted Q-values.
- Model Transfer: When transferring networks from a source task to a target task, additional considerations may be necessary to accommodate differences in observation and action space corresponding to the new game environment. This involves adding additional input and output layers to the existing network structure to adapt to the new task requirements.
- Model Tuning: During the model tuning phase, the network parameters transferred from the source task are adjusted to adapt to the requirements of the target task. The learning process occurs in discrete steps determined by the learning step size provided to the Adam optimizer.

  In the initial stages of learning the target task, a technique known as coarse tuning is applied. Here, the parameters of the initial convolutional layers transferred from the source task are frozen, while the remaining parts of the network are updated using comparatively larger learning steps. This coarse-tuning approach is implemented to bring the network near to the optimal point more quickly and the parameters are frozen to prevent catastrophic forgetting, where the network may lose previously learned knowledge while adapting to new data.

  After a fixed number of coarse-tuning steps, the frozen parameters are unfrozen, and the network enters the fine-tuning phase. During this phase, smaller learning steps are employed to refine the network's parameters further. Fine-tuning allows the network to make finer adjustments and learn more subtle patterns specific to the target task.

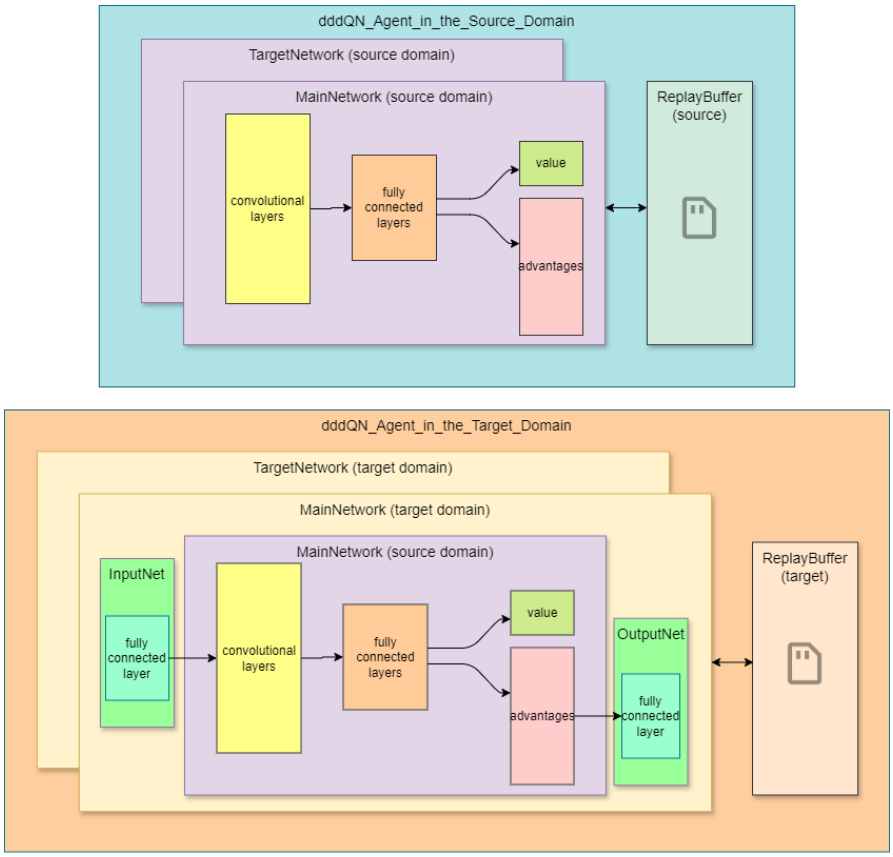

Figure 10: TL in Dueling DDQN - An Overview

3.1.2 Proximal Policy Optimization

PPO is an actor-critic algorithm, meaning each agent is equipped with two CNNs: the actor and the critic. The actor-network is primarily responsible for learning and updating the agent's policy. Given the current state of the environment, it processes this information and produces a probability distribution over the available actions, guiding the agent's decision-making process towards actions that maximize expected rewards. Conversely, the critic network focuses on estimating the value associated with being in a particular state. By evaluating the potential cumulative reward starting from a given state, the critic provides crucial feedback to the actor regarding the quality of its actions. This estimation aids the actor in assessing the desirability of different actions within a given state, ultimately guiding the refinement of its policy.

Both actor and critic networks share approximately similar architecture, featuring convolutional layers for feature extraction, followed by fully connected layers, and finally the output layer. Rectified Linear Unit (ReLU) activation functions are used after each convolutional and fully connected layer except for the final layer. It is a popular choice for activation functions in Deep Learning (DL) due to its simplicity and effectiveness in promoting sparse activations. The actor network uses a sofmax output layer for giving out the probabilities of different actions, possible from the input state. On the other hand, the critic uses a fully connected layer.

Additionally, the PPO framework utilizes a memory component to store experiences encountered during interactions with the environment. By storing and replaying these experiences, the agent can leverage past interactions to improve its decision-making and learning process. While each agent learns from a batch of experiences during each training iteration, it's common practice to clear the memory after each training iteration to prevent it from becoming overwhelmed with outdated experiences. By clearing the memory, the agent can focus on accumulating new experiences more relevant to its current learning objectives, thereby improving its adaptability and ability to navigate changing environments.

The loss functions for updating network parameters differ between the actor and critic networks. In the case of the actor network, the loss function is derived from the negative surrogate objective function, which combines policy gradient and a clipped ratio. This loss function is computed based on the advantage estimate and the ratio of new and old probabilities of actions. It encourages the actor to adjust its policy parameters to favor actions with higher advantages. This aligns with the concept of Stochastic Gradient Ascend (SGA), where the objective is to maximize a given function by iteratively updating network parameters in the direction of the gradient.

Conversely, for the critic network, the loss is calculated as the MSE between the predicted value and the target value. This loss encourages the critic to minimize the error in predicting the value associated with different states accurately. While the actor-network adjusts its parameters to maximize expected rewards through SGA, the critic network adjusts its parameters to minimize the MSE loss through SGD.

This dual optimization process ensures that both networks are effectively trained to fulfill their roles in the RL setting: the actor network learns a better policy, and the critic network learns to better estimate the value function. Both networks utilize the Adam optimizer to update their parameters based on their respective loss calculations. The total loss is a combination of the actor and critic losses, with the actor loss typically weighted more heavily than the critic loss (by a factor of 0.5). This weighting helps maintain balance in the contributions of the actor and critic networks to the overall training process.

During each step of the learning process, agents utilize their current understanding of the environment and the policy learned through training to decide on actions. The actor network takes the current environmental state as input and generates a probability distribution across available actions. From this distribution, the agent stochastically selects an action, considering the likelihood assigned to each action by the policy. By integrating this stochastic decision-making process, the agent effectively balances between exploring new actions and exploiting known favorable actions. This balance enables the agent to uncover optimal strategies while avoiding premature convergence to suboptimal policies. This exploration-exploitation trade-off is fundamental to the agent's ability to learn and adapt to complex and uncertain environments effectively. Through iterative

action selection, execution, and learning from feedback, the agents refine their policies over time, ultimately converging towards actions that maximize their long-term cumulative reward.

Just like in Dueling DDQN, the parameters of the initial layers in the PPO networks can also be transferred to a similar network for subsequent games, facilitating faster learning and knowledge transfer from prior experiences. These transferred parameters are then fine-tuned for the new environment in the same way, to adapt to its specific characteristics and requirements.

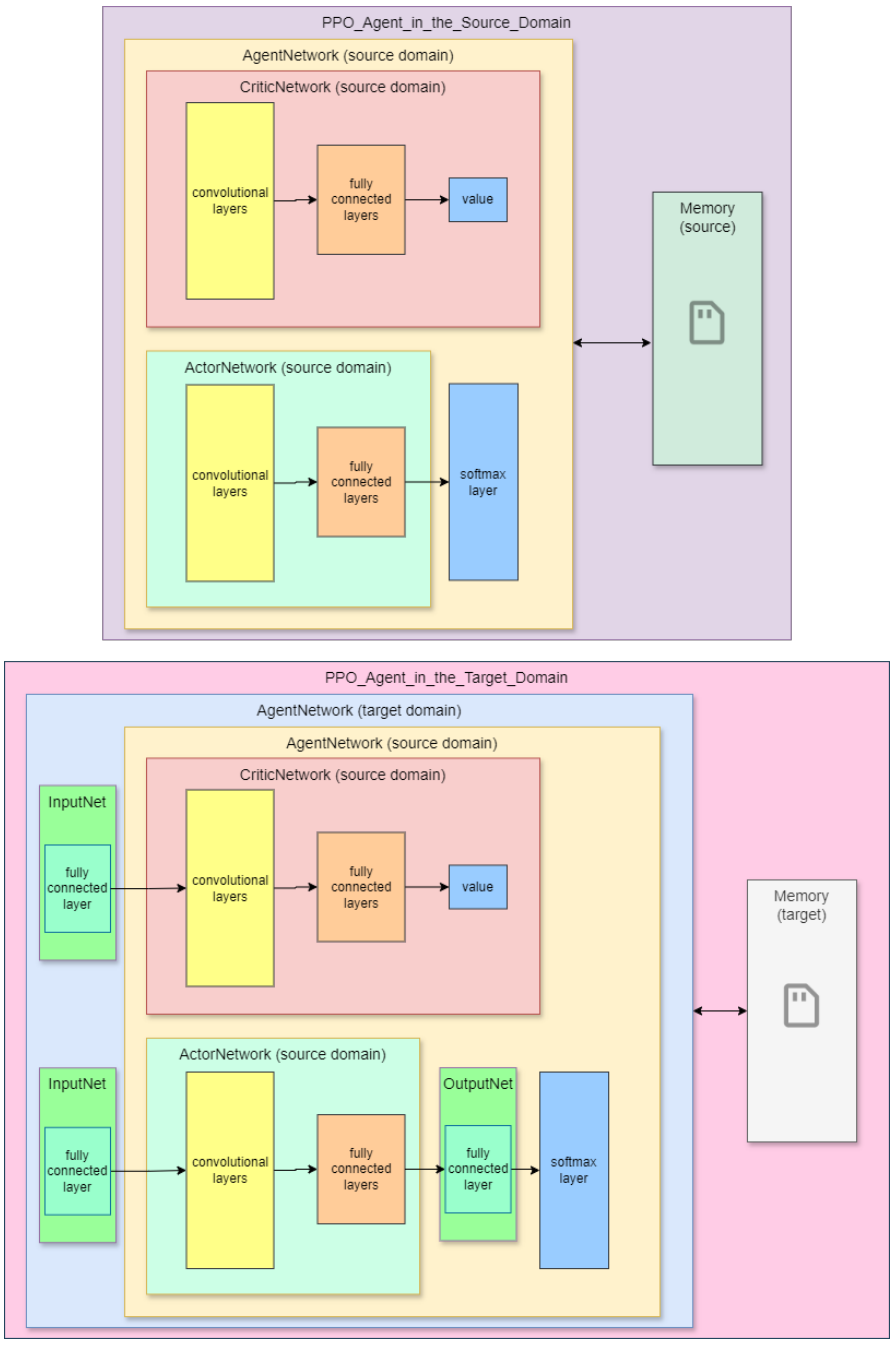

Figure 11: TL in PPO - An Overview

3.2 Implementation in the DFT Environment

1. Environment Setup:
    - Environment Creation: The environment is created dynamically from a DFT defined in

an XML file, representing the structure and behavior of the system under analysis.

- Agent Initialization: The game unfolds between two adversarial agents, the red_agent and the blue_agent, which are instantiated within the environment.

2. Reward Design: The rewards are designed to incentivize the agent behaviors that align with their respective objectives within the DFT environment.

   - red_agent: Aims to induce system faults by introducing failures to specific components within the DFT. It receives a substantial reward of 10,000 upon successfully causing a system fault at any action. No rewards are given for non-fault conditions.
   - blue_agent: Aims to maintain system reliability. It receives a base reward of one for each action taken, provided the system remains fault-free. Additionally, receives a bonus of 1000 if the game ends without any system fault.

3. Observation Space: The observation space in this environment is structured as a Multi-binary space. It dynamically represents the state of each basic event within the fault tree system. Each event can transition between functioning and failed states over time, influencing the observations of both agents.

4. Action Space: Actions correspond to activating or deactivating events within the dynamic fault tree system. Activation actions involve repairing or restoring functionality to a failed event, while deactivation actions induce failures. System-level operations such as skipping a turn or taking no action are also included.

5. Training Algorithm: The Q-learning algorithm using Dueling DDQN is implemented due to its observed efficiency in training compared to PPO, as demonstrated in the Simple Adversary environment.

6. Performance Evaluation: Performance is evaluated based on metrics such as system reliability, fault induction rate, convergence speed, and agent learning efficiency. Agents are trained on tasks built on DFTs with varying numbers of base events and dynamic states. Source tasks with both lower and higher complexities are considered to understand the scalability and robustness of the learning. The validation curves are plotted for the agents by testing them in another instance of the same environment, as they are trained.

7. Experiment Execution:

   - Pretraining in source task: Both the agents are trained in the DFT environment using Q-learning with Dueling DDQN. Trained agents are validated through multiple episodes to ensure consistent behavior and performance.
   - Modification of DFT and Creation of New Environment: Changes are made to the DFT structure, introducing new events, modifying existing ones, or altering fault propagation dynamics. A new game environment is then created based on this modified DFT, reflecting the updated system behavior and fault scenarios.
   - Transfer and Adaptation of Models: Pretrained models from the previous version of the game are transferred to the new environment. These models serve as initial starting points for adaptation. These models are tuned to adjust the model parameters to align with the dynamics and objectives of the updated DFT.

4 Results and Discussions

4.1 TL in the "Simple Adversary" Game

In the experiments with the Simple Adversary environment, the source task involves a game between 1 adversary and 2 agents, resulting in 2 landmarks. Conversely, the target task entails a game between 1 adversary and 3 agents. This means that the complexity and dynamics of the environment increase as the number of players grows, challenging their ability to collaborate and strategize effectively.

Insights gained from each experiment are promptly integrated into subsequent iterations. Performance evaluation of TL is conducted using both the APIs are tested for Q-learning, followed by the selection of the optimal API for PPO implementation.

4.1.1 Training with Duelling DDQN using Parallel API

In the curves for pretraining with Parallel API using the Q-learning algorithm, as depicted in Figure 12, the agents are learning to approximate the Q-value by iteratively playing multiple games. It is observed that the loss curves are decreasing until they reach a minimum point, after which they start to rise again, indicating potential overfitting. Additionally, the step rewards show less fluctuation between steps 3000 and 4000. This implies that the model is optimal in that region, where it has learned to generalize well without overfitting the training data. Furthermore, the validation curves strengthen this inference. We can observe approximately similar rewards for successive validations after step 3000, for all the agents. Thus, it is acceptable to consider stopping the training somewhere near that region, as the models have likely reached a satisfactory level of performance and generalization.

Similarly, in the case of the target task trained from scratch, we could observe an optimal region around step 3000, as shown in Figure 13.

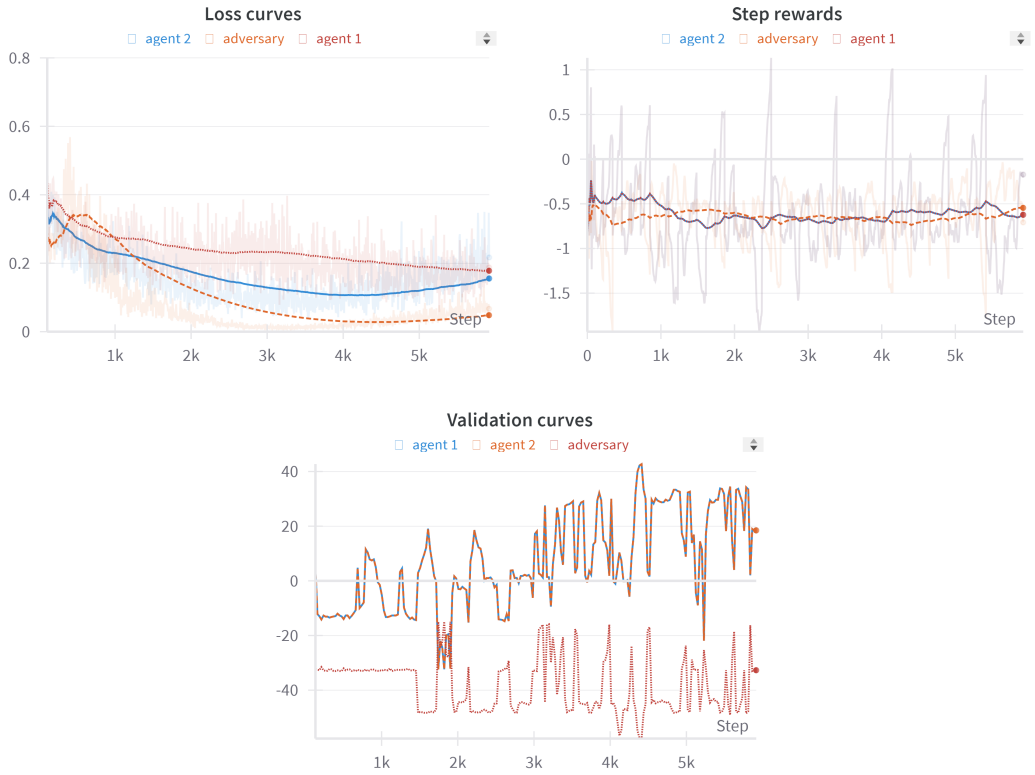

Figure 12: Duelling DDQN (Parallel API) - Pretraining on the Source Task

The Figures 14 and 15 demonstrate the improved learning performance of the agents in the target task compared to that shown in Figure 13, where the agents are learning from scratch. The loss curves exhibit very steep slopes in the beginning, implying faster learning due to the basic knowledge they already have from the source task. An important point to note here is that, while transferring the models from the source task with two agents to the target domain, where there are three agents, the best model for the agent is transferred for all of them. From the loss curves shown in Figure 13, it is observed that agent 2 has the better model. Hence, it is used as the base model for all the agents in the target task.

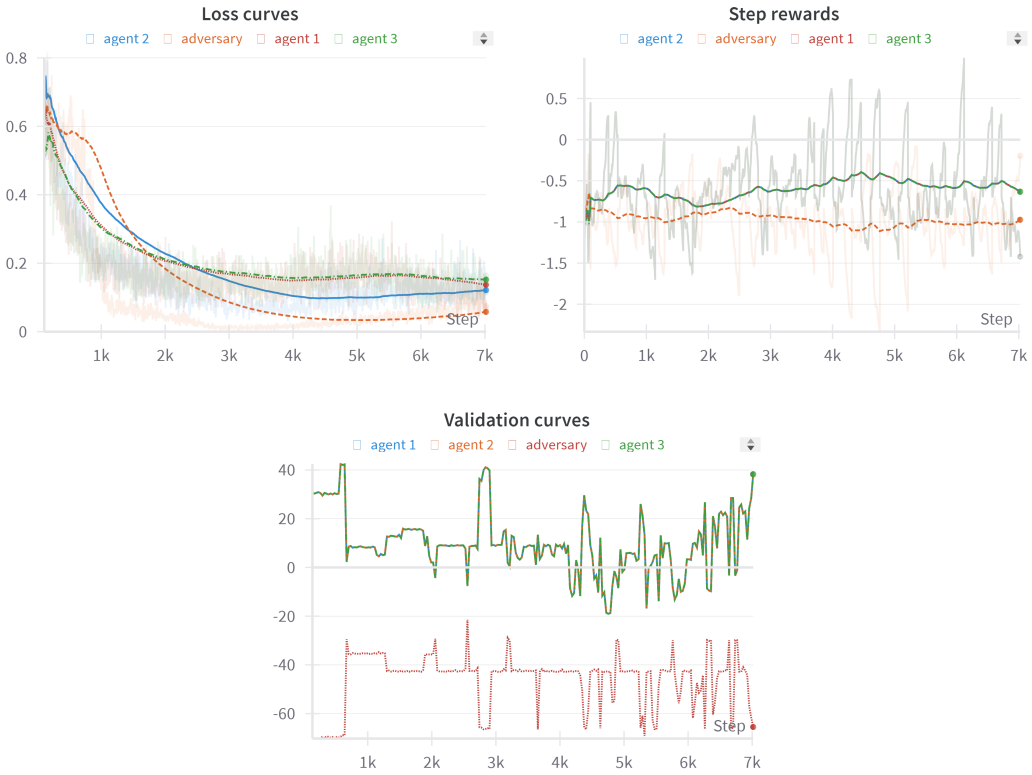

Figure 13: Duelling DDQN (Parallel API) - Target Task Trained from Scratch

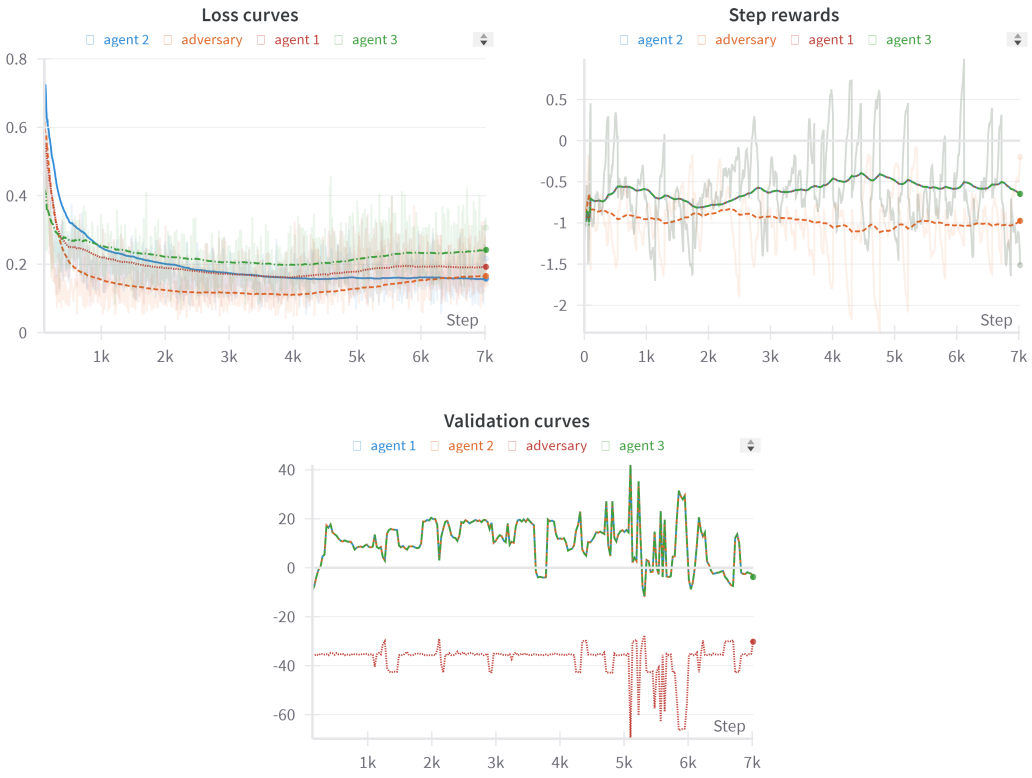

Figure 14: Duelling DDQN (Parallel API) - Target Learned from Source with Frozen CNNs

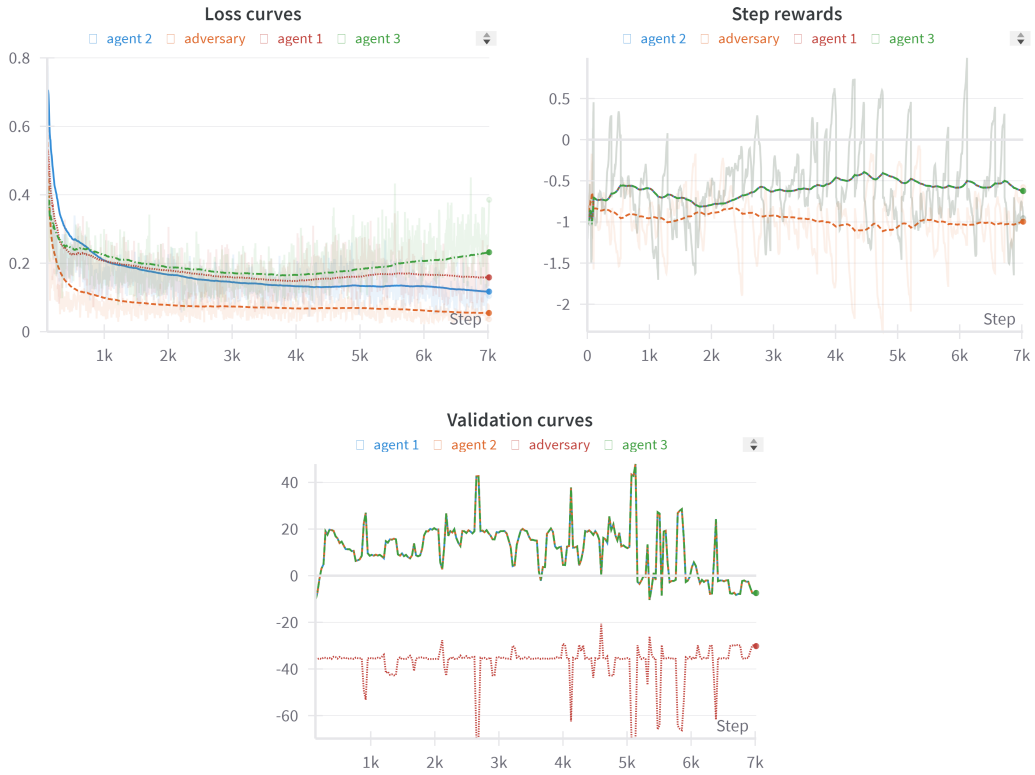

Figure 15: Duelling DDQN (Parallel API) - Target Learned from Source with Defrozen CNNs

Figure 14 depicts the experiment with frozen initial layers. This means that the weights and biases of the CNNs are not changed while learning, and only the other layers of the network are trained on this new task. On the other hand, in Figure 15, the CNNs are also learned to adjust to the new task. Intuitively, the second approach yields better results, and it is evident in the curves. The loss curves are steeper and reach the optimal point without much fluctuation when the CNNs are allowed to adapt along with the other layers of the network.

4.1.2 Training with Duelling DDQN using AEC API

Training using the AEC API introduces a turn-based approach to the game, where players take actions based on the last performed step of the opponent team. While the Parallel API is suitable for applications where all agents perform actions simultaneously without considering the current strategy of other players, the AEC implementation enables agents to adapt their strategies based on the evolving state of the game and the actions of their opponents, making it well-suited for complex systems like DFT.

From the Figures 18 and 19, it is evident that TL enables the agents to learn faster compared to learning from scratch, as shown in Figure 17, even when using the AEC API. Moreover, the TL approach with both coarse and fine-tuning outperforms the one with only fine-tuning of the parameters. In both methods, the CNNs are unfrozen, as it was found to be more beneficial from previous experiments with the parallel API. This indicates that transfer learning, especially when combined with coarse and fine-tuning, can significantly accelerate learning and improve performance in the AEC environment.

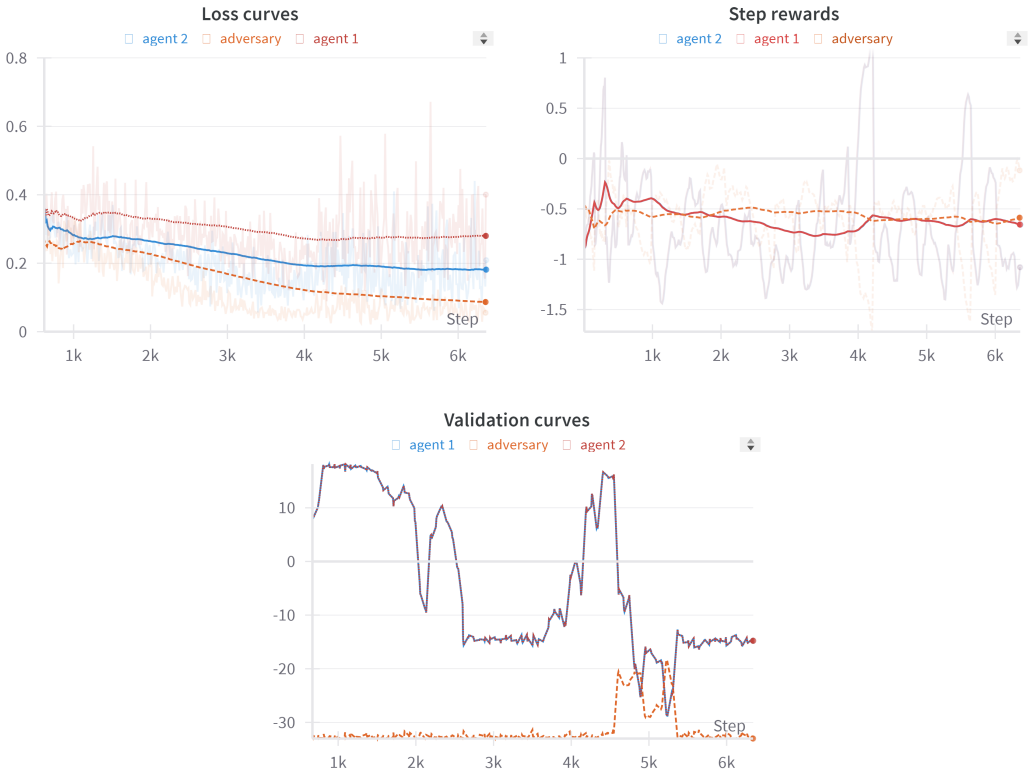

Figure 16: Duelling DDQN (AEC API) - Pretraining on the Source Task

Figure 17: Duelling DDQN (AEC API) - Target Task Trained from Scratch

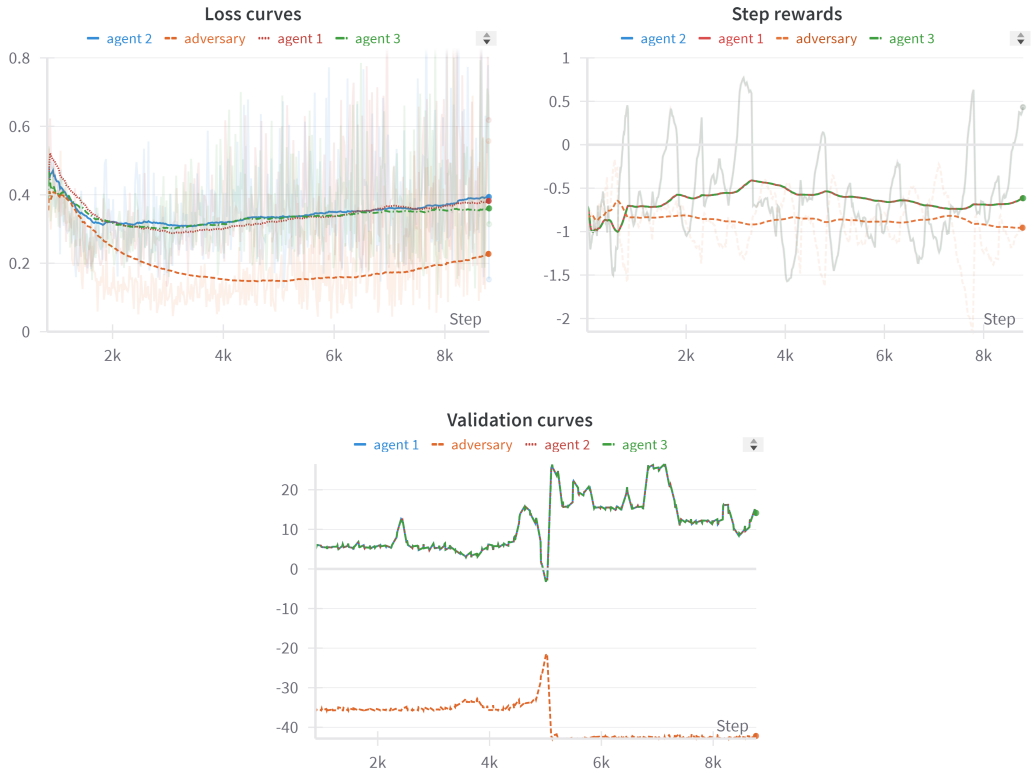

Figure 18: Duelling DDQN (AEC API) - Target Learned from Source (Fine-tuning Only)

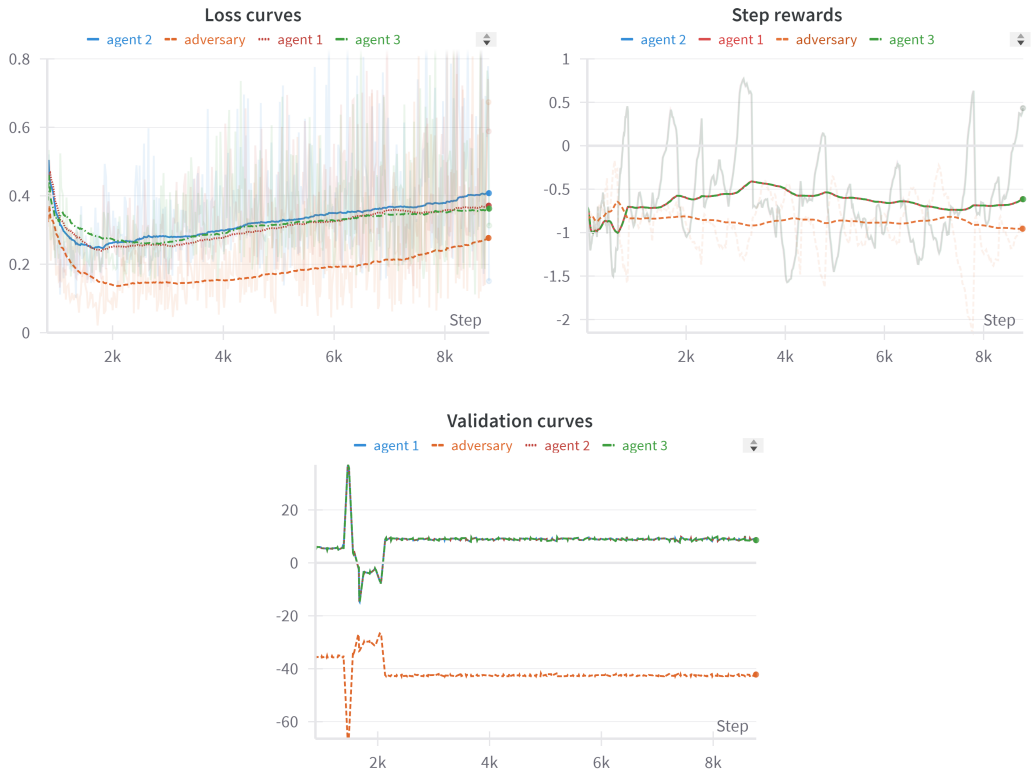

Figure 19: Duelling DDQN (AEC API) - Target Learned from Source (Coarse-to-Fine Tuning)

It is noteworthy to point out that compared to the parallel implementation in Figure 12, the AEC implementation in Figure 16 exhibits lesser fluctuation in the validation curve, resulting in a smoother curve overall. This means that the agents are learning to select actions that lead to more reliable and consistent rewards, rather than relying on random actions. It indicates a higher level of strategic decision-making and learning efficiency in the AEC implementation compared to the parallel implementation.

Coarse tuning allows the agents to rapidly progress toward the optimal point during the initial steps by preventing significant updates to the frozen initial layers. This ensures that the learned lessons are not lost while teaching the rest of the network to adapt quickly to the task environment. By freezing the initial layers, catastrophic forgetting is prevented, as these layers retain the knowledge learned from previous tasks. On the other hand, fine-tuning steps enable the initial layers to adjust to the new settings, further refining the network's performance and allowing it to achieve even better results.

4.1.3 Training with PPO Algorithm using AEC API

From the experiments with the Q-learning technique, it's already known that coarse and then fine-tuning is better for TL. Hence, the same approach is taken in this case with the PPO algorithm. A notable observation is that training takes much time compared to Q-learning, as the agent policies are learned in refining steps with clipping. Therefore, fluctuations in the loss and reward curves, as shown in Figure 20 for the pretraining, are more pronounced. Despite the longer training time and fluctuations, the PPO algorithm still demonstrates the potential for effective TL, especially when following the coarse-to-fine-tuning approach.

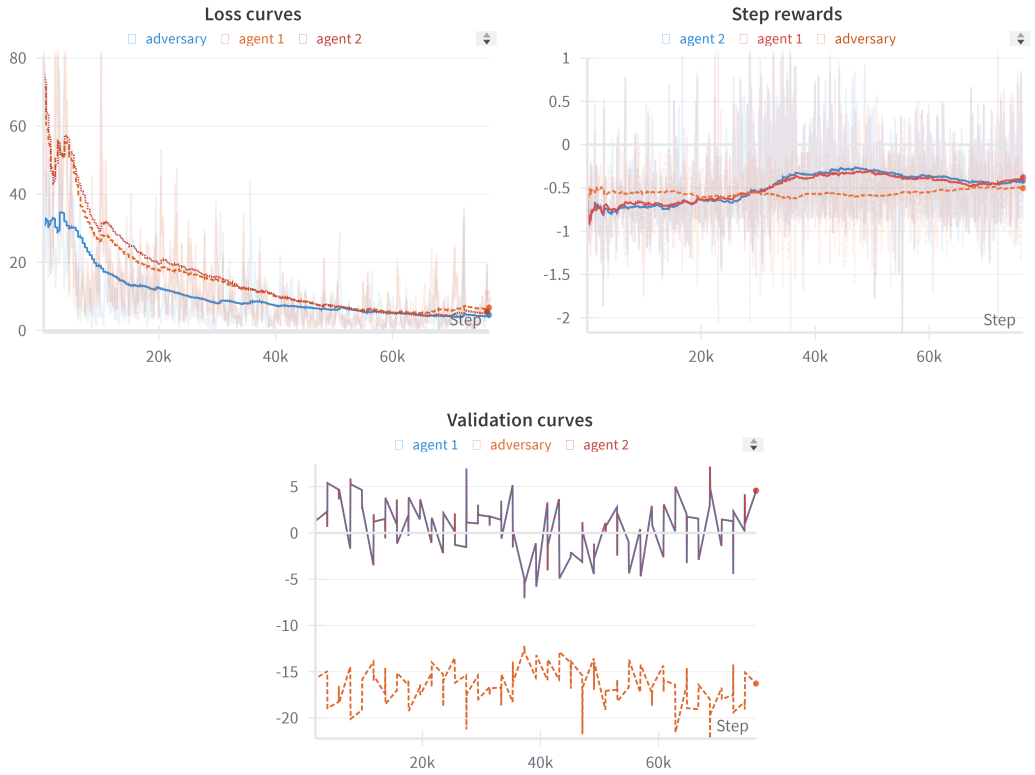

Figure 20: PPO (AEC API) - Pretraining on the Source Task

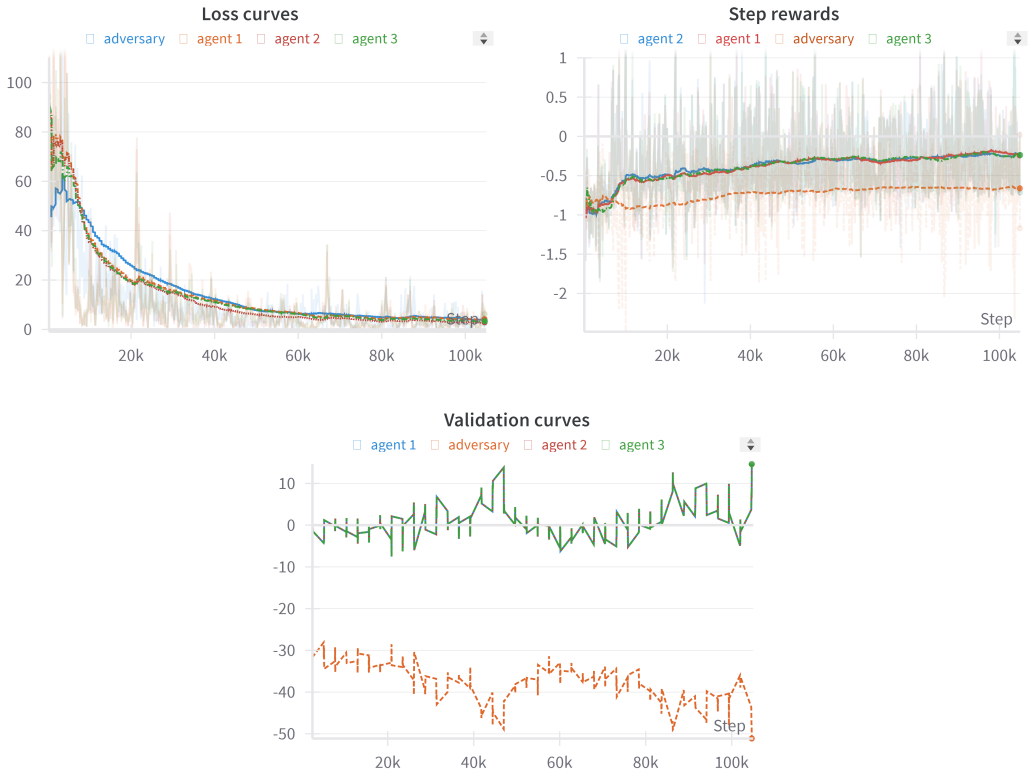

Figure 21: PPO (AEC API) - Target Task Trained from Scratch

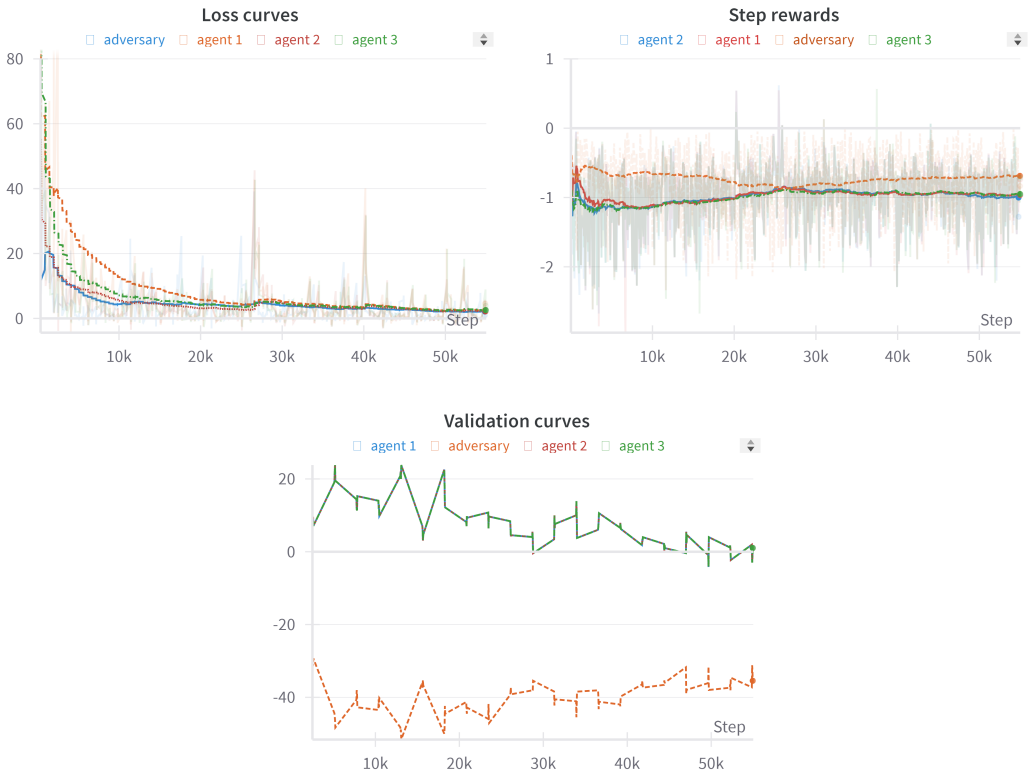

Figure 22: PPO (AEC API) - Target Learned from Source (Coarse-to-Fine Tuning)

Given the larger training time required for the PPO approach compared to the Q-learning technique to achieve similar performance, the implementation on the DFT environment is done using Dueling DDQN. This decision ensures efficient utilization of computational resources while still achieving satisfactory performance in training and convergence. The Dueling DDQN algorithm offers a balance between training speed and effectiveness, making it well-suited for our application.

4.2 Implementation of Dueling DDQN for DFT Environment (AEC API)

The experiments are divided into two approaches based on the complexity of the DFT. In one approach, the source task involves a DFT with fewer basic events compared to the target task, while in the other approach, it's the opposite.

From the graphs, it's evident that the number of training steps needed increases as the complexity of the system increases. This observation aligns with intuition, as more complex systems require more training to learn effective strategies and policies. Additionally, it's noticeable that it's easier for the agent to learn a low-complexity task if it's already trained on a complex task. TL also helps in speeding up the training.

However, it's important to note that hyperparameters play a crucial role in the learning process. Choosing appropriate hyperparameters can significantly impact the training efficiency and final performance of the agents. An interesting observation was obtained when using MSE: the error terms became notably large. This peculiarity can be attributed to the distinctive reward function employed by the environment, where agents receive substantial rewards upon accomplishing their goals. To mitigate the impact of these large error terms and ensure stable training, Huber loss was opted for. This loss function offers robustness against outliers, making it well-suited for scenarios where extreme rewards can lead to disproportionately large errors. By employing Huber loss, smoother and more stable training dynamics can be ensured.

TL has effectively reduced the number of training steps required in the target task. This is evident when comparing the number of games played during training from scratch versus training with knowledge transferred from the source task. Remarkably, to achieve similar performance levels, training with TL capability requires fewer games in both experiments.

4.2.1 DFT with lesser number of basic events as the Source Task

In real-world scenarios, Dynamic Fault Tree (DFT) systems often start with a smaller set of basic events in their initial configuration. As the system evolves and undergoes modifications, additional basic events may be introduced to capture its increasing complexity and dynamic behavior. This scenario is simulated for TL capability, using the Dueling DDQN algorithm. The two agents are initially trained in the source task, which represents the DFT system with fewer basic events. This serves as a foundation before deploying them for further training in the target task, which involves a DFT system with a greater number of basic events.

From Figure 23, it's apparent that both agents reach an optimal state or Nash equilibrium around step 3000. However, beyond this point, the loss curve for the red agent exhibits greater fluctuations, suggesting potential overfitting. Consequently, training for the agents in the source task is halted at this juncture.

Moving to Figure 24, one could observe the learning curves for agents tackling the target task from scratch. Meanwhile, Figure 25 illustrates the learning trajectory of agents benefiting from a pre-trained model. Notably, from these graphs, it's evident that agents with TL capability, trained from the pre-trained model, reach a similar state approximately 5000 steps ahead compared to those trained from scratch.

It's worth mentioning the peculiar shape of the step rewards for the red agent, resembling a ribbon. This phenomenon arises from the substantial bonuses it receives when the system experiences faults,

as per the system model.

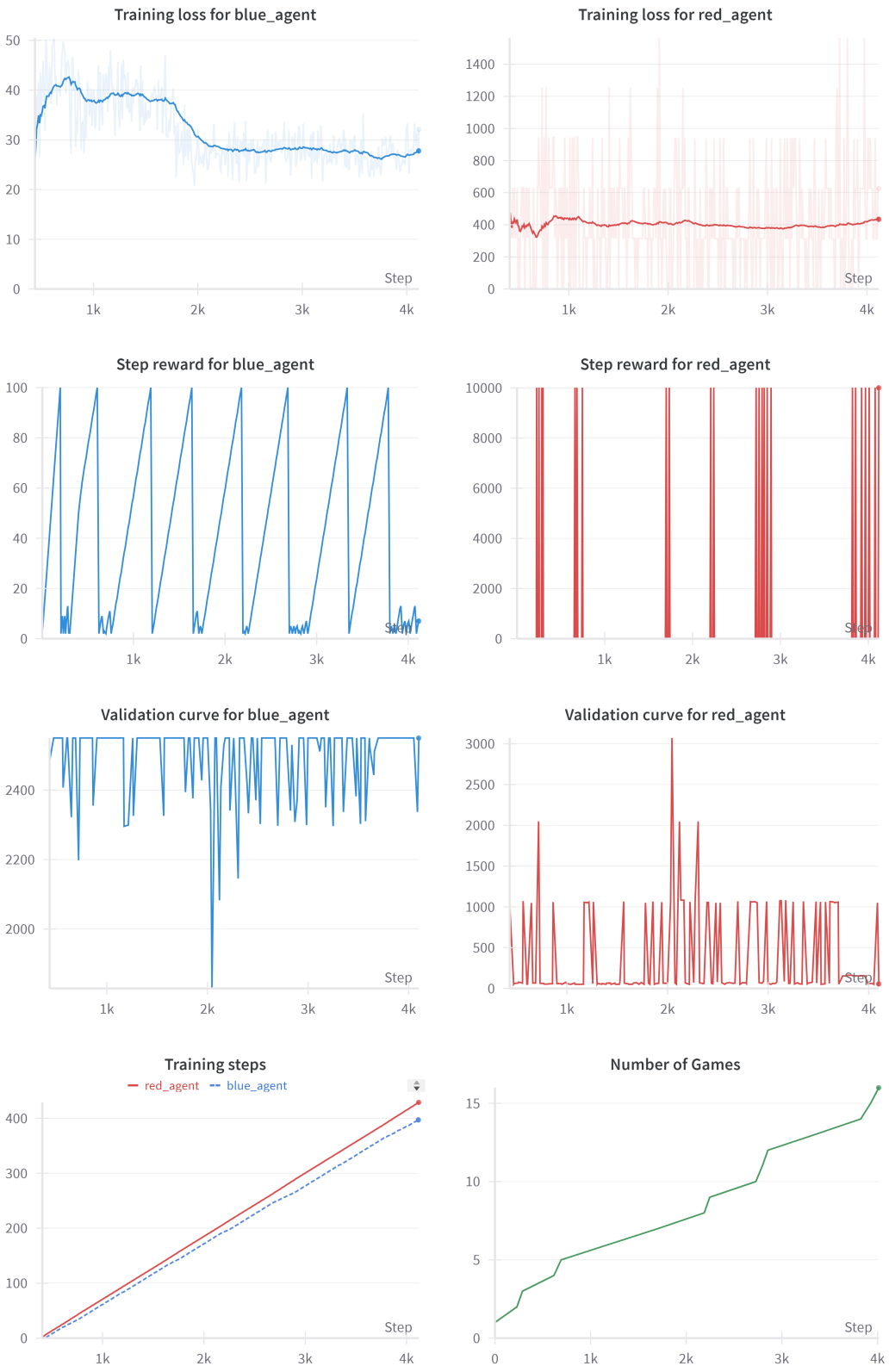

Figure 23: Pretraining on the Source Task (low dimensional DFT)

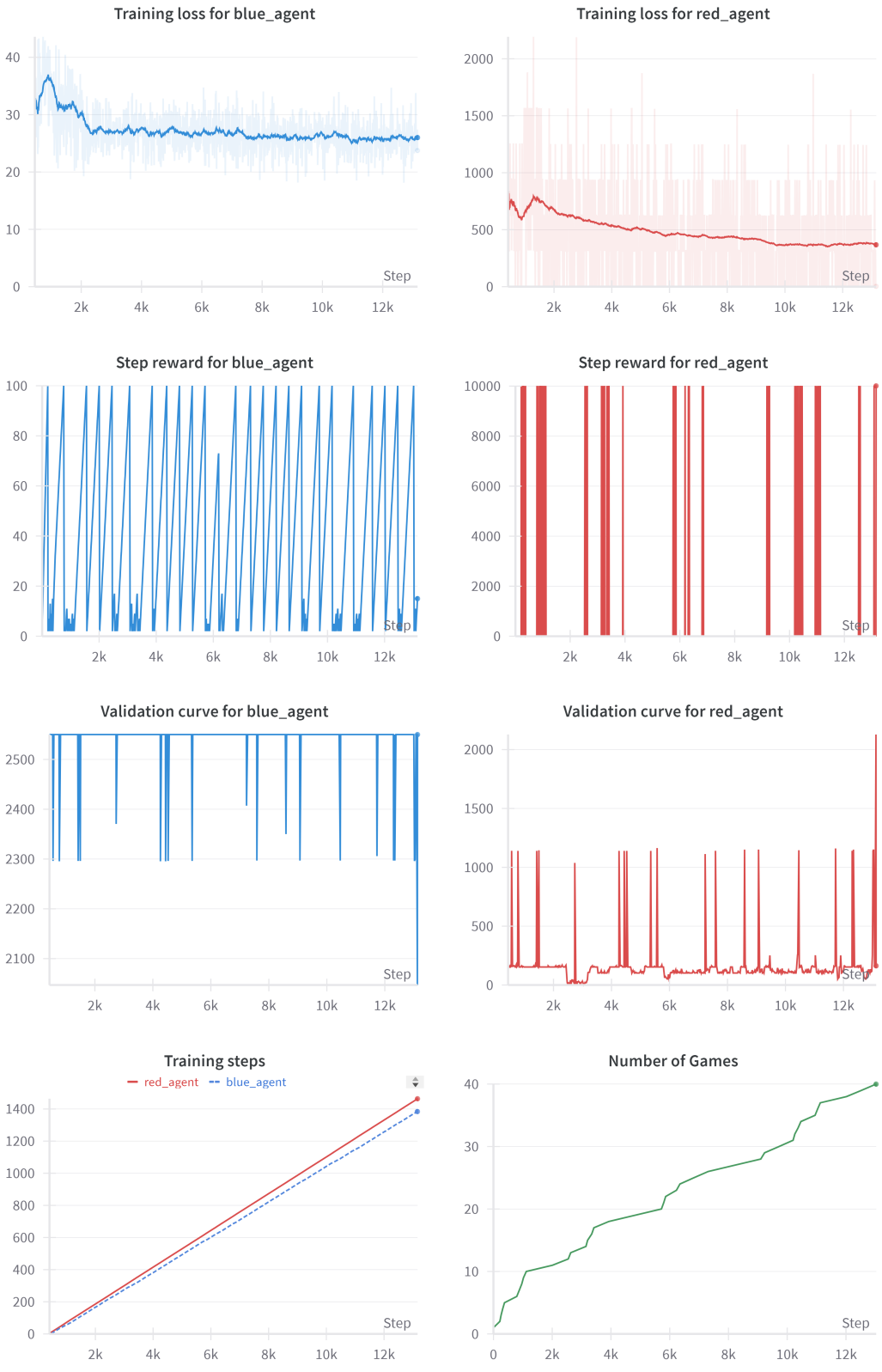

Figure 24: Target Task (high dimensional DFT) trained from Scratch

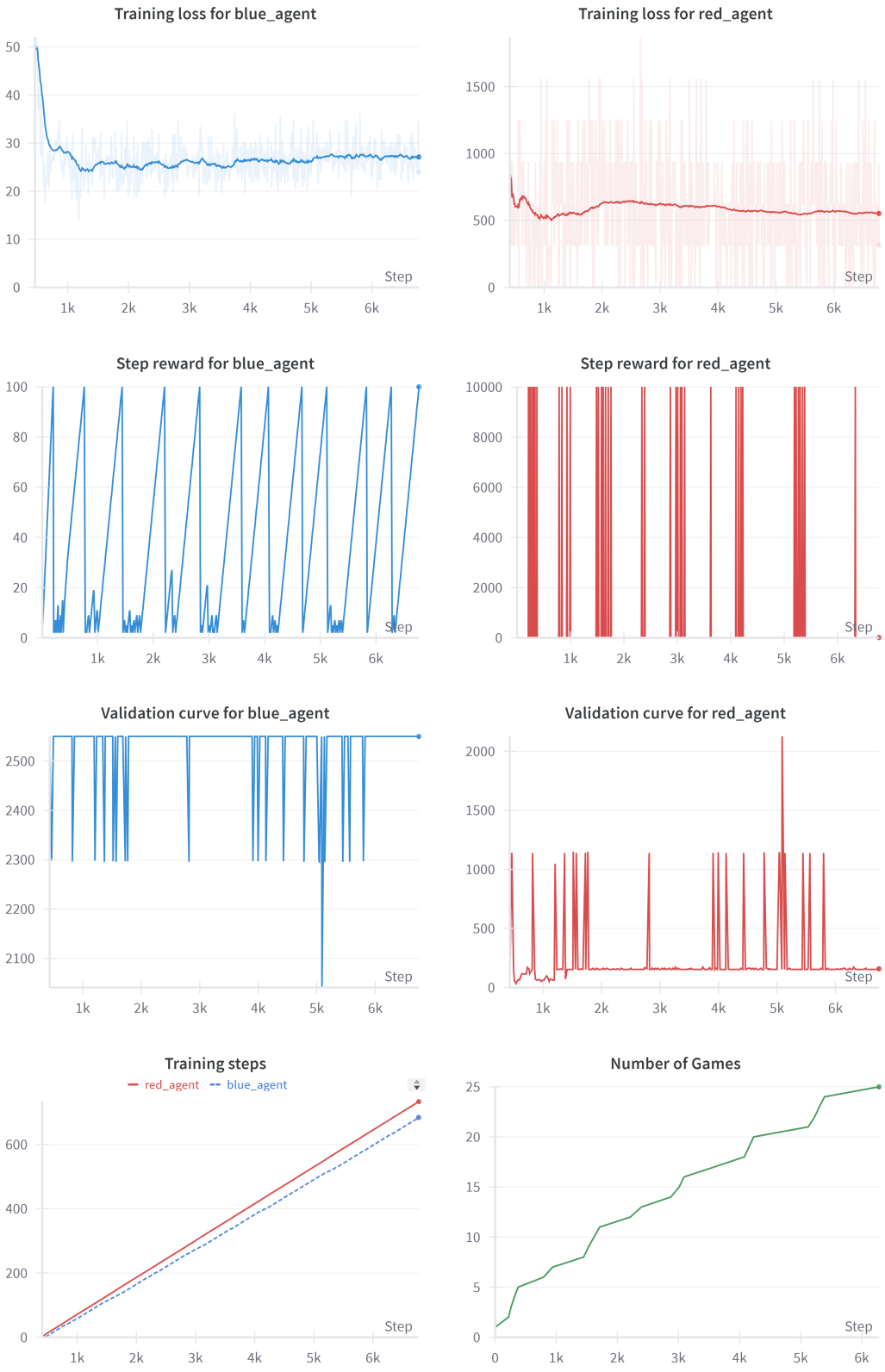

Figure 25: Target Task (high dimensional DFT) trained from Source

4.2.2 DFT with a higher number of basic events as the Source Task

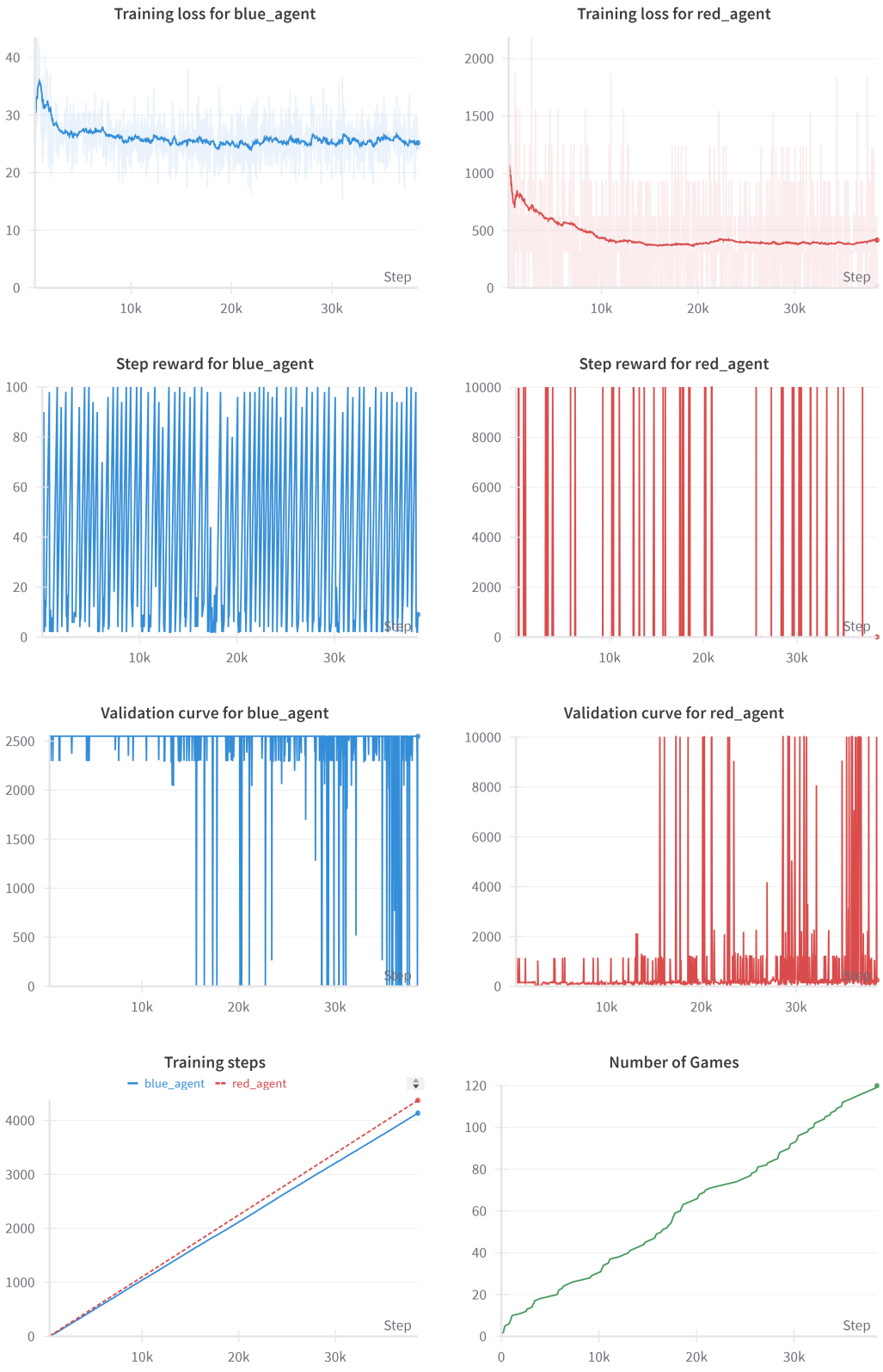

Figure 26: Pretraining on the Source Task (high dimensional DFT)

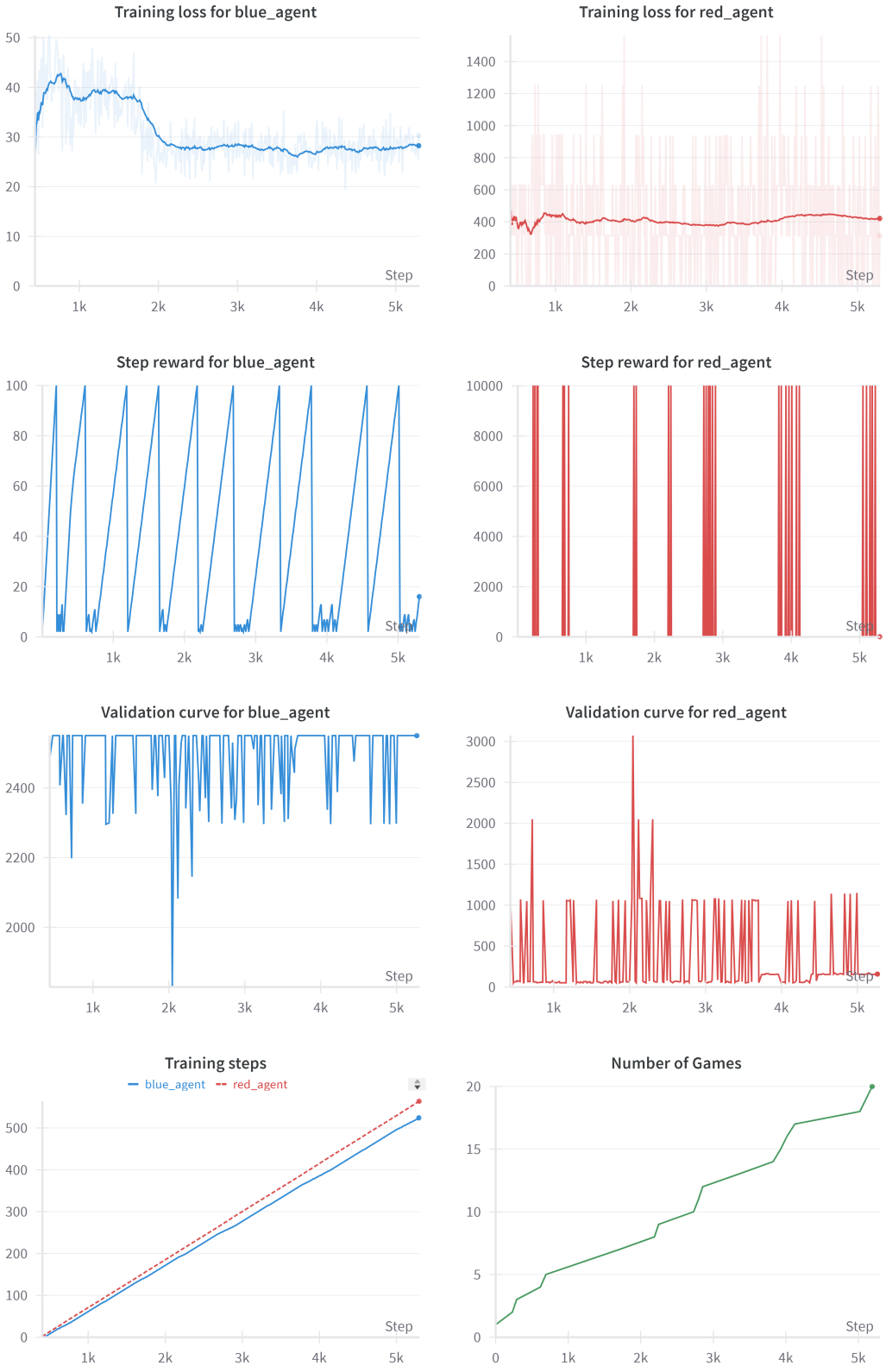

Figure 27: Target Task (low dimensional DFT) trained from Scratch

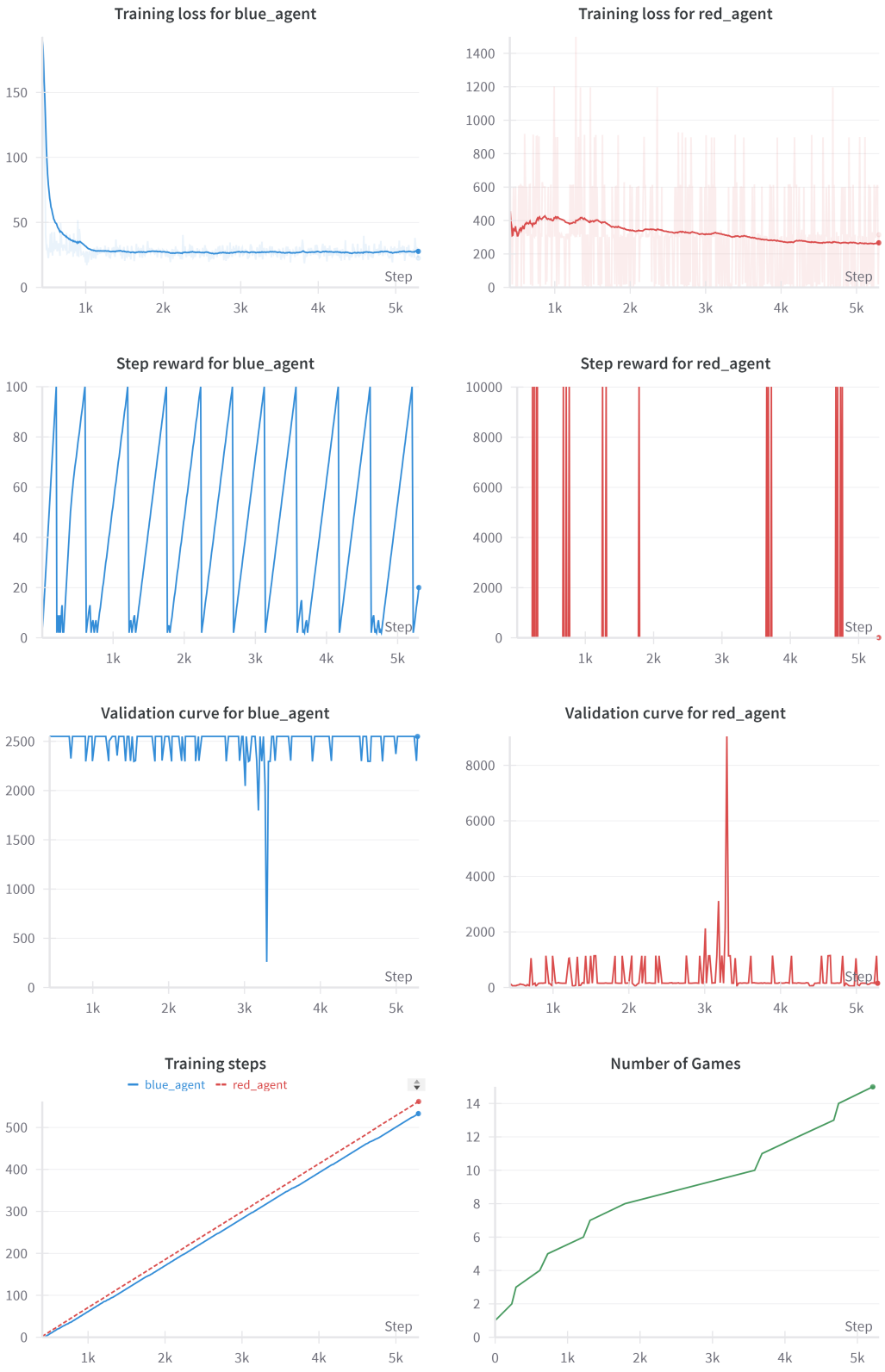

Figure 28: Target Task (low dimensional DFT) trained from Source

This subsection presents the opposite scenario, where the DFT with a higher number of basic events serves as the source task, contrasting the previous subsection. Here also, TL accelerates the training process in the target domain, as evident in Figure 28. Notably, agents reach the optimal

state in approximately 1000 steps, compared to the 3000 steps required for agents trained from scratch, as seen in Figure 27.

5 Conclusion

In conclusion, this research on Adversarial Multi-Agent Transfer Learning (MATL) has unveiled valuable insights into fault-tolerant system design within dynamic environments. Through the seamless integration of neural networks and MARL, this study has demonstrated the transformative potential of TL in significantly enhancing the adaptability and learning efficiency of adversarial agents. Notably, neural networks, and in particular CNNs, proved to be effective representations for agents in MARL scenarios.

The implementation of techniques such as pretraining on a source task and coarse-to-fine tuning has played a pivotal role in facilitating rapid learning and adaptation of agents in new environments. Furthermore, the comparative analysis of two prominent RL algorithms, Q-learning and PPO, within an adversarial setting, has highlighted the effectiveness of TL in improving agent performance.

The exploration of different APIs for the game environment and investigation of their TL perfomance, particularly the AEC API for the DFT environment, suggest promising avenues for further research into developing and harnessing the potential of the Parallel API. Additionally, evaluating the performance of TL using PPO over the DFT environment presents an intriguing area for future investigation.

These findings collectively emphasize the potential of TL to revolutionize fault-tolerant system design by enabling agents to learn more effectively, adapt to new tasks efficiently, and enhance overall system resilience in dynamic and complex environments. TL facilitates the efficient utilization of resources by leveraging preexisting knowledge, thereby reducing the need for extensive training in each new environment.

Moreover, expanding the current idea to include continuous learning over multiple game versions, enabling the network to adapt to larger systems capable of injecting faults and modifying the environment, presents an exciting avenue for future research. Further investigations could also explore more sophisticated reward structures tailored for the DFT system and explore network architectures with fully connected layers or Recurrent Neural Networks (RNNs).

With the continuous advancements in the field, it's foreseeable that new algorithms and methodologies will emerge, further advancing our understanding and capabilities in this domain.

Bibliography

Albrecht, Stefano V., Filippos Christianos and Lukas Schäfer (2024). *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press. URL: https://www.marl-book.com.

Ammar, Haitham Bou and Matthew E. Taylor (2011). 'Common Subspace Transfer for Reinforcement Learning Tasks'. English. In: *Adaptive and Learning Agents*. Lecture Notes in Computer Science. Germany: Springer Verlag, pp. 21–36.

Barreto, André, Diana Borsa et al. (2019). *Transfer in Deep Reinforcement Learning Using Successor Features and Generalised Policy Improvement*. arXiv: 1901.10964 [cs.LG].

Barreto, André, Will Dabney et al. (2018). *Successor Features for Transfer in Reinforcement Learning*. arXiv: 1606.05312 [cs.AI].

Burguillo, Juan C. (2018). 'Game Theory'. In: *Self-organizing Coalitions for Managing Complexity: Agent-based Simulation of Evolutionary Game Theory Models using Dynamic Social Networks for Interdisciplinary Applications*. Cham: Springer International Publishing, pp. 101–135. ISBN: 978-3-319-69898-4. DOI: 10.1007/978-3-319-69898-4_7. URL: https://doi.org/10.1007/978-3-319-69898-4_7.

Chemali, Jessica and Alessandro Lazaric (2015). 'Direct policy iteration with demonstrations'. In: *Proceedings of the 24th International Conference on Artificial Intelligence*. IJCAI'15. Buenos Aires, Argentina: AAAI Press, pp. 3380–3386. ISBN: 9781577357384.

Correia, André and Luís A. Alexandre (2023). *A Survey of Demonstration Learning*. arXiv: 2303.11191 [cs.LG].

Czarnecki, Wojciech Marian et al. (2019). *Distilling Policy Distillation*. arXiv: 1902.02186 [cs.LG].

Da Silva, Felipe Leno and Anna Helena Reali Costa (Jan. 2019). 'A survey on transfer learning for multiagent reinforcement learning systems'. In: *J. Artif. Int. Res.* 64.1, pp. 645–703. ISSN: 1076-9757. DOI: 10.1613/jair.1.11396. URL: https://doi.org/10.1613/jair.1.11396.

Dayan, Peter (1993). 'Improving generalization for temporal difference learning: The successor representation'. In: *Neural Comput.* 5.4, pp. 613–624. ISSN: 0899-7667. DOI: 10.1162/neco.1993.5.4.613. URL: https://doi.org/10.1162/neco.1993.5.4.613.

Devin, Coline et al. (2016). *Learning Modular Neural Network Policies for Multi-Task and Multi-Robot Transfer*. arXiv: 1609.07088 [cs.LG].

Devlin, Sam and Daniel Kudenko (2011). 'Theoretical considerations of potential-based reward shaping for multi-agent systems'. In: *The 10th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '11. Taipei, Taiwan: International Foundation for Autonomous Agents and Multiagent Systems, pp. 225–232. ISBN: 0982657153.

— (2012). 'Dynamic potential-based reward shaping'. In: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems - Volume 1*. AAMAS '12. Valencia, Spain: International Foundation for Autonomous Agents and Multiagent Systems, pp. 433–440. ISBN: 0981738117.

Fang, Bin et al. (Dec. 2019). 'Survey of imitation learning for robotic manipulation'. In: *International Journal of Intelligent Robotics and Applications* 3. DOI: 10.1007/s41315-019-00103-5.

Fernández, Fernando and Manuela Veloso (2006). 'Probabilistic policy reuse in a reinforcement learning agent'. In: *Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems*. AAMAS '06. Hakodate, Japan: Association for Computing Machinery, pp. 720–727. ISBN: 1595933034. DOI: 10.1145/1160633.1160762. URL: https://doi.org/10.1145/1160633.1160762.

Fernando, Chrisantha et al. (2017). *PathNet: Evolution Channels Gradient Descent in Super Neural Networks*. arXiv: 1701.08734 [cs.NE].

Gagniuc, Paul (May 2017). *Markov Chains: From Theory to Implementation and Experimentation*. ISBN: 978-1-119-38755-8. DOI: 10.1002/9781119387596.

Harutyunyan, Anna et al. (Jan. 2015). 'Expressing Arbitrary Reward Functions as Potential-Based Advice'. In.

He, George, Daylen Yang and Kelly Shen (2019). 'Improving DQN Training Routines with Transfer Learning'. In.

Hester, Todd et al. (2017). *Deep Q-learning from Demonstrations*. arXiv: 1704.03732 [cs.AI].

Ho, Jonathan and Stefano Ermon (2016). 'Generative adversarial imitation learning'. In: *Proceedings of the 30th International Conference on Neural Information Processing Systems*. NIPS'16. Barcelona, Spain: Curran Associates Inc., pp. 4572–4580. ISBN: 9781510838819.

Junges, Sebastian et al. (June 2016). 'Uncovering Dynamic Fault Trees'. In: DOI: 10.1109/DSN.2016.35.

Mnih, Volodymyr et al. (2013). 'Playing atari with deep reinforcement learning'. In: *arXiv preprint arXiv:1312.5602*. URL: https://arxiv.org/pdf/1312.5602.pdf.

Mordatch, Igor and Pieter Abbeel (2017). 'Emergence of Grounded Compositional Language in Multi-Agent Populations'. In: *arXiv preprint arXiv:1703.04908*.

Murata, Tadao (Apr. 1989). 'Petri Nets: Properties, Analysis and Applications.' In: *Proceedings of the IEEE* 77.4, pp. 541–580.

Ng, A., Daishi Harada and Stuart J. Russell (1999). 'Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping'. In: *International Conference on Machine Learning*. URL: https://api.semanticscholar.org/CorpusID:5730166.

Parisotto, Emilio, Jimmy Lei Ba and Ruslan Salakhutdinov (2016). *Actor-Mimic: Deep Multitask and Transfer Reinforcement Learning*. arXiv: 1511.06342 [cs.LG].

Rusu, Andrei A., Sergio Gomez Colmenarejo et al. (2016). *Policy Distillation*. arXiv: 1511.06295 [cs.LG].

Rusu, Andrei A., Neil C. Rabinowitz et al. (2022). *Progressive Neural Networks*. arXiv: 1606.04671 [cs.LG].

Schaul, Tom et al. (2015). 'Universal Value Function Approximators'. In: *Proceedings of the 32nd International Conference on Machine Learning*. Ed. by Francis Bach and David Blei. Vol. 37. Proceedings of Machine Learning Research. Lille, France: PMLR, pp. 1312–1320. URL: https://proceedings.mlr.press/v37/schaul15.html.

Schulman, John et al. (2017). *Proximal Policy Optimization Algorithms*. arXiv: 1707.06347 [cs.LG].

Sherstov, Alexander A. and Peter Stone (2005). 'Improving action selection in MDP's via knowledge transfer'. In: *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*. AAAI'05. Pittsburgh, Pennsylvania: AAAI Press, pp. 1024–1029. ISBN: 157735236x.

Shevchenko, Nataliya (Dec. 2020). *An Introduction to Model-Based Systems Engineering (MBSE)*. Carnegie Mellon University, Software Engineering Institute's Insights (blog). Accessed: 2024-Mar-22. URL: https://insights.sei.cmu.edu/blog/introduction-model-based-systems-engineering-mbse/.

Stengel, Bernhard von and Theodore Turocy (Dec. 2003). 'Game Theory'. In: *Encyclopedia of Information Systems* 2. DOI: 10.1016/B0-12-227240-4/00076-9.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. The MIT Press. URL: http://incompleteideas.net/book/the-book-2nd.html.

Taylor, Matthew E., Peter Stone and Yaxin Liu (2007). 'Transfer Learning via Inter-Task Mappings for Temporal Difference Learning'. In: *Journal of Machine Learning Research* 8.1, pp. 2125–2167.

Teh, Yee Whye et al. (2017). *Distral: Robust Multitask Reinforcement Learning*. arXiv: 1707.04175 [cs.LG].

Terry, J. K. et al. (2021). *PettingZoo: Gym for Multi-Agent Reinforcement Learning*. arXiv: 2009.14471 [cs.LG].

Wang, Ziyu et al. (2016). *Dueling Network Architectures for Deep Reinforcement Learning*. arXiv: 1511.06581 [cs.LG].

Wiewiora, Eric, Garrison Cottrell and Charles Elkan (July 2003). 'Principled Methods for Advising Reinforcement Learning Agents'. In.

Zhang, Amy, Harsh Satija and Joelle Pineau (2018). *Decoupling Dynamics and Reward for Transfer Learning*. arXiv: 1804.10689 [cs.LG].

Zhu, Zhuangdi, Kaixiang Lin, Bo Dai et al. (2020). *Learning Sparse Rewarded Tasks from Sub-Optimal Demonstrations*. arXiv: 2004.00530 [cs.LG].

Zhu, Zhuangdi, Kaixiang Lin, Anil K. Jain et al. (2023). *Transfer Learning in Deep Reinforcement Learning: A Survey*. arXiv: 2009.07888 [cs.LG].

Zhuang, Fuzhen et al. (2020). *A Comprehensive Survey on Transfer Learning*. arXiv: 1911.02685 [cs.LG].

Appendix

A Pseudo codes - Dueling DDQN

A.1 Training Loop (Parallel API)

```
for each game in range(num_games) do
    for each epoch in range(epochs) do
        Reset environment and get the initial state and information
        while game is not done do
            if any agent terminated or truncated then
                End the game
                Log total episode rewards during training
            else
                Increment total_steps
                for each agent do
                    Select action using agent model
                end for
                Take joint actions in the environment
                for each agent do
                    Update buffers and rewards
                    if buffer has enough experiences then
                        Train the agent's model
                        Update exploration noise
                        Log training loss
                        Increment total_training_steps
                    end if
                end for
                if enough training steps have passed then
                    Evaluate policy
                    Log evaluation score
                    Save models if necessary
                end if
            end if
        end while
    end for
end for
```

A.2 Training Loop (AEC API)

```
for each game in range(num_games) do
    for each epoch in range(epochs) do
        Reset environment
        for each agent in env do
            Increment total_steps for agent
            Get observation, reward, termination, and truncation from environment
            if not the starting point then
                Update next state, reward, and done indicator
                Add experience to replay buffer
            end if
            if not terminated or truncated then
                Get action from agent model
                Update current state and action
            else
                Set the agent's action as None
            end if
            Step through environment with action
```

```
            if enough experiences in replay buffer then
                Train the agent's model
                Update exploration noise
                Log training loss
                Increment total_training_steps for agent
            end if
            if enough training steps have passed then
                Evaluate policy
                Log evaluation score
                Save models if necessary
            end if
        end for
    end for
end for
```

A.3 Action Selection

```
if not evaluate then
    ϵ ← self.exp_noise
    if random number < ϵ then
        Choose a random action
    else
        Forward pass state through Q-network to get Q-values
        Choose action with highest Q-value
    end if
else
    Choose action with highest Q-value deterministically
end if
return chosen action
```

A.4 Network Training

```
Sample a batch from the replay buffer
for each sample in the batch do
    Compute target Q value
    Compute current Q estimates
end for
Compute MSE loss between current Q estimate and target Q
Compute gradients of the Q-loss with respect to the model parameters
Update model parameters using the computed gradients
Update the target network if necessary
Freeze target network parameters to prevent backpropagation
return Q loss
```

A.5 Policy Evaluation (Parallel API)

```
for each episode in range(num_episodes) do
    done ← False
    Reset eval_env and get initial observations and info
    while not done do
        for each agent in eval_env do
            Get state from observations
            Select action using agent's model and state
        end for
        Take joint action in eval_env
        Get next_observations, rewards, terminations, truncations, and info
        Check if any agent has terminated or truncated
```

```
        Update done
        for each agent in eval_env do
            Add rewards[agent] to total_reward[agent]
        end for
        Update observations with next_observations
    end while
end for
for each agent in eval_env do
    Compute average reward for agent over num_episodes
    Store average reward in total_reward[agent]
end for
return total_reward
```

A.6 Policy Evaluation (AEC API)

```
for each episode in range(num_episodes) do
    Reset eval_env
    for each agent in eval_env do
        Get observation, reward, termination, and truncation from the environment
        if terminated or truncated then
            Do nothing
        else
            Update agent's total rewards
            Get action from the agent model
            Step through the environment with action
        end if
    end for
end for
for each agent in eval_env do
    Compute average reward over num_episodes
    Store average reward in total_reward[agent]
end for
return total_reward
```

A.7 Model Transfer

```
for each agent in the target_environment do
    Get the new observation and action dimensions for the agent
    Load source model parameters from a saved file
    Get the observation and action dimensions for the transferred model
    Modify input and output layers of the model accordingly:
      - Adjust input layer to match new observation dimension
      - Adjust output layer to match the new action dimension
    Initialize optimizer for modified model for coarse tuning
end for
```

B Pseudo codes - PPO

B.1 Training Loop (AEC API)

```
for each game in range(num_games) do
    Reset environment
    for each agent in env do
        Initialize starting_point[agent] as True
        Set done[agent] to 0
    end for
    for each agent in env do
        Increment total_steps for agent
        Get observation, reward, termination, and truncation from environment
        if not starting_point[agent] then
            Remember observation, reward, and termination information
            Log rewards for the agent
            if termination or truncation then
                Set done[agent] to 1
            else
                Set done[agent] to 0
                Store current state, action, probability, value, reward, and done status
            end if
            Set starting_point[agent] to True
        end if
        if termination or truncation then
            Set action to None
        else
            Get agent model
            Remember current observation
            Choose action using the model
            Remember chosen action, probability, and value
            Set starting_point[agent] to False
        end if
        Perform action in the environment
        Get agent model
        if enough total_steps for the agent has passed then
            if transfer_train is True then
                if coarse_tuner_counter for the agent is less than 50 then
                    Perform coarse tuning
                else
                    Perform fine tuning
                end if
            end if
            Train the agent model
            Increment total_training_steps for the agent
            Log training loss for the agent
            if enough training steps have passed then
                Save the trained model if necessary
                Evaluate the policy
            end if
        end if
    end for
end for
```

B.2 Action Selection

```
Obtain probability distribution over actions from the actor network for the given state
Sample action from the probability distribution
```

3: Calculate log probability of the chosen action from the distribution
4: Predict value of the state using the critic network
5: **Return** action, probability, value

B.3 Network Training

1: **for** each epoch in range(epochs) **do**
2: Generate batches of experiences from memory
3: Calculate advantage using Generalized Advantage Estimation for each sample in the batch
4: **for** each batch in the generated batches **do**
5: Forward pass through the actor network to get action distribution
6: Forward pass through critic network to get critic values
7: Compute new probabilities from action distribution
8: Compute probability ratios and weighted probabilities
9: Compute clipped weighted probabilities
10: Compute actor loss as the minimum of weighted and clipped weighted probabilities
11: Compute critic loss as the mean squared error between returns and critic values
12: Compute total loss as the sum of actor loss and 0.5 times critic loss
13: Log losses
14: Update actor and critic networks by backpropagating the total loss
15: **end for**
16: **end for**
17: Clear memory after all epochs are completed

Declaration of Compliance

I hereby declare to have written this work independently and to have respected in its preparation the relevant provisions, in particular those corresponding to the copyright protection of external materials. Whenever external materials (such as images, drawings, text passages) are used in this work, I declare that these materials are referenced accordingly (e.g. quote, source) and, whenever necessary, consent from the author to use such materials in my work has been obtained.

Date and Signature: