|  |  |  |
| --- | --- | --- |
| **FA 3613** | | |
| Adversarial Multi-Agent Transfer Learning | | |
| **Amal Chulliyat Jose** | | |
|  | | |
| **Conception** | | |
|  | Examiner: | Jun.-Prof. Dr.-Ing. Andrey Morozov |
|  | Supervisor: | Joachim Grimstad, M.Sc.Ing |
| Begin: 16.10.2023 | | End: 01.04.2024 |
|  | |  |

**Document Version Management**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Version | Author | QA | Date | Status | Changes |
| 0.1 | ACJ |  | 20.12.23 | in progress | Creation |
|  |  |  |  |  |  |

# Table of Contents

1 Table of Contents 2

2 Description of the Conception 3

2.1 Overview 3

2.2 Learning Approach 3

2.3 Learning Algorithm 4

2.4 Evaluation Metrics 6

2.5 Implementation Plan 7

2.6 Potential Challenges 7

3 Summary 7

# Description of the Conception

## Overview

Dynamic Fault Trees (DFT) are an extension of Static Fault Trees to model dynamic systems, where the conditions change over time, such as repairs, maintenance, or system reconfiguration. This project aims to train and develop neural networks for two adversarial agents who activate/deactivate the root-level events of a DFT to cause and avert a system failure, respectively. The training is done over different fault scenarios such that the agent that has been trained to create faults will autonomously identify vulnerabilities or weak points in the system and attempt to exploit them, while the agent trained to avert faults will continuously monitor the system for any signs of instability or potential fault conditions and will take proactive measures to mitigate the risks.

When we feed a DFT into this network, the two agents compete with each other in this environment like a game and output their scores. Based on the scores, some modifications are brought into the DFT, and the new DFT is again fed into the neural networks. But this time, they should use the knowledge from the previous environment and fine-tune it to the new environment, which is known as Transfer Learning.

## Learning Approach

The transfer learning algorithm to be adopted is based on the double deep Q-learning networks (DDQN), where convolutional neural networks are used 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 fine-tuned for the new environment, thus enabling better and faster learning from the prior lessons.

The game in our case is played between two adversarial agents; red and blue. These agents interact with an environment that is a Dynamic Fault Tree, in such a way that the red agent tries to create a system failure while the blue agent tries to avert it. The Q-networks training is carried out until the respective Q-values converge to a stable value. 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. Convergence of Q-values signifies that the networks have learned to approximate the Q-values accurately but to guarantee the satisfaction of the Nash equilibrium, some more points need to be considered.

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 to avoid agents getting stuck in an unstable state. For this, we use an epsilon-greedy method, where the agent selects the action with the highest Q-value with a probability of and selects a random action with a probability of . The value of is gradually decreased over time to encourage the agent to exploit its learned Q-values more often as training progresses. Along with this, we also use an experience replay buffer to encourage exploration and avoid catastrophic forgetting. Experience replay involves storing the agent's experiences in a replay buffer and randomly sampling a batch of experiences from the buffer to train the agent's Q-network.

The transfer learning process can be summarised as follows:

* Pretrain the neural networks on the given DFT environment to find the acceptable estimation of the respective Q-values. This helps the agents learn basic strategies.
* Simulating a game between the adversarial agents over this given DFT to output the final scores. The current DFT 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 to that of the current environment.
* Fine-tuning the parameters for the new environment.

## Learning Algorithm

As mentioned, a variant of the Deep Q-learning RL approach, Double Deep Q-learning Network(DDQN) combined with the concepts of Dueling DQN, is used here. 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 these together help to mitigate overestimation bias and to 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: Evaluates the advantage of each action relative to others for a given state.
* Merging layer: Combines the value and advantage streams to reconstruct the Q-values.

Thus, the key components of the Dueling DDQN 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 samples of mini-batches 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 a certain number of 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.
* 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 predicted Q-values and the target Q-values obtained from the frozen target network.

The basic pseudo-code for the training loop is as given below:

Initialize the main and the target networks for both the agents with random weights.

Initialize the replay buffer for both agents.

Initialize exploration parameter ()

Initialize hyperparameters (learning\_rate, discount\_factor, batch\_size, target\_update\_freq etc.)

for episode in range(num\_episodes):

Reset environment for each episode

done = False

while not done:

# Agents select action using epsilon-greedy strategy

action\_red = epsilon\_greedy\_action\_selection(state, Q\_main\_red, )

action\_blue = epsilon\_greedy\_action\_selection(state, Q\_main\_blue, )

# Perform actions in the environment

next\_state, reward, done = env.step(action\_red, action\_blue)

# Store experiences in replay memory for both agents

replay\_memory\_red.store\_transition(state, action\_red, reward['red'], next\_state, done)

replay\_memory\_blue.store\_transition(state, action\_blue, reward['blue'], next\_state, done)

# Sample mini-batch from replay memory for both agents

batch\_red = replay\_memory\_red.sample\_batch()

batch\_blue = replay\_memory\_blue.sample\_batch()

# Train the agents

loss\_red = train\_agent(Q\_main\_red, Q\_target\_red, batch\_red)

loss\_blue = train\_agent(Q\_main\_blue, Q\_target\_blue, batch\_blue)

# Update state

state = next\_state

# Update target networks periodically

if episode % target\_update\_freq == 0:

update\_target\_network(Q\_main\_red, Q\_target\_red)

update\_target\_network(Q\_main\_blue, Q\_target\_blue)

# Decay epsilon for exploration-exploitation trade-off

decay\_epsilon()

# Check for convergence or termination criteria

if episode > min\_episodes\_to\_converge:

if check\_convergence():

break

During the pretraining on the first DFT, the agents learn to recognize the game state and take actions to maximize their rewards. Following that a game is simulated between the trained agents to observe how they perform in the environment and to evaluate their learned policy. The pseudo-code for the game simulation is given below:

# Assuming that Q\_main\_red, Q\_main\_blue are trained models

state = env.reset()

done = False

while not done:

# Agents select action using the learned policy

action\_red = Q\_main\_red.predict(state)

action\_blue = Q\_main\_blue.predict(state)

# Interact with the environment

next\_state, reward, done= env.step((action\_red, action\_blue))

\*

# Update state

state = next\_state

To transfer the knowledge to another environment, the convolutional layers of the neural networks are copied and fine-tuned as already mentioned. Its pseudo-code is given below:

# Initialize target network with source network weights

target\_network\_red = clone\_network(source\_network\_red)

target\_network\_blue = clone\_network(source\_network\_blue)

# Fine-tune the target network

for episode in range(num\_episodes):

# ... Existing code for training loop ...

## Evaluation Metrics

* Average Reward: The average reward obtained by the agent over a certain number of episodes. It reflects how well the agent performed in the environment.
* Learning Curve: Plotting the average rewards over episodes can display how the agent's performance improves with training. This curve helps visualize the learning progress.
* Convergence: Convergence refers to the stabilization of the learning process. It can be measured by observing changes in the Q-values over episodes.
* Time Taken for Training: Assessing the time or number of episodes required for the agent to reach a certain level of performance or convergence.
* Exploration Rate (): Tracking the exploration rate over episodes can help visualize how the agent transitions from random exploration to more refined exploitation of the learned knowledge. The exploration rate usually starts at a high value during the initial episodes of training and is reduced over time. Higher values encourage the agent to explore various actions while the lower values encourage it to exploit the actions that it has found to be more rewarding based on its experience. If it decreases too quickly, the agent might exploit suboptimal strategies. If it decreases too slowly, it might explore excessively and take more random actions, hindering the learning process. Thus a right balance is needed.

## Implementation Plan

* Software and hardware requirements:
  + Software:

Deep learning framework (PyTorch), library to create the environment (PettingZoo), data processing, and visualization tools (NumPy, Pandas, Matplotlib).

* + Hardware:

GPU

* Data Collection:

Collect data related to the Dynamic Fault Trees (DFT).

* Training Procedure:
  + Environment Setup:

Create an environment where the agents will interact. Implement the reward system and action space suitable for adversarial training.

* + Network Architecture:

Define the architecture for the Dueling Deep Q-Network (DDQN). Set up the main and target networks with appropriate heads and merging layers. Configure the loss function, optimizer, etc.

* + Training:

Train the agents using the adversarial approach. Use techniques like experience replay, epsilon-greedy strategy, and convergence checks to guide the training process. Monitor the learning curves and convergence of the agents' policies.

* + Evaluation:

Measure the performance metrics.

* + Transfer Learning:

Implement transfer learning to adapt the agents to the modified DFT. Fine-tune the network parameters on the new environment while reusing knowledge from the previous training.

* + Hardware Considerations:

Optimize the code for parallel processing on GPU.

* Hyperparameter Tuning:

Adjust learning rates, batch sizes, exploration-exploitation trade-offs, etc to optimize the network's performance.

## Potential Challenges

* Exploration-Exploitation Trade-off: Balancing exploration for discovering new strategies and exploitation of learned policies to maximize rewards can be challenging.
* Hyperparameter Selection: Select suitable hyperparameters for network architecture, learning rates, exploration strategies, etc.
* Overfitting: Agents may overfit to specific scenarios.
* Reward function: Designing appropriate reward structures that encourage learning the desired behavior can be challenging

# Summary

This concept involves training two adversarial agents, Red and Blue, within a Dynamic Fault Tree (DFT) environment using a Double Deep Q-learning Network (DDQN) with a dueling architecture. The objective is to induce and avert system failures, respectively, creating a competitive game-like scenario. Transfer learning between environments improves the learning process by fine-tuning knowledge gained from the prior environment. Reward shaping and epsilon-greedy exploration strategy guide the learning process, while experience replay and fixed Q-targets contribute to stabilization and avoidance of overestimation biases. The implementation plan provides a comprehensive framework for training the agents apart from the software and hardware requirements.