Institute of Computer Science IV

University of Bonn

Bachelor of Science in Computer Science

Comparing Detection Mechanisms for Adversarial Attacks within Reinforcement Learning based Tactical Networks

Bachelor's Thesis

Submitted by

Luca Liberto

Matriculation Number: 3204178

Email: s6lulibe@uni-bonn.de

Examiner: Prof. Dr. Meier[1] and Dr. Paulo H. L. Rettore[2]

Supervisor: Johannes Loevenich[2]

[1]Head of Institute Computer Science IV, University of Bonn, Germany
[1]Head of Institute, Fraunhofer FKIE, Wachtberg, Germany
Email: mm@cs.uni-bonn.de

[2]Research Scientists, Fraunhofer FKIE, Bad Godesberg, Germany
Email: [johannes.loevenich, paulo.lopes.rettore] @fkie.fraunhofer.de

In collaboration with the Fraunhofer Institute for Communication, Information Processing and Ergonomics (FKIE), Bonn, Germany

February 14, 2023

1 Abstract

The ever growing field of artificial intelligence, especially Reinforcement Learning (RL), has led to many possible deployments in various aspects of society, industry, and military applications, such as establishing radio communication through tactical networks. However, these deployments in high-risk scenarios often raise questions about security and the vulnerability of these systems. It has already been demonstrated that a tactical network trained with RL is susceptible to attacks, such as using adversarial attacks, which can reduce its performance and prevent the network from achieving its goal of establishing reliable and robust communication. As a result, the rise of adversarial attacks in tactical networks trained with RL has raised concerns about the security of these systems. Detecting such attacks is critical, as tactical networks cannot perceive them on their own. Therefore, this bachelor thesis focuses on the detection of attacks, comparing and presenting two prominent detection mechanisms. The results of this thesis demonstrate the effectiveness and suitability of these detection mechanisms in tactical network environments, as well as their general positive and negative aspects.

2 Acknowledgement

First, I would like to express my gratitude to Prof. Dr. Michael Meier and Dr. Paulo H.L. Rettore for taking their time to examine my bachelor thesis. A special thanks goes out to my supervisor Johannes Loevenich for the foundational works that lead to the idea of this bachelor thesis. Moreover, I would like to thank my fellow bachelor colleagues namely, Jonas Bode, Tobias Hürten and Florian Spelter for the group effort, that resulted in the associated paper.

Contents

3 Introduction

Due to the high dynamics of Mobile Ad-Hoc Networks (MANETs) it finds a variety of usages and one such composition are tactical networks, which consists of a number of independent nodes that form a highly dynamic and adaptive network. These networks have a strong demand of security and reliability, especially within cases of critical communication scenarios such as military operations. Since the network is not centralized, each node (e.g radios, vehicles, mobile devices etc.) contributes to the overall security and reliability of the network and bears a responsibility to keep the network stable. Over the years MANETs have become more and more integrated in nowadays usages which in turn raises the question on how these networks can be made more efficient and robust. One major improvement in this area has been shown by utilizing Reinforcement Learning (RL) routing algorithms seen in [1] and [2], where the authors have conducted several experiments with RL - Routing approaches that resulted in an improvement of robustness and efficiency without human intervention.

Therefore, one can conclude that Machine Learning approaches are becoming more of a standard norm to solve these routing problems and in addition improve the network. However, with the ever increasing integration of RL - Routing Protocols in MANETs one can not overlook the simultaneous development of attacking procedures that focus solely on impairing the performance of RL - training methods. These so called *adversarial attacks* aim to confuse the functionality of a RL - Method without directly influencing the inner workings. Furthermore, these attacks can handle their tasks without the target RL - Agent knowing that it is being attacked. In [3] experiments were conducted with two novel attack methods on a tactical network environment. The results have shown that the attacks were able to diminish the performance drastically which begs the question on what kind of impact it could have on a real-world scenario like autonomous vehicles or radio communication.

There have been several proposals to counteract these attacks in various ways [4] however in this study, the focus will mainly lie on how to *detect* adversarial attacks and if these attacks can even be detected at all. These mechanisms are called *adversarial detections* and have a variety of approaches on how to achieve the task of detecting adversarial attacks. For this study a suite of different detection mechanisms will be presented in theoretical and practical manner on an attacking scenario within a tactical network and a comparison will be conducted to highlight the key differences and performances.

The sections for this study will be divided in the following way. In section 3 the necessary theoretical preliminaries will be established followed by the actual problem, that is the main discussion of this study. In section 4 the different detection methods will be presented to give a general understanding of their differences and inner workings in addition the practical steps will also be presented. Furthermore, an example on how detection methods itself can be deceived and made obsolete. In section 5 the results will be presented in a comparative manner to highlight the performance differences. This study ends with the conclusion in section 6 with a general overview and a summary.

4 Problem

4.1 Preliminaries

To understand the main task and purpose of adversarial detections one must consider its foundational structure first and why these detection mechanisms are necessary to consider, which is the purpose of this section. The focus will be on three vital preliminaries, that will establish a better understanding for the overall approach and the context of this work.
First, the basic foundations of Reinforcement Learning have to be considered, since the network will work autonomously without human intervention to achieve maximal reliability and the best results.
Next, a delve into adversarial attacks will be conducted to convey the danger and vulnerabilities in Reinforcement Learning Models. The general approach of these attacks will be presented but also some example novel attack methods to gain a better understanding of the varieties of attacks.
Finally, a description of how a tactical network functions and a look into Mobile Ad-Hoc Networks dynamics will be shown, since it is the main environment on which our problem focuses on.

4.1.1 Concepts of Reinforcement Learning: Motivation

Reinforcement Learning is a concept of Machine Learning with the unique attribute of not taking any human data into a training process, while still aiming for the best possible results at the end. The agent, which is the learning component of the training process, must acquire the knowledge of the given environment for itself by traversing through the environments surroundings and using the available actions. This creates a trial-and-error process, in which the agent achieves bad results at first due to the lack of knowledge and provided information on its surroundings and actions but gradually improves its performance, as it learns from past mistakes and results. In some ways, one could say that the learning process is comparable of that of a human, when confronted with an unknown environment and a given problem. Furthermore, if the agent reaches a peak in its learning process the results are quite often much more superior than other machine learning concepts or human results, which shows the effectiveness and the potential of an RL - Approach.
The most prominent example for this in recent times, was the milestone of AlphaGo Zero [5] for the board Game *Go*. An AI was trained on the principles of Reinforcement Learning without providing it the possible game actions or other necessary information that one usually needs to understand the game, by playing against itself but the lack of this knowledge was an integral part as to why it outperformed its predecessors in a short timespan.
Not only did AlphaGo Zero maximized its performance and became unbeatable in just over a month, it also developed new playing patterns that were not known before, which in itself is a historic achievement considering the age of this board game.
This showcases how powerful Reinforcement Learning can be when utilized correctly and not only in games, but there have also been various proposals and implementations in other fields such as robotics, autonomous driving, industrial areas and other different areas in society.

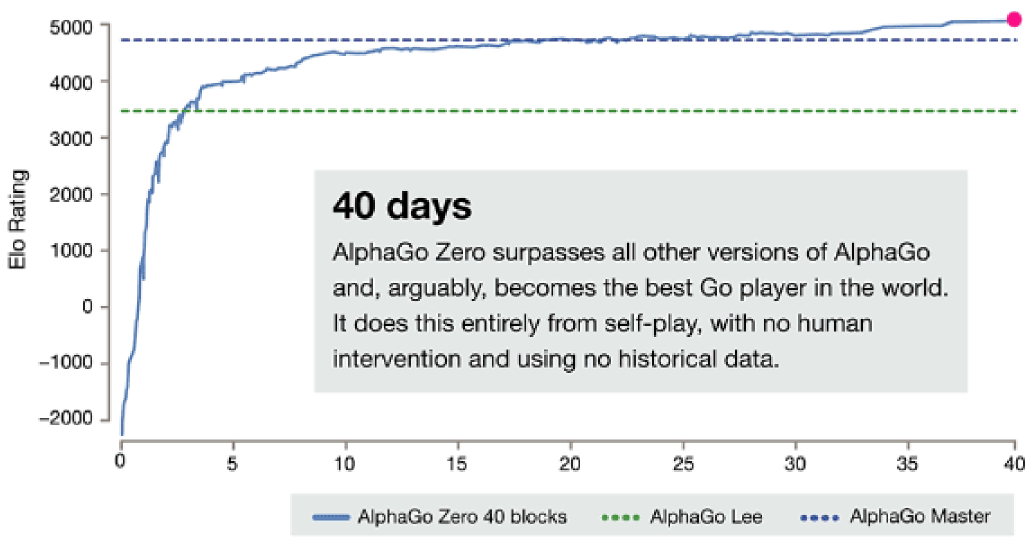


Figure 1. from [6] a depiction of AlphaGo's progress

4.1.2 Concepts of Reinforcement Learning: Basics

Reinforcement Learning in its core essence follows a very basic principle that is applied in every implemented RL-algorithm and there are a few fundamental aspects that build the basic structure of a learning model. The simplified depiction in Fig.2 describes the basic interaction of an agent that can take certain actions $a \in A$ within an **observed** state $s \in S$ or s_t of an environment, whereas t simply denotes the *timestep*, that points out which state is currently being considered. After the action is performed, the environment will in turn give the agent feedback in form of a **reward** $r \in R$ with regard on how the agent has performed. The ultimate goal of an agent is to maximize the possible obtainable rewards, which are returned in form of a numerical value that are differently defined for every environment.

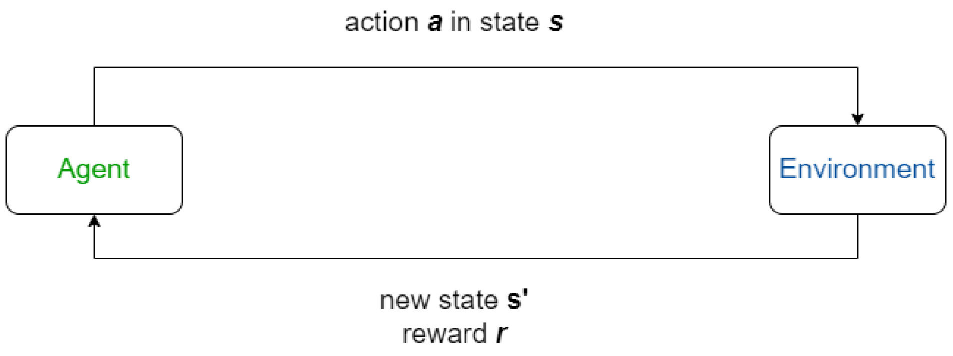


Figure 2. Basic interaction in RL

Furthermore, a new state $s' \in S$ or s_{t+1} will be returned where the agent will advance to and repeat the steps of performing an action. Generally speaking, a learning process can be

divided into *model-free learning*, which means the agent does not take any prior knowledge of the environment (e.g game rules, actions etc.) and learns the environmental properties on its own. On the contrary, a learning process can also be *model-based* and takes information of the environment into its learning process meaning, it learns the model explicitly or the model is given prior to the training process. Furthermore, the transition is influenced by the agent's behaviour, which is called a ***policy*** π that can additionally divided into *deterministic* $\pi(s) = a$ that maps an action to the current state and *stochastic* $\pi(a|s)$ in which the probability of choosing an action a is determined on the condition of the current state. Both of these definitions are vital on how the agent transitions to the next state. These terms will be expanded later on as it goes into more advanced fields of RL.
It is to be noted that the transition between s_t to s_{t+1} in an environment is not definitive and rather calculated with a *transition function*

$$P(s'|s,a) = \Sigma_{r\in R} P(s',r|s,a) \tag{1}$$

This function basically describes the probability of transitioning to a certain next state, while considering the taken action in the current state s. On top of that, a reward function[1] can be derived to calculate an *expected* numerical value for the returned reward r also with consideration of the action in the current state

$$\begin{aligned} R(r|s,a) &= \mathbf{E}[r|s,a] \\ r(s,a) &\doteq \mathbf{E}[R_t = r|S_{t-1} = s, A_{t-1} = a] \end{aligned} \tag{2}$$

This reward function is more formalized and elaborated later on, when discussing *Value functions*. As mentioned at the beginning of the section, the learning process can be either model-free or model-based. If we put these two newly introduced functions into perspective, the general approach is that a model-based process is either given the transition function and the reward function or it is not given but the agent's focus in the learning process is to acquire them (e.g rules on how to win the game). On the other hand, if the agent learns without a model, then it does not focus on learning these functions and the learning process is only consisted of memorization of well rewarded states.

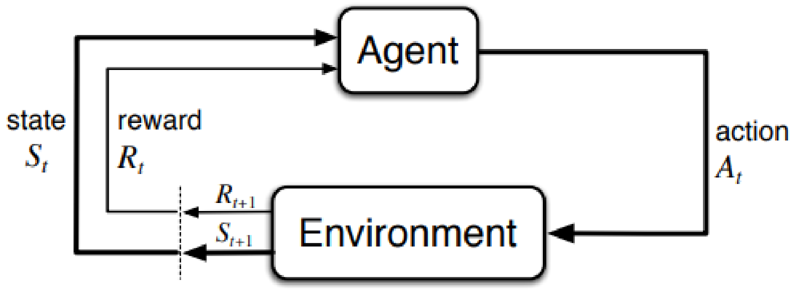


Figure 3. A depicted Markov-Decision-Process from [7] "Reinforcement Learning, second edition: An Introduction"

[1]often simplified as only state dependend $r_t = R(s_t)$

The process of state, action reward and new state (s, a, r, s') is repeated within an *episode*, or also often denoted as *trajectory* in literature such as [7], which consists of iterations between states s_t, actions a_t and rewards for the upcoming state s_{t+1} and ends with the terminal timestep T.

$$\tau = (s_1, a_1, r_2, s_2 ... s_{T-1}, a_{T-1}, r_T, s_T) \tag{3}$$

If the terminal state of one episode has been reached, then the sequence is complete and a new episode can begin. Thus, with these components a **Markov-Decision-Process** (Fig.3) can now be defined, which encapsulates the agent-environment relationship more formally than depicted in Fig.2, but it also retains a certain degree of abstraction because of the variety of possible environment descriptions and actions. Therefore, it functions as a basic template for Problems and in general, all RL-Problems can be represented as a Markov-Decision-Process as it utilizes a feature called *Markov Property*, that ensures the next state s_{t+1} is only dependent on the current state s_t and does not take any previous state into account.

$$\mathcal{P}(s_{t+1}|s_t) = \mathcal{P}(s_{t+1}|s_1, ..., s_t) \tag{4}$$

Or more formally defined in [7] as a state-reward transition function that also takes the current action as a conditional dependency into consideration, which is derived from the transition function that has already been introduced .

$$p(s', r|s, a) \doteq Pr(S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a) \tag{5}$$

This clarifies that the calculated probability for the next state will only derive from the current state and is not disrupted by any former states that already have been surpassed, which means that the future is independent from the past.
As already stated, the main goal and task of the agent is to maximize the accumulated rewards over the episodes but as stated in [7], anything that the agent can not interact with indiscriminately is considered part of the environment however, the agent is able to assess the worth of a particular future state, in which it determines how useful the state can be by computing a ***Value function***. This function is comprised of future reward values that come *after* the current reward, which is summarized as a ***return***[2] G_t.

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k_{t+k+1} \tag{6}$$

The γ notation represents a value in a $[0, 1]$-interval called a *discount factor*. This factor displays the future rewards worth based on how far in the future it lies, while it is observed from the present and the closer the value is to 1, the more important it becomes to consider. Thus, the state-value function, or referred in [8] as *On-Policy Value function*, under a policy π, can now be described to assess a state's utility, by computing an expected value from the return based on the state at the current timestep.

$$\begin{aligned} V_\pi(s) &= \mathbf{E}_\pi[G_t|S_t = s] \\ V_*(s) &= max_\pi V_\pi(s) \end{aligned} \tag{7}$$

[2]note that it can also be summarized as: $G_t = r_{t+1} + \gamma G_{t+1}$

The second function describes the **Optimal Value function**. Since it follows a policy π, one can also determine the Value function that follows an optimal policy $\pi_* = argmax_\pi V_\pi(s)$, meaning the worth of that particular policy is better than all other policies and thus, it gives out the maximized return.
In the same manner, the action that is being taken in the state can also be included into a Value function. This is called a Q-function or *On-Policy Action-Value Function* [8] and returns the respective Q-value which, similarly to the state-value function, uses $\pi = argmax_\pi Q_\pi(s, a)$ for the optimal Q-function.

$$\begin{gathered} Q_\pi(s,a) = \mathbf{E}[G_t|S_t = s, A_t = a] \\ Q_*(s,a) = max_\pi Q_\pi(s,a) \end{gathered} \tag{8}$$

Consequently, the agent can now assess, via the returned Q-value, which state and action would benefit the most and choose accordingly. Now that these two functions are introduced, we can now evaluate an action not necessarily by how good the action can be but how much *better* the action is compared to other actions for a policy π. This is called the **Advantage Function** [8].

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s) \tag{9}$$

The difference between the Q-function and the state-value function will return the average *advantage* of a particular action to that of other random selected actions, which makes it easy for the agent to choose an optimal action in a given state. Therefore, there is no need for the agent to necessarily find the best action since it can estimate an action's consequences by comparing them. These functions will play a pivotal role later on in the advanced concepts.
This covers the basic structures and introductory approaches to Reinforcement Learning, which is fundamental to understand the more advanced concepts but also the main environment that this thesis is based on. It is to be noted that there are more concepts for this section that can be expanded on however, it would stray from the focus and the goal of this study, which is why only the most essential concepts are covered. The same can be said for the advanced concepts, where only the most important aspects, that are connected to the basics, are being presented.

4.1.3 Concepts of Reinforcement Learning: Advanced

As mentioned already, the in-depth look will only focus on the more vital parts of the advanced fields, to not get off the main working field which are tactical networks, that are being introduced later on.
To broaden the topic and taking a deeper look into Reinforcement Learning, the basic aspects can be more elaborated to fit more into the context on what an RL- Agent should achieve. One such aspect is the value function that can be comprised into so called ***Bellman Equations***, that transforms the value function and the Q-function into *immediate reward* function with discounted future rewards.

$$V(s) = \mathbf{E}[G_t|S_t = s] = \mathbf{E}[R_{t+1} + \gamma V(s_{t+1})|S_t = s] \tag{10}$$

This returns the value of the current state when an action is chosen alongside with the expected values from future states and actions. Just like the original value - and Q-function, they can be further expanded when following an included policy π.

$$V_{\pi}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_{\pi}(s_{t+1})] \tag{11}$$

This is an extension of the first equation and it simply states a calculation over the sum's of a, s' and r with a computation of the probabilities. It averages the possibilities, by evaluating the probability of how likely one possibility can occur. Therefore, it gives us the value of the current state followed up by the values of the states that come after it, while following a policy π and its according actions.
With the bellman equations, it is now possible to introduce certain approaches that solve RL - Problems. A basic approach would be **Dynamic Programming**(DP), which focuses on evaluating value functions and improving the given policy π. Three steps are being introduced to fully envelope the principle of dynamic programming. The first step is called *policy evaluation* and as the name states, it computes a value function V_{π} for the policy. Considering the former equation $Eq.11$ again, an *update rule* can now be formed to obtain each successive approximation for the value function in view of all states.

$$V_{k+1}(s) = \sum_{a} \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V_{k}(s_{t+1})] \tag{12}$$

This means, for every k a followed up approximation can be computed in $k + 1$ or in other words, the value of the former state is replaced by the value of the newer state that is computed with the successors of the former state. The next step is called *policy improvement*, which in essence aims to generate a better policy based on the current computed value functions for a policy π that were discussed before. Therefore, one must consider the action that is selected when following the policy to evaluate, if a better policy can be achieved. Thus, we take a look at the action-value function again in a expanded form.

$$Q_{\pi}(s,a) = \sum_{s',r} p(s',r|s,a)[r + \gamma V_{\pi}(s_{t+1})] \tag{13}$$

The main question is now to assess if $Q_{\pi}(s, \pi'(s)) \geq V_{\pi}(s)$ is true, while $\pi'(s)$ and $\pi(s)$ are a pair of policies, whereas $\pi'(s)$ is greedily chosen $\pi'(s) = argmax_a Q_{\pi}(s,a)$. If the comparison is true, then it means that $\pi'(s)$ is at least as good as π or even better, in which case it means $V_{\pi'}(s) \geq V_{\pi}(s)$ is also true and the new policy $\pi'(s)$ is being followed instead of the former policy. Lastly, when these two steps are accomplished, it remains to summarize them and form it into a single process called *policy iteration*. This process is basically a sequence to improve the policies and value functions with the two first steps that have been described.

$$\pi_0 \xrightarrow{\text{evaluation}} V_{\pi_0} \xrightarrow{\text{improve}} \pi_1 \xrightarrow{\text{evaluation}} V_{\pi_1} \xrightarrow{\text{improve}} \pi_2 \xrightarrow{\text{evaluation}} \ldots \xrightarrow{\text{improve}} \pi_* \xrightarrow{\text{evaluation}} V_* \tag{14}$$

This guarantees that each policy is an improvement to the former policy until it reaches the best evaluation. Furthermore, the value function also converges to an optimal solution, since it is the result out of the evaluation of the current perceived policy, which in turn has been

improved one step before. This concludes the dynamic programming approach and to summarize, this method aims to provide an improvement to the given policy which also results in evaluating an ever improving value function to the current policy.
Another approach called the **Monte-Carlo Method**, acts according to the experience that is learned from each episode. Moreover, this process does not require the knowledge of the environmental dynamics to function. The sole idea is to compute a mean return that approximates the expected return from complete episodes meaning, every episode has to terminate eventually so that the experience can be gathered from it more specifically, it approximates using the action-value function $Q_\pi(s, a)$. Furthermore to put it into a more compact perspective, the method to evaluate the optimal policy is quite similar to that of the policy iteration step from dynamic programming. The idea is to improve the policy greedily as well with $\pi(s) = argmax_a Q(s, a)$ and from that newly evaluated policy, it generates a new episode. This episode is further processed to approximate the aforementioned action-value function. From that function it repeats the steps from the beginning by evaluating a new and better policy.
The last common approach, that is being considered to be a combination of dynamic programming and Monte-Carlo Methods, is called **Temporal Difference Learning** (TD-Learning) and is considered to be the novel solving method. The central essence of TD-Learning is that it learns from episodes of experience like other methods such as Monte-Carlo Methods[7], which means it does not need a model to learn the environment and to improve the learning process. In fact, it utilizes a method called *Bootstrapping* meaning, it does not need to rely on complete returns and rather learns from estimates and guesses hence, there is no need to await the final outcome of an episode and can learn from *incomplete* ones. The goal of TD-Learning is to update the Value function $V(s)$ to an estimated return G_t called a *TD-Target* and thus, improving the function and policy alike.

$$\begin{aligned} V(s) &\leftarrow V(s) + \alpha[R_{t+1} + \gamma V(s_{t+1}) - V(s)] \\ &\text{and for Q-function} \\ Q(s, a) &\leftarrow Q(s, a) + \alpha[R_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s, a)] \end{aligned} \tag{15}$$

Note that the parameter α states the learning rate, that controls the extent of the update and $R_{t+1} + \gamma V(s_{t+1})$ is the expanded return function G_t. There are algorithms that use TD-Learning to form an optimal policy π_*. First, we have to distinguish between *On-Policy* and *Off-Policy*. On-Policy means, that the target policy is used to generate samples and data for the algorithm's training. On the other hand, off-policy method make no use of the target policy and focus rather on behaviour policy, which returns a distribution of episodes.
To showcase the difference, a short theoretical examination on some algorithms will be presented, that differ in their usage of the target policy. The first algorithm is called *SARSA*, a reference to the agent-environment interaction sequence $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$, is an on-policy approach, which focuses on updating the Q-Value (action-value) rather than $V(s)$ and utilizes a method called ϵ - *Greedy*. This method simply estimates the action-value by observing past experiences and the average rewards of the target action a with a parameter $\epsilon \in (0, 1)$. So when an action is chosen in a given state, it will either choose a random action from all available actions with the probability ϵ or it will choose an action with $1 - \epsilon$ probability with greedy intent based on the current knowledge and information. Thus, SARSA can now apply ϵ - greedy to its own learning process, when choosing the first action and

receiving the reward R_{t+1} in the following state. After that, it will choose the next action a_{t+1} in the same manner and update the Q-function according to Equation (12). This step will be repeated and each choice of the next action relies on the current policy.
The next algorithm called *Q-learning* is an off-policy approach. In its essence, it is quite similar to SARSA, since it also updates the Q-function. Because it is off-policy, the current policy will not be considered to choose the action. Furthermore, the choice of an action lies on the estimated maximum reward of the next state meaning, a Q_* (optimal value) is estimated out of the Q - values. Thus, the update function has to be adjusted according to these steps.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_{a \in \mathcal{A}} Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{16}$$

Note the max statement, which implicates that Q_* is now in consideration, without the current policy. The behaviour policy is rather derived directly from the action-value function.
This covers the basis of the advanced section in Reinforcement Learning but as already mentioned, there are several more approaches and expansions that can be covered and deepened with more mathematics but are not relevant to the current study. This short in-depth look aimed to provide a better comprehension to the working processes and the dynamics of RL-Methods, that can applied to various tasks.

4.1.4 Adversarial Attacks

Reinforcement Learning in itself, if applied correctly, can be a powerful tool to create automated processes, that can be expanded into several aspects of society and the industry. However with the ever growing fields on where Reinforcement Learning can be applied, this also raises a question of security and on how the safety of RL-Models can be compromised. One such approach is called an ***Adversarial Attack***. In essence, these attacks are aiming to confuse an agent in a given environment to deteriorate the performance and thus, the rewards of the agent. Hence, it can lead to critical consequences in high-risk areas such as in autonomous driving or military usages, which is why it is crucial to have a basic understanding of these attacks. This section will give an overall understanding of adversarial attacks and at the end what kind of attacks are being considered for this thesis.
There are two main approaches, in which an attacker can diminish an agent's performance. First, an attacker can craft a manipulation called an *adversarial sample* with certain *perturbation* δ that can be added to the agent's current *observation* x, which is in general the equivalent to the state s but in some cases an observation can contain less information than a state however, from now on we will assume $x = s$ to avoid confusion and misconceptions. This perturbation varies and depends on the current environment but since these attacks where originally created to confuse image recognition and what the agent observes, it usually perturbs pixels of a small amount that is unrecognizable for the human eyes. It then leads the agent to assume that a perturbed observation $x' = x + \delta$ is the correct return of the environment, which then causes the agent to perform an action that is more harmful than beneficial, as can be seen in the example of Fig.4.

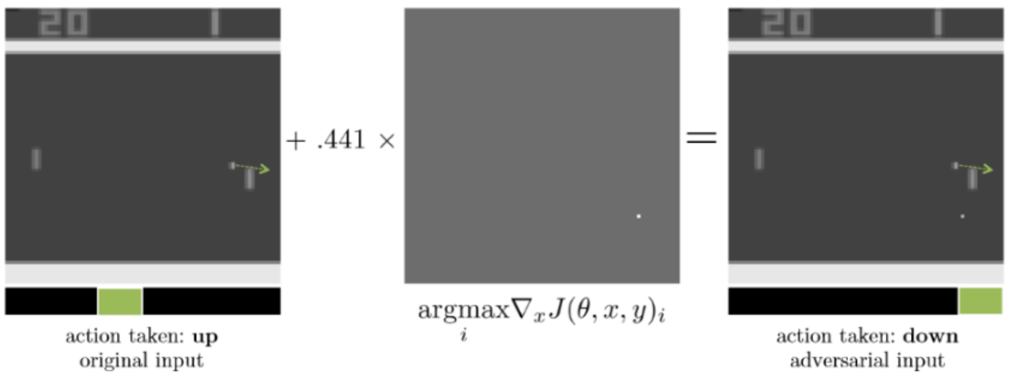


Figure 4. from Huang et al.[9] an example of the game "Pong" to show what a perturbation can cause

In other cases, an agent can be trained to identify and classify images and an adversarial attack would cause the agent to misclassify an image as described in Goodfellow et al.[10]. For example, a slight change of pixels in an image of an animal would still look the same to a human but the agent would perceive it as a completely different animal.Furthermore, another form of attack is to form a maliciously trained agent inside the given environment of the victim agent. The goal of the malicious agent is then to diminish the rewards by performing actions that would yield bad results. This also leads to the question on how a perturbation can be crafted to then be added to the original observation, resulting in an adversarial sample. To start off, there are several methods on crafting adversarial samples however, not all of them are relevant for this study, which focuses on how these adversarial samples can be detected regardless of its crafting process since the strength of an attack would not matter for most detectors but this will be clarified later on. For now, it is just important to know that an adversarial sample x' can be crafted with various methods and applied to the original observation, that leads to a disadvantageous outcome for the agent, with regards of how the perturbation is scaled with a set value ϵ. In spite of that, the important part in this section is to focus on *when* an adversarial sample is applied meaning, there are specific ***attack timings*** that an attacker can consider. For this thesis, we will include two attack timings.
The first one is called ***Uniform attack*** and is considered to be the most basic attack timing for its simplicity. In essence, it will consider every timestep of an episode valid for an attack meaning, every observation that the agent receives will be perturbed with a corresponding adversarial sample x', resulting in a massive drop of performance from the very start. One might think that this approach can be considered to be the optimal solution for an attack however, applied in a real-life scenario it would make the attack too obvious, issuing human intervention most likely to interfere with the attack process. This would obviously defeat the purpose of an optimal attack, which is why some strategical approaches have been developed to answer this problem. This leads to the second attack timing called ***Strategically timed attack*** introduced by Lin et al. [11]. This approach introduces a boundary of when an adversarial sample should be applied in form of a preference function called *c-function* with its corresponding *c-value*. Considering the current observation and a policy π, the function calculates the difference of the most preferred action to the least preferred one.

$$c(s_t) = max_{a_t}\pi(s_t, a_t) - min_{a_t}\pi(s_t, a_t) \tag{17}$$

The value of this function will be further processed with a comparison of a pre-determined threshold β, to evaluate the strength of the preference.If the c-value is greater or equal than the value of the threshold, then it means that the agent has a high preference for an action, that would lead to a very beneficial reward. This is an indicator for the attacker that an adversarial sample x' should be crafted to attack the agent. On the other hand, if c does not surpass β, then it means the observation is not worth attacking, as the preference of the action is not clear enough. To summarize, the attacker must choose an ideal threshold, which could result in a decent number of attacks while also diminishing the agent's reward intake. Choosing a too large threshold would result in few attacks and choosing a too small one would result in many attacks, that could prompt the same problem as described with Uniform attack.
For this study, two attacks have been chosen that are implemented in the study of [3] as well namely being, the *gradient attack* and the *adversarial policy attack*, which is also described by the authors of [12]. In essence, the gradient attack's task is to make an agent's best action less likely while also making the worst possible action in the current scenario more likely by weighing the likelihood of each action and decide accordingly if, in an adversarial probability distribution, the action is the worst possible one. As for the adversarial policy attack, it functions with a trained *adversarial agent*, that trains on the same states as the normal agent with receiving the negative feedback of the normal agent and therefore learns how to minimize the reward. This concludes the essential insights for adversarial attack in this thesis. There are several more attack timings and strategies that have been developed over the years however, they are not relevant for this study as only these above described ones are being experimented with.

4.1.5 Tactical Networks

Tactical networks are a special case of Mobile Ad-Hoc-Networks (MANETs) [13] and just like MANETs, they encompass a working process of independent nodes, representing the different devices and stations that communicate with one another. Furthermore, tactical networks are designed to work in critical scenarios such as military operations or high-security communication, which begs the unavoidable question of how reliable and secure these networks are. Since it functions with independent node, the entire network is based upon the reliability of each separate node meaning, the nodes contribute to the overall reliability.

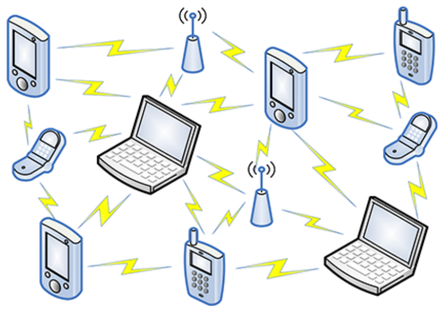


Figure 5. example of Mobile Ad-Hoc Network

The environmental description of the model described in [3] consists of a graph $G = (V, E)$ with weighted edges and a function $w : E \rightarrow \mathbb{R}^3$ that can assign a certain weight $w(e)$ to an edge $e \in E$. The model of a Barabasi–Albert (BA) is used to ascertain a number of graphs with different network topologies, simulating several diverse networks. The goal is to traverse a path ϕ from the starting node S to a destination node T, by taking the shortest ϕ^*. This means, the path with the chosen edges has to have the minimum set of weights. In detail, a set of metrics are defined for the simulated network, that are used to weight the edges primarily being, a queue length $q_e \leq Q$, a packet loss rate $l_e \leq L$ and a data rate d_e. Therefore, a link between nodes is described as an edge between the sender node $v_i \in V$ and receiver node $v_j \in V$, which can be further constructed as an entire path $\forall e = (v_i, v_j) \in \phi | w(e) = [q_e, l_e, d_e]$. In terms of the RL-Agent, it can perform an action by choosing to take certain nodes that connect with the current perceived node. Should the agent choose a node that is not in direct link with the current one, then it is returned as an illegal action with its corresponding numerical reward of -5. This discourages the agent from directly jumping to the target node or taking loops. Should the agent reach the target node T it will receive a reward of 100, thus ending the episode. The rewards for all other cases, is calculated as a function of the defined performance metrics, as can be seen in the reward function $Eq.18$. Whereas w_1, w_2, w_3 represent the weights to the three parameters q_e, l_e, d_e.
This concludes the essential description of the main environment where the experiments and tests are conducted upon. As already shown in [3], this network can be a target of adversarial attacks and reduce the rewards drastically, meaning the attacker leads the agent to perform actions that are illegal and thus, accumulate negative rewards throughout the episode. If performed in a real scenario, this could lead to dire consequences especially when tactical networks are set to be used in these types of extreme environments. Therefore, the question leads what countermeasures exist to alleviate the threats of these attacks and how these countermeasures can be implemented.

$$r(e) = \begin{cases} 100, \text{ reached target node} \\ \frac{1}{w_1} \cdot q_e + \frac{1}{w_2} \cdot (1 - (1 - l_{e_{v_i}})) + \frac{1}{w_3} \cdot \mathbb{E}[d_e] \\ -5, \text{ selecting an illegal action} \end{cases} \tag{18}$$

4.2 Main Focus

Considering all the preliminaries, this leads to the main focus of this study, which are detection methods within the given tactical network environment. In general, a detector can be described as a passive sub-routine that does not actively interfere with the main model and rather *observes* the process and returns within. In some cases, as it will be described later, a detector can also function as a classifier, separating In the simple depiction of Fig.6 one can generalize the basic structure of a detector. First, the input for the detector is actually the output of the main environment being, the observed state and in other cases with a reward as well. The input is then further processed by the given detector, which is the part where different methods come into play and gives an output according to the chosen method, which can vary. This process is evidently run simultaneously to the current main training process to ensure that every step is covered.To present the threat of attacks, we want to take a closer look on the final results of [3], in which we can determine the severity when attacks can run freely and without the agent nor the human being able to realize it.

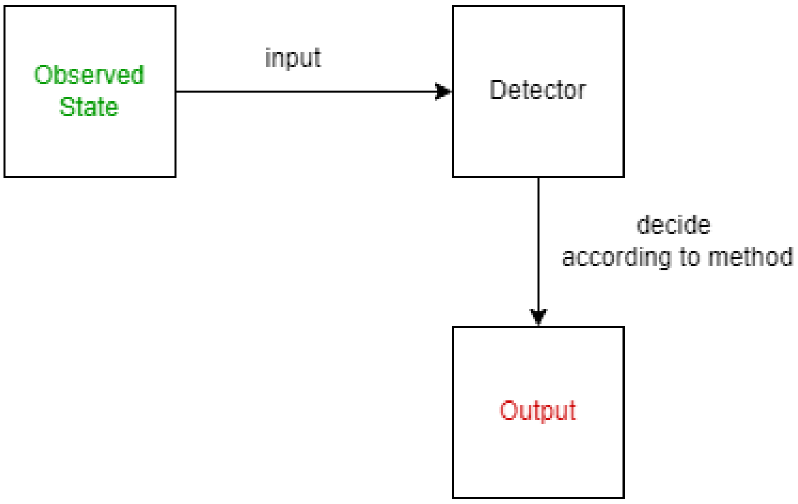
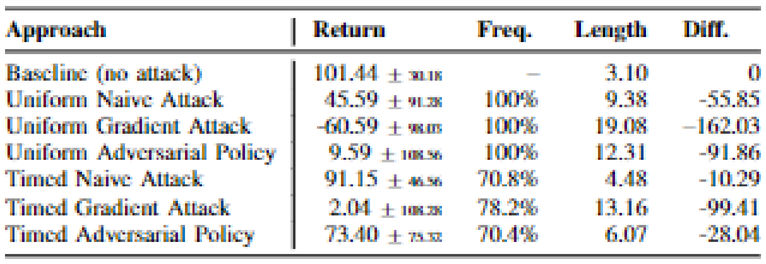


Figure 6. general approach of Detector

As previously covered in *Adversarial Attacks* (Sec.3.1.4), a perturbed observation x' is created to let the agent make a bad decision which would lead to a heavy reward loss and drop of performance. In [3], it has been determined that these adversarial attacks can reduce the performance of the network. This process has been evaluated with different kinds of attacks 7 and timings to present the varieties in which a tactical network can be attacked.



Approach	Return	Freq.	Length	Diff.
Baseline (no attack)	101.44 $\pm$ 30.18	–	3.10	0
Uniform Naive Attack	45.59 $\pm$ 91.28	100%	9.38	-55.85
Uniform Gradient Attack	-60.59 $\pm$ 98.05	100%	19.08	−162.03
Uniform Adversarial Policy	9.59 $\pm$ 108.56	100%	12.31	-91.86
Timed Naive Attack	91.15 $\pm$ 46.56	70.8%	4.48	-10.29
Timed Gradient Attack	2.04 $\pm$ 108.28	78.2%	13.16	-99.41
Timed Adversarial Policy	73.40 $\pm$ 75.32	70.4%	6.07	-28.04

Figure 7. from [3] results of adversarial attacks on a tactical network environment

Observing the results from [3] in Fig.7, we can determine that some of the attacks can reduce the reward so much that it reaches near zero or even lower, which would greatly impair a networks communication performance when perceiving it from a realistic point of view. In addition to that, these results also show the various attack methods that can be used, if an attacker wants to run a different strategy and still reducing the reward with a great amount. Therefore, the task of a detector must now be to correctly recognize the perturbation δ within an observation, no matter what kind of attack is being used or what attack timing is in the working.

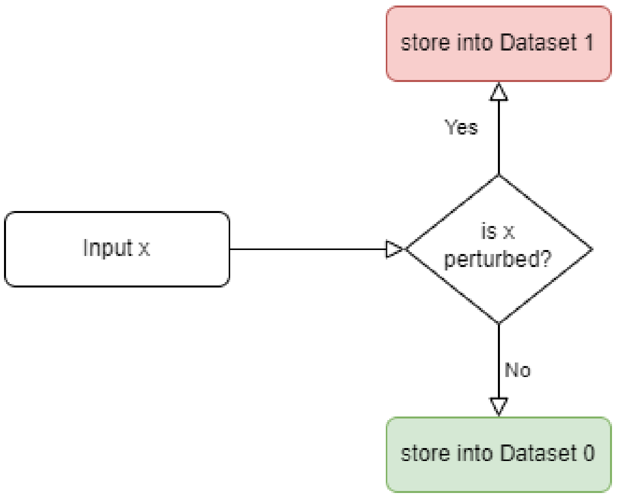


Figure 8. idea of binary detector

5 Methods

In the following, two prominent detection methods for adversarial attacks are presented in a theoretical manner and also the steps on how it could function on a tactical network. Furthermore, we will discuss a self-crafted method on how detectors itself can be attacked. It is important to note that several detection methods have been proposed to counter and alleviate the threat of adversarial attacks however in this study, the focus will only lie on detectors that can actually work with the presented environmental dynamics and achieve effective results, that showcases the ability of detectors through example implementations. Furthermore, an outline will be conducted at the end to present the downsides of such detectors.

5.1 Binary approach

One basic method would be the so called ***binary detector*** or ***binary classifier*** as it has been titled by the authors of [14] and [15]. This approach supports the simple idea of detecting adversarial samples from the original observation, by separating and labelling them with a classifier. In detail, if the current observation x is perturbed into x', then the goal of the detector would be to determine δ out of the observation.

$$x = x' \setminus \delta$$

An important note here is that it only **detects** perturbations and thus, the return would be labeled as true and false or rather *binary* for every observation. However, since this method has been conducted for image datasets and subsequently image recognition algorithms, the components have to be altered for a tactical network but the approach itself will stay the same.For example, the authors of [14] set X as a set of images but for a tactical network it would be the set of possible states that the agent can reach. In addition, the labelling of images and the notation of pixels is not needed, as there is no such requirement in a tactical network.

As for the current given environment, which is the tactical network, there will be no comparison by pixels but rather the inner properties of an observation. However, since an adversarial attack will perturb the observation into something that has different properties than the original observation, the observations can simply be "simulated" as images which means, for the binary detector to detect and classify the given observations it has to compare the inner values of the state that has been set by the environment. This means for the first step of the detector, it has to retrieve the current observation at the beginning and check if there are any overwriting processes of the adversarial attack, since the main procedure of the attack is to replace the returned state x with the perturbed one $x' = x + \delta$. It performs this step, by temporarily copying and storing x before the agent receive it as a return and then monitors the returned observation for the agent, which could be potentially x'. After that, it performs a comparison of the observations and can therefore determine whether the temporarily stored observation is the same as the one that the agent has received. To simulate the labelling that are vital for separating the various observations after the initial test, two lists are being initialized meaning each observation that is being considered is either stored in the *perturbed* list or *non-perturbed* list, which generates the principle of a binary detector. Afterwards, when the whole process is complete it will return a percentage of the overall observations that have been perturbed as well as the normal non perturbed state. This should simulate, that the detector has thoroughly classified each observation. To get a clearer picture of these steps, the simplified pseudocode of Alg.1 will be explained and it shows again the steps more in detail as described previously. To abbreviate the descriptions of the single processes, the lines will be denoted as L_n. As already mentioned, lists are needed to simulate the labelling process by separating the observations namely, clean[] for the set for unperturbed observations, adv[] the set to determine if an observation is indeed perturbed and lastly another list is introduced as to monitor the summarized observations.

Algorithm 1 Simple binary on Tac.network environment

```
1:  obs[]
2:  adv[]
3:  clean[]
4:  **Input**: $x_{orig} = x$ from main environment   ▷ $x$ has to be gathered before agent receives it
5:  obs[] $\leftarrow x_{orig}$
6:  after agent receives observation $x$ begin evaluation:
7:  **if** $x_{orig} == x$ **then**
8:      $x$ is same as $x_{orig}$
9:      store $x$: clean[] $\leftarrow x$                         ▷ "labelling"
10: **else**
11:     $x$ is $x_{adv}$
12:     store $x$: adv[] $\leftarrow x$                           ▷ "labelling"
13: **end if**
14: **return** length.lists of stored observations
```

This is also the reason why the first operation the detector does is to store the original observation into the list and repeats it for every observation. Afterwards, the detector will again include the main environments observation, in which the central operation begins with the

aforementioned evaluation in L_7 and reassures if the observations are still the same and that x has not been tampered with. Since one observation of the tactical network contains several values, that have been mentioned in *Tactical Network*(Sec.3.1.5), such as "packet loss","data rate" and "queue length", an adversarial attack would create a sample that manipulates these values. Therefore, the detector has to check each and every value within the observation to ensure that no value has been altered and would notice if a single value is different from the values of the original observation x. If this is the case, then it is obvious that x is an adversarial sample[3] because the one or more inner values of an observation is not the same as the initial received one. Consequently, one such observation will be stored into the associated list adv[]. The same process is executed if the observation carries no perturbation and it will be stored into clean[]. Afterwards, when the whole process is finished, the detector will return the length of the lists so that a count of those stored observations can be conducted.
This covers the process on how the binary detector could work on a tactical network environment with slight alteration to the original intended idea. Detailed results and explanations can be found later on in *Results*(Sec.5). In general, this method is a basic approach on which other detection methods can be build upon. Furthermore, it is effective for detection purposes as it acts as a classifier in which the perturbed observations are separated from the clean ones. However, there are also several limitations to this approach such as, it has no other purpose or action save from detecting observations and requires human intervention once adversarial samples have been detected, which makes it rather insufficient if there is actual need to defend from adversarial attacks. This is especially critical within high-risk environments such as tactical networks as quick intervention would be needed if such a scenario takes place.Despite this, the binary detector or binary classifier stands as a solid basic foundation for other possible detection methods and the main principle can be adapted to every environment since the sole focus of this method is to recognize a perturbation δ within a given observation regardless of how the environment dynamics, such as rewards or transitions are build or what actions the agent can take.

5.2 Foresight approach

A more sophisticated and advanced method has been proposed in [16] that also works with image datasets, which focuses more on the agent itself rather than the perturbed state. In detail, this method takes one frame at a time that could vary from each environment (e.g frames from video games, or simple pictures) but the important thing is that the agent has actions to take within the given environment. The task is not only to detect adversarial attacks but to make them as ineffective as possible throughout the training process and to do that, the main consideration is located within the ***action distribution*** $\pi_\theta(x_t)$ of a state x_t, that the agent calculates. An action distribution describes the likelihood of an action that the agent can take and in an optimal unattacked scenario the action that yields the best result will have a higher likelihood to be chosen by the agent. An attacked scenario would result in the opposite outcome of the agent most likely choosing the worst possible action.
To detect an attack, while utilizing the action distribution, the authors of [16] propose a ***frame prediction model*** denoted as G_{θ_g} that predicts, on the current considered observation[4]

[3] x_{adv} is the same as $x' = x + \delta$

[4] note that x can also be denoted as s

x_t, a predictive state $\hat{x}_t$, by taking the previous m observations x_{t-m} and the possible actions that can be chosen, as inputs for the prediction.
Thus, the prediction model also returns an action distribution for the predicted state $\hat{x}_t$ just like the actual agent, which can be further utilized to estimate an attack by comparing the distribution of the agent $\pi_\theta(x_t)$ with that of the prediction model $\pi_\theta(\hat{x}_t)$. If an attack has not occurred, then the result of the comparison should be somewhat similar but not necessarily the same, since $\hat{x}_t$ is only a predicted state and lacks the precise estimation of the actual state. However, should the current observation be a perturbed state x'_t then the corresponding action distribution $\pi_\theta(x'_t)$ should differ massively from $\pi_\theta(\hat{x}_t)$. Since it is not known if an attack has been conducted on x_t, a distance metric D is used to evaluate the similarity and a threshold H is introduced as a boundary to decide, whether the state is perturbed or not. The distance metric is calculated with the l_1 - norm or more commonly known as the manhattan distance, which generally calculates the distance between vectors, using the sum of the vector components. In regards to π_θ, the metric is denoted as $D(\pi_\theta(\hat{x}_t), (\pi_\theta(x_t)))$. To decide the occurrence of a perturbation, the detector will utilize the threshold to determine the result.

$$\begin{aligned} &\textbf{Case 1: } D(\pi_\theta(\hat{x}_t), (\pi_\theta(x_t))) > H \\ &\textbf{Case 2: } D(\pi_\theta(\hat{x}_t), (\pi_\theta(x_t))) \leq H \end{aligned} \tag{19}$$

If the first case occurs, then it would mean that the evaluated distance between these distributions has overreached the threshold and the detector would classify x_t as a perturbed state x'_t. The second case only states, that the distance is still within the boundary and will be treated as a normal state, even if a perturbation has been added, the detector will consider the state unperturbed as the H-Threshold has not been crossed.
In addition of detecting these attacks, there is another novelty that will be executed at the end of the evaluation process. This can be seen at the bottom of Fig.9, when the distance metric has passed the boundary of H and will return x_t as an adversarial state x'_t with its belonging distribution $\pi_\theta(x'_t)$. After detecting the attack, an attempt to preserve the agent's performance is made, by replacing the action distribution of the perturbed state with the distribution of the predicted state. Therefore, this so called ***action suggestion*** would minimize the risk of getting a low performance, even if the predicted distribution is not optimal, it would still be better than the perturbed distribution.
Considering the current tactical network environment, some adjustments need to be made in order for the main principle of this detector to be applicable and to function properly once implemented. Just as described with the binary method, the foresight detector also takes in the current observation in before it reaches the agent but in addition to that, it needs access to the disposition of the agent's original action choice to the current observation. In addition, it also takes on the task of separation but instead of separating perturbed observations from clean ones, it is focused on discerning the observations where an action replacement was necessary and those where no action needed to be replaced. As described previously the foresight detector's main focus is to use the aspect of similarity by comparing the distance between the actions. This particular aspect can also be converged to a tactical network environment however the manner in which the distance is calculated has to be changed to make it easier for the detector on the given environment and similar to the binary detector, the focus on this environment will be on the inner values of the individual observations. Rather focusing on the action distribution it is much easier in a tactical network to discern the distance

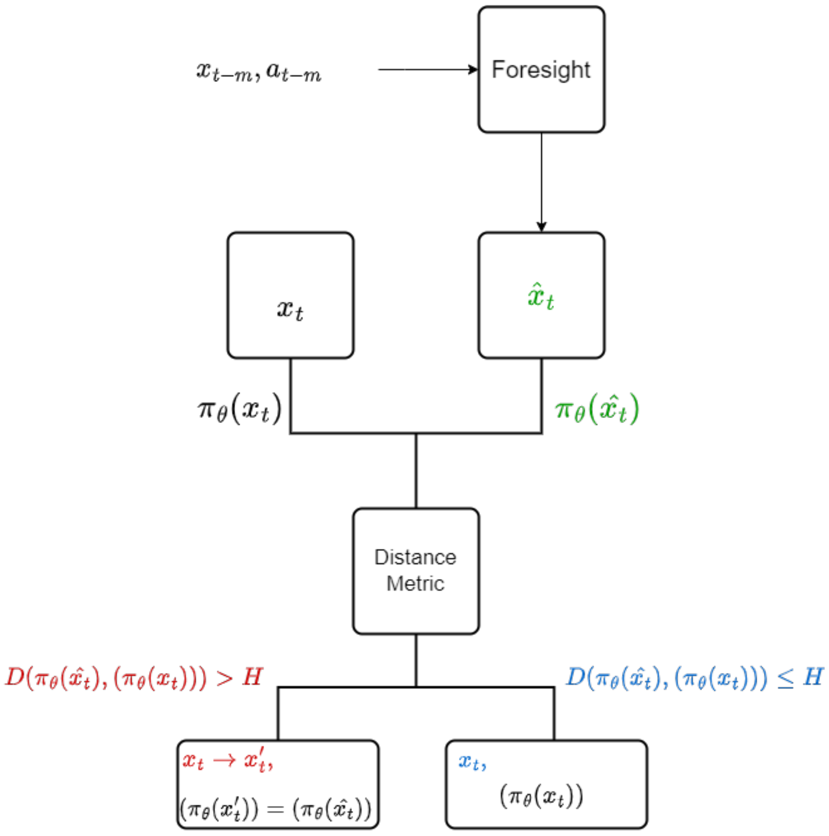

x_{t-m}, a_{t-m}
Foresight
x_t
$\hat{x}_t$
$\pi_\theta(x_t)$
$\pi_\theta(\hat{x}_t)$
Distance Metric
$D(\pi_\theta(\hat{x}_t), (\pi_\theta(x_t))) > H$
$D(\pi_\theta(\hat{x}_t), (\pi_\theta(x_t))) \leq H$
$x_t \rightarrow x'_t,$
$(\pi_\theta(x'_t)) = (\pi_\theta(\hat{x}_t))$
$x_t,$
$(\pi_\theta(x_t))$

Figure 9. Process of foresight

with the observation and thus, the difference with the many values that are stored within and later on, feeding back the predicted action, In the following, we will define $x^* = sum(x)$ as the summed up value of one element x, whereas the element is the given observation from the environment. The values itself consist of certain float and integer numbers, that are used to measure properties such as, *queue length*, *packet loss* and other properties that describe the observation. The aspect of similarity is also upheld in this case, since every unperturbed observation has a similar sum of the values, which is further clarified when seeing the results. One might question on why it is more beneficial summing up the values of the observations instead of focusing on the values of the action and the answer is simply that during testing phases, it has been discovered that the sum of one unperturbed observation differs greatly from that of a perturbed one, more on that will be later further explained and shown in *Results*(Sec.5) however for now, it is sufficient enough to know that the difference can be quite obvious between an unperturbed observation x and a perturbed observation x'. In addition, the threshold will be calculated by taking in the mean sum of the observation's values within the range of the pre-determined episode count. To give a better understanding of the general process, two pseudocodes Alg.3 and Alg.2 are being presented.

The first step is to calculate the threshold H by summing up each returned observation's values x^* and storing them into a list (Alg.2). Next, to actually find a valid threshold for the current process (Alg.3, a mean value will be taken from the returned list mean_threshold[] of summed up observations (L_5). Since the sums are quite similar to one another, this should pose as a valid threshold to be installed within the foresight detector.

Algorithm 2 threshold foresight detector

```
1: mean_threshold[]
2: Set episode_range
3: for 1 to episode_range do
4:     calculate $x^* = sum(x)$        ▷ $x$ is returned from environment and unperturbed
5:     mean_threshold[] $\leftarrow x^*$
6: end for
7: return mean_threshold[]
```

Algorithm 3 Foresight detector for tactical network

```
1: replace_counter[]
2: non-replaced[]
3: get mean_threshold[]                          ▷ from threshold calculation
4: Input $x_{orig} = x$                          ▷ from main environment
5: calculate $H = mean$(mean_threshold[])
6: procedure START EVALUATING:($x$)
7:     predict $\hat{a}$ for $x_{orig}$
8:     calculate $sum(x)$ & $sum(x_{orig})$
9:     calculate distance $D = sum(x) - sum(x_{orig})$
10:    if $D > H$ then                            ▷ Case 1
11:        replace current action $a$ with $\hat{a}$
12:        replace_counter[] $\leftarrow \hat{a}$
13:    else if $D \leq H$ then                    ▷ Case 2
14:        non-replaced[] $\leftarrow \hat{a}$
15:    end if
16: end procedure
17: return replace_counter[],non-replaced[]
```

Afterwards, the evaluation process starts in order to determine the two different cases that were described before. In other words, the detector wants to find out if an action needs to be replaced or not in order to alleviate reward loss. Identical to the binary detector, it also takes in the current observation in before it reaches the agent (L_4). This observation will be used to predict a certain action $\hat{a}$, which will be important later on once the evaluation is concluded. With regards on the calculation of the distance, the summed up values need to be considered again, with $sum(x)$ being the sum of the observation x that the agent has now received and $sum(x_{orig})$ being the sum of the observation that has been retrieved at the very beginning and therefore is known to be unperturbed. After that, a simple calculation $D = sum(x) - sum(x_{orig})$ (L_9) determines the distance between the two considered observations and the detector can now determine whether x is perturbed with an adversarial sample and replace the action or x is not perturbed and no replacement will be necessary. This is the penultimate step of Fig.9 whereas we set $\hat{a} = \pi_\theta(\hat{x_t})$ and $a = \pi_\theta(x_t)$. The algorithm checks now if $D > H$ is true, in which case a replacement will be made with the predicted action $\hat{a}$ being fed back to the agent to replace a bad action caused by the adversarial attack and the replacement action will be further stored into replace_counter[] to measure the amount of replacement that were necessary. On the other hand, if D does not exceed H then no replacement is necessary, since the boundary has not been crossed even if x is perturbed meaning, it does not matter if the agent will perform an action caused by an adversarial attack because the attack in itself was not strong enough to be considered critical. Therefore, we can store $\hat{a}$ into the non-replaced[] list and when the whole procedure is finished, a percentage on how many actions had to be replaced and on how many where not replaced will be returned.
To conclude this method, the foresight approach uses a prediction so that it can form another action distribution for a comparison. Since the work was conducted on image recognition, the approach for this study's environment has to be slightly altered. This is especially important when considering the action distribution in a tactical network environment, which would be quite different from simple atari games. Therefore, the threshold and the evaluation method have to be adjusted to the environments conditions as well as the prediction model. In detail, the changes were made on how the distance is calculated as well as the evaluation. Despite this, the main core of the foresight detector's principle were upheld by correctly replacing malicious actions with unperturbed predicted actions. The adjustments that needed to be made also imply, that the foresight detector is not as flexible as the binary detector because of the various environments that the detector has to adapt to in order for it to function. However, if used correctly it can be a viable reactive defensive technique to counter adversarial attacks, in which the reward loss will be alleviated through the predicted action.

5.3 Evading Detection

After discussing two detectors that could function well on a tactical network, a possible question remains on whether these detection methods can be somehow deceived as well. In this section a simple self-crafted method will be proposed to attack detection methods. Furthermore, an example of this attack on the *Binary detector* (Sec.4.1) will be conducted to showcase, if the attack achieves its goal.
In essence, this method is similar to an adversarial attack but the key difference is, that no

perturbation δ or perturbed state x' is needed to cause damage. As discussed in Sec.3.2, the main task of a detector is to detect δ within a given observation, which is why the main target of this attack will be focused on this particular step of the detector, since a detector needs access to the given observation before the agent receives it to verify its validity. Therefore, we can "invert" the process of the adversarial attack without creating anything by ourselves meaning, instead of creating a sample δ to fool the detector, like it is done with the adversarial attack on the agent, one can simply obtain the original unperturbed version of the current given observation. Just like the detector, the attacker may intercept the observation x before it reaches the agent and make a copy for itself. For the sake of simplicity and to avoid confusion, the observation that the attacker[5] receives will be labeled as $x_{env} = x$.

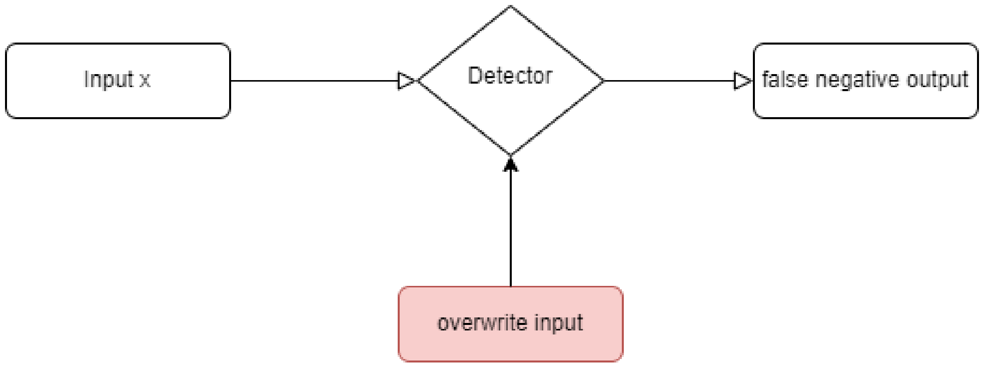


Figure 10. abstract process of evading detection

As one can observe in the simple depiction of Fig.10, the attacker aims to overwrite the input of the detector so that it can create false negative outputs meaning, it classifies a potential perturbed observation as unperturbed and thus, the adversarial attack avoids detection. Now that the attacker has intercepted and stored the original observation, it can overwrite the step where the detector evaluates the observation that the agent receives (L_6 of Alg.1). This however implies, that the attacker has knowledge of the inner workings of the detector, which would not always be the case. Therefore, one has to somehow guarantee that the attacker is able to overwrite the input regardless of which detector is chose. This can be easily achieved by conducting the attack *after* the creation of the adversarial sample, so that it ensures the input of the detector will always be overwritten no matter which detector is in place. If the attack is now conducted, the detector will falsely classify the observation as a *non-perturbed* observation, since the evaluation will be conducted as $x == x_{orig}$ whereas x will always be the same as x_{orig} since it remains unperturbed. This is why, the detector will always store x into the clean dataset and the original attack will be continued without being noticed. To summarize, there are now three observations that are active in play here. One for the adversarial attack and the agent respectively x, one for the detector x_{orig} and another one for the attack to bypass the detector x_{env}. This is clarified more in detail with the simplified pseudocode of Alg.4. As already stated before, L_n is to label the lines of the algorithm.
The first step is to initialize a list where the potential observation to overwrite the detector's input are stored in. Just like the detector, it receives an input as the original observation from the main environment at the start of the process, where an observation is returned. L_3

[5]attacker in this case does not mean adversarial attack

Algorithm 4 bypass detection:Binary

```
 1: replace[]                                  ▷ store original intercepted observation
 2: Input $x_{env} = x$ from Environment
 3: if $x$ != $x_{env}$ then
 4:     $x \rightarrow x_{adv}$ true
 5:     Set $x = x_{env}$
 6:     replace[0] $\leftarrow x$
 7:     replace[0] is to be given to detector
 8: else
 9:     $x == x_{env}$ true                    ▷ no need to do anything
10: end if
```

is used to check if the adversarial attack is being conducted and thus, rendering the original observation into a perturbed one. This is supposed to happen *before* the detector receives x as an input to classify. Therefore, if L_3 returns true, then an adversarial attack has happened and we know that x is x_{adv}. Continuing, the attack will restore the attacked observation to the original one and store it into the list. One important thing to note is that it will always store the observation into the first place of the list meaning, other spaces are simply not needed. This simplifies the attack as it will merely replace the first place of the list after every created observation so that $x_{env} ==$ replace[0] is always true. This is important, as replace[0] will be fed to the detector and replace the initial evaluation of L_6 in the binary algorithm1. After replace[0] is given to the detector, it will falsely classify it as a non-perturbed state because ultimately the evaluation will be between two identical unperturbed observations, while the perturbed observation by the adversarial attack remains undetected and causes damage to the agent. Otherwise if x_{env} is the same as x then nothing shall happen, as no attack has been conducted and the binary detector would classify it as unperturbed regardless. It is important to understand that this attack only targets the detector and does *not* feed the restored observation (L_5) back to the main environment as it would give the agent a correct observation, which would obviously counteract the attackers intention. This concludes the self-crafted exemplary method on how a detector can be deceived into classifying observation as false negatives and thus, not noticing the adversarial samples and δ respectively.
One could ask the question on why the attack is not conducted on the *Foresight detector*(Sec.4.2) as well. To answer this question, we recall the basic principle of the foresight detector, which is to alleviate the reward loss caused by the adversarial attack. This is achieved through predicting an action and feeding it back to the agent so that the malicious action can be overwritten. As for the method of bypassing this detector, it would simply result in another adversarial attack because the detector receives a predicted action and an attack would simply be to perturb the observation again so that the agent is deceived once more. Therefore, this attack method is not designed for the foresight detector because of the lack of classification tasks.
The testing results of this attack are being presented at the end of the *Results*(Sec.5) section and will further highlight on what the attack is tasked to achieve against a detector with classification tasks.

6 Results

In the following section the results to the former mentioned methods will be presented with an explanation on the various values that occur. To evaluate these results, tables will be presented to get a clear overview on what the detector does respectively to the outcomes. Since we want to evaluate the performance and working process of a detector, results will be presented in a tabular manner rather than graphical as we are only interested in the values that the detector returns rather than the performance of the agent. The code and all used results can also be found in the git repository [17] to get a closer look on the implementations that were described in *Methods*(Sec.5).

6.1 Results on binary

To begin with, the binary detector will be evaluated on an attack called *Adversarial Policy Attack* however as mentioned in previous sections, it does not matter which attack will be conducted since this detection method only focuses on if a perturbation exists and not on how strong the attack can be on the agent. For now it is sufficient to know, that this attack diminishes the agent's reward accumulation that it will yield a mean reward of < 0 in every process. Therefore, any viable adversarial attack can be used to evaluate the binary detector on its classification task. The focus will rather be on the attack timings and starting off with the *Uniform Attack* timing, we can already derive if the attack crafts an adversarial sample for every timestep then the detector will classify the entire process as perturbed. We start off by performing an example of this scenario with an episode count of 300.

Episodes :	300	Observations:	7178
Uniform Attack		detected:	7178 (100.0%)
		non perturbed:	0 (0.0%)

Table 1. Binary detector on Uniform attack

As we can see in Tab.1, the detector correctly classifies the perturbed observations for this process of 300 episodes which also yielded a mean reward of -7.32. This means, that the method described in *Binary Approach*(Sec.4.1) successfully separates observations that have been targeted with an adversarial sample from clean observations for the uniform attack timing. However as mentioned already, this is example shows the expected result from the binary detector and is therefore, quite trivial to understand. Moving on, when using the *Strategically timed Attack* we recall that the attacker has to pass a certain boundary β for the attack to commence. Considering this, we can now shift the value of β back and forth and observe the results that the binary detector achieves. This will be tested for different β values, whereas ϵ to scale the perturbation is a set value of 0.5, and each process will be run up to 500 episodes and evaluated after every 100 episodes to showcase the continuity of the attack with the current selected β value. This also means, that the binary detector showcases the strength of the attack, because of the boundary it has to pass in order to attack. One important note is, that the amount of the created observations is arbitrary to the

total_episodes: 500	$\beta = 0.7$	**Strategically timed Attack**	**Binary**
curr_episodes :	100	Observations:	1549
		detected: non perturbed:	1547 (99.87%) 2 (0.13%)
curr_episodes :	200	Observations:	3837
		detected: non perturbed:	3830 (99.82%) 7 (0.18%)
curr_episodes :	300	Observations:	6293
		detected: non perturbed:	6283 (99.84%) 10 (0.16%)
curr_episodes :	400	Observations:	8144
		detected: non perturbed:	8134 (99.88%) 10 (0.12%)
curr_episodes :	500	Observations:	10048
		detected: non perturbed:	10037 (99.89%) 11 (0.11%)

Table 2. Binary detector Strategically timed attack $\beta = 0.7$

agent and hence differs with each process. In this process of Tab.2 the β value was set on 0.7 which is a considerable low threshold, resulting in plenty of attacks, that can clearly be seen in the episode iteration. Thus, the attacks that have been detected are almost comparable of that to the uniform attack Tab.1 meaning, that β was set low enough for the adversarial attack to almost commence at every observation. In fact, the attack is so strong, that it yields the same results for $\beta = 0.8$ and $\beta = 0.9$, detecting for almost 100% of the observations as perturbed. This means, that one might scale β even higher to test out differences. Consequently, for testing purposes, we set the β value on 0.999 to showcase if there are any changes in the attackers behaviour. In Tab.3 we can observe one such case, with the episode count unsurprisingly being similar to that of Tab.2. Furthermore, we can clearly see that the attack frequency is still at a high level with only a slight change of the behaviour, with the non perturbed observations going up to around $\approx 1\%$ from $< 0.25\%$. Hence, we can conclude that this attack timing is powerful enough so much so that it can overcome a high set threshold β and conducting frequent attacks on the agent. This also means, that the action preference function described in *Adversarial Attacks*(Sec.3.1.4) yields a high preference value $c(s_t) = max_{a_t}\pi(s_t, a_t) - min_{a_t}\pi(s_t, a_t)$ for the most preferred action of the agent, in which it was able to surpass the boundary. We can conclude as expected that if the β value is set high, then more observations will be classified as non perturbed and on the other hand if the value is set much lower, then similar results to that of the uniform attack timing will be returned, labelling almost every observation as perturbed and thus resulting in a percentage of $\approx 100\%$.

These results also show, that the detector classifies for each episode the same percentage amount of non perturbed observations meaning, that there is no correlation between the amount of episodes that is set and the attack timing. Furthermore, we can determine that if the β value is set progressively higher, then more unperturbed observations will be detected, which is logical, since the preference model has to surpass the threshold. As already mentioned, this can be done with any kind of attack and sample creation because the binary

total_episodes: 500	$\beta = 0.999$	**Strategically timed Attack**	**Binary**
curr_episodes :	100	Observations:	2010
		detected:	1990 (99.0%)
		non perturbed:	20 (1.0%)
curr_episodes :	200	Observations:	3680
		detected:	3641 (98.94%)
		non perturbed:	39 (1.06%)
curr_episodes :	300	Observations:	5784
		detected:	5722 (98.93%)
		non perturbed:	62 (1.07%)
curr_episodes :	400	Observations:	7427
		detected:	7335 (98.76%)
		non perturbed:	92 (1.24%)
curr_episodes :	500	Observations:	9487
		detected:	9376 (98.83%)
		non perturbed:	111 (1.17%)

Table 3. Binary detector Strategically timed attack $\beta = 0.999$

detector acts indifferently to the methodology of an adversarial attack and is only interested in perceiving the perturbation. It also shows, that even when using an attack timing with a boundary such as the strategically timed attack, frequent attacks are being conducted due to the high preference to an action of the agent meaning, that the environment is set up for the agent to have a clear action to choose from.
Since this detection method yields similar results to every change of the parameters, we can conclude the binary detector as a static but effective evaluation method on finding out perturbations within the returned observations.

6.2 Results on foresight

Contrary to the binary detector, we can evaluate different attacks for the foresight detector, as it operates with a certain threshold H to ascertain if a replacement of the current action is needed in order to alleviate loss. This means, that we can also examine the strength of an attack because of the distance metric that is introduced in the foresight detector. We will again take a look on each 100 episode iteration to potentially ascertain the meaning of the various values. An important note here is, that the value of the threshold is not the same as the reward value, as the threshold is a simple calculation of the sum to the individual observations within an episode.
The attack that has been used in the evaluation process of Tab.4 was called *gradient attack*, which was also used in [3] to attack the agent. As already pointed out, in addition to the attack itself, we can scale the perturbation with a fixed value ϵ, which was 0.5 in this case. To begin with, the attack timing was set on Uniform attack with a calculation of the mean threshold for every 100 episode iteration. We can observe that the threshold is around ≈ 57.25 for each iteration, confirming the assumption that every unperturbed observation is of a similar nature. Another interesting note is that the percentage amount of actions that needed to be

total_episodes: 500	$\beta = 0.0$	**Uniform Attack**	**Foresight**
curr_episodes :	100	Mean Threshold:	57.24
mean reward after replacements:	104.4	replaced actions:	45.6 %
		non replaced actions:	54.4 %
curr_episodes :	200	Mean Threshold:	57.31
mean reward after replacements:	105.11	replaced actions:	43.46 %
		non replaced actions:	56.54%
curr_episodes :	300	Mean Threshold:	57.26
mean reward after replacements:	104.9	replaced actions:	42.38%
		non replaced actions:	57.62%
curr_episodes :	400	Mean Threshold:	57.2
mean reward after replacements:	104.72	replaced actions:	42.37%
		non replaced actions:	57.63%
curr_episodes :	500	Mean Threshold:	57.19
mean reward after replacements:	104.25	replaced actions:	41.05%
		non replaced actions:	58.95%

Table 4. Foresight detector Uniform attack timing

replaced, is reduced by a small amount after every 100 episode iteration and subsequently the amount of non replaced actions is increased. Since the threshold does not increase massively, we can assume that the reason for this could be due to the attack itself meaning, the attacked observations were as not drastically changed as the observations before them. Consequently, this would result in a low distance value and therefore, fewer actions have to be replaced to uphold the reward loss. From these results, we can further conclude that it should be even fewer replaced actions when using the strategically timed attack since it uses an additional threshold to determine attacks. The evaluation of the process described in Tab.4 resulted in a mean reward of around ≈ 105 without attacks and detection. After the action's replacement with the predicted action $\hat{a}$, the reward was still approximately close to that initial reward value for each episode iteration. This means, that the predicted action, the agent had to take, was very similar to that of the original intended action and thus, the rewards have a close proximity to the baseline rewards. As already hinted at, the strategically timed attack method should yield different results however, we have to take into consideration that the preference of the can be quite arbitrary with each observation and in the overall episode iteration. Considering the results with the binary detector with the timing of strategically timed attack, we can begin directly with a considerable β value.
As we can observe from the results in Tab.5, the threshold is unsurprisingly still in the same value field and once again the mean reward of the baseline was around ≈ 105 meaning, the replaced actions yielded slightly worse rewards but still considerably high. An interesting examination occurs in the evaluation and as we predicted, the amount of replaced actions were just slightly reduced indicating, that either the observation's distances were within the boundary H and the observations were not so different from each other or the boundary for the attack has not been passed. To prove this, we can evaluate a different attack namely the *Adversarial Policy Attack* that was already used in the binary detector as well. We begin with the uniform attack and already we can see a clear difference to the gradient attack 4.

total_episodes: 500	$\beta = 0.8$	**Strategically timed attack**	**Foresight**
curr_episodes :	100	Mean Threshold:	57.35
mean reward after replacements:	103.9	replaced actions:	44.21 %
		non replaced actions:	55.79 %
curr_episodes :	200	Mean Threshold:	57.19
mean reward after replacements:	104.97	replaced actions:	40.05 %
		non replaced actions:	59.95%
curr_episodes :	300	Mean Threshold:	57.03
mean reward after replacements:	104.39	replaced actions:	35.4%
		non replaced actions:	64.6%
curr_episodes :	400	Mean Threshold:	57.16
mean reward after replacements:	104.42	replaced actions:	36.81%
		non replaced actions:	63.19%
curr_episodes :	500	Mean Threshold:	57.15
mean reward after replacements:	104.34	replaced actions:	38.28%
		non replaced actions:	61.72%

Table 5. Foresight detector strategically timed attack timing with $\beta = 0.8$

total_episodes: 500	$\beta = 0.0$	**Uniform attack**	**Foresight**
curr_episodes :	100	Mean Threshold:	57.14
mean reward:	103.69	replaced:	100.0 %
		non replaced:	0.0 %
curr_episodes :	200	Mean Threshold:	57.34
mean reward :	103.64	replaced:	100.0 %
		non replaced:	0.0%
curr_episodes :	300	Mean Threshold:	57.25
mean reward:	104.07	replaced:	100.0%
		non replaced:	0.0%
curr_episodes :	400	Mean Threshold:	57.31
mean reward:	104.17	replaced:	100.0%
		non replaced:	0.0%
curr_episodes :	500	Mean Threshold:	57.32
mean rewards:	104.31	replaced:	100.0%
		non replaced:	0.0%

Table 6. Foresight detector uniform attack timing

The initial examination of Tab.6 is, that all actions had to be replaced in order to alleviate reward loss. This is clearly due to the fact, that the adversarial policy attack perturbs the observation x in a way, where the difference between the original observation is so great that the distance metric D will always return a value higher than H. Indeed, when experimenting the average value of one such perturbed observation by the adversarial policy attack was around ≈ 200 to 300, while a single unperturbed one had a value of ≈ 55 to 60. This is obviously a huge difference and is the main reason on why the threshold H will always be surpassed, which averages around ≈ 57. Furthermore, we can now assume that this is the similar case for strategically timed attack as well especially when picking a low β value, which is why we skip directly to a high β value and evaluate the values from there.

total_episodes: 500	$\beta = 0.8$	**Uniform attack**	**Foresight**
curr_episodes :	100	Mean Threshold:	57.47
mean reward:	104.53	replaced:	93.89 %
		non replaced:	6.11 %
curr_episodes :	200	Mean Threshold:	57.08
mean reward :	103.52	replaced:	93.91 %
		non replaced:	6.09%
curr_episodes :	300	Mean Threshold:	57.24
mean reward:	103.76	replaced:	94.77%
		non replaced:	5.23%
curr_episodes :	400	Mean Threshold:	57.07
mean reward:	103.7	replaced:	94.97%
		non replaced:	5.03%
curr_episodes :	500	Mean Threshold:	57.1
mean rewards:	103.98	replaced:	95.03%
		non replaced:	4.97%

Table 7. strategically timed attack timing $\beta = 0.8$

total_episodes: 500	$\beta = 0.999$	**Uniform attack**	**Foresight**
curr_episodes :	100	Mean Threshold:	56.98
mean reward:	104.46	replaced:	84.78 %
		non replaced:	15.22 %
curr_episodes :	200	Mean Threshold:	57.29
mean reward :	104.55	replaced:	82.01 %
		non replaced:	17.99%
curr_episodes :	300	Mean Threshold:	57.21
mean reward:	104.5	replaced:	82.39%
		non replaced:	17.61%
curr_episodes :	400	Mean Threshold:	57.22
mean reward:	104.25	replaced:	82.46%
		non replaced:	17.54%
curr_episodes :	500	Mean Threshold:	57.17
mean rewards:	104.25	replaced:	82.63%
		non replaced:	17.37%

Table 8. strategically timed attack timing $\beta = 0.999$

As presented in Tab.7, the amount of replaced actions just changes slightly and the overall reward can be determined around ≈ 103.8. Considering Tab.8 with a β value of 0.999, we can see that there are indeed more observations that need no replacement however, the overall value of it remains still high due to the nature in which these observations are perturbed. Another interesting note is, that the ϵ value to scale the perturbation has no major impact

for the replacement evaluation. This could be due to the fact, that the observation itself is already changed in a shape that it differs too greatly from the original observation. The result also implies, that if the β boundary is set even higher then logically, fewer actions have to be replaced because of the reduced attack frequency.
From these results we can conclude, that the foresight detector can be an effective tool to alleviate loss and help the agent's performance to stabilize. Furthermore, we have discovered that the distance metric to evaluate, if an action should be replaced is quite different from the two adversarial attacks that were considered namely, *gradient attack* and *adversarial policy attack*. However as already mentioned in *Foresight approach*(Sec.5.2), this method is quite inflexible and therefore, has to be readjusted and reworked for each environment so that the main principle of the foresight detector can be applied on the current scenario.

6.3 Results on evading detection

total_episodes: 500	$\beta = 0.0$	**Uniform attack**	**binary**
curr_episodes :	100	Observations:	1924
		Detected: unperturbed:	0 (0.0%) 1924 (100.0%)
curr_episodes :	200	Observations:	4130
		Detected: unperturbed:	0 (0.0%) 4130 (100.0%)
curr_episodes :	300	Observations:	6653
		Detected: unperturbed:	0 (0.0%) 6653 (100.0%)
curr_episodes :	400	Observations:	8953
		Detected: unperturbed:	0 (0.0%) 8953 (100.0%)
curr_episodes :	500	Observations:	11672
		Detected: unperturbed:	0 (0.0%) 11672 (100.0%)

Table 9. bypass detection uniform attack

After examining the results on how adversarial samples can be detected, we finally take a closer look on the self-crafted method on deceiving detection methods. This method will be specifically focused on the binary detector, for the reasons already explained in *Evading Detection*(Sec.4.3), and unlike the detection methods, the attack timing and attack method is not relevant in this case. Nevertheless, just to showcase an example we will still evaluate a result with the uniform attack, as it has already been shown at the beginning of this section, that the uniform attack on a detector will return a percentage of 100% of perturbed observations, since it attack at every observation. To recall the main step, the attacker feeds the detector a clean observation to override its evaluation process and thus, the adversarial attack will not be detected and harm the agent's performance.
As we can see in Tab.9, this is exactly the case where the return can almost be perceived as an inverted process to that of the unattacked detector. It is important to note, that the attack has

succeeded and the rewards were reduced from ≈ 105 to around ≈ -14.5, showing that the adversarial attacks was quite effective on its part. Just as mentioned, the attacking strategy is actually of no concern here because of the method described in *Evading Detection*(Sec.5.3) meaning, the saved original observation at the first place of the list will always replace the potential perturbed observation for the detector. Therefore, the attack could hypothetically work with any adversarial attack timing so long as, the intended observation for the evaluation of the binary detector is overwritten with a clean observation, which leads the detector to perceive and inquire if $x_{orig} == x_{orig}$ is true. Just to present and underline the aspect that an attack timing does not matter, an evaluation process with the strategically timed attack timing is presented as follows.

total_episodes: 500	$\beta = 0.999$	**Uniform attack**	**binary**
curr_episodes :	100	Observations:	2141
		Detected: unperturbed:	0 (0.0%) 2141 (100.0%)
curr_episodes :	200	Observations:	4140
		Detected: unperturbed:	0 (0.0%) 4140 (100.0%)
curr_episodes :	300	Observations:	6461
		Detected: unperturbed:	0 (0.0%) 6461 (100.0%)
curr_episodes :	400	Observations:	7977
		Detected: unperturbed:	0 (0.0%) 7977 (100.0%)
curr_episodes :	500	Observations:	9931
		Detected: unperturbed:	0 (0.0%) 9931 (100.0%)

Table 10. bypass detection strategically timed attack

Considering Tab.10 we can see, there are no major changes when using a different attack timing in fact, even the count of observations has stayed similar and remains so with every change of parameter, be it β to set a boundary for the strategically timed attack threshold or ϵ to scale the strength of the perturbation.
We can conclude from these results, that this attack method to bypass detection is quite effective on a simple detection strategy such as the binary detector. Furthermore, we can potentially conclude that it works against any kind of classification task so long as it can return a wrong object for evaluation to the detector, leading to false negative classifications.

7 Conclusion

In this bachelor thesis, we have tackled the question on how adversarial samples can be perceived within a Reinforcement Learning based tactical network. Two main methods have been evaluated namely being, the *binary detector* and the *foresight detector* with two different attack timing strategies one being the *Uniform attack* and the other *Strategically timed*

attack. The first detection method was build on a basic principle of classifying each perceived observation and labelling it correctly so that it returns two correctly classified datasets. To broaden this, limitations of this particular method have been pointed out such as, being too static and superficial because of the human intervention that it requires after it has detected attacks. One the other hand, the binary detector can serve as a good basic foundation for other detection methods, as the core principle of it is very flexible to use and thus, can be executed on various environments. Afterwards, we looked into the more elaborated Foresight detector, which could almost be classified as a reactive defense method. This method not only detected adversarial samples but also helped alleviate the agent's reward loss by replacing the damaging action, caused by an attack. However contrary to the binary method, the foresight detector lacks the flexibility and therefore, has to be adjusted for every different environment. Finally, there are many more detection methods that are used to help against adversarial attacks, however a lot of these detection methods are not well suited for a tactical network environment for example in [18], a method of k-nearest-neighbor is used to determine adversarial samples, which would be ill-suited in the given environment of a tactical network.
On top of detection methods, a simple yet effective self-crafted method to evade detection and help the adversarial attack has been presented. This is shown to be quite impactful on the binary detector, as it always returned false negative outputs and thus, the adversarial attack has evaded detection. Further research could be conducted on defending detection methods itself or creating better and more robust but especially flexible detection methods. This is especially important in critical high-risk scenarios such as military communication where tactical networks play a key role.
As for the question on which detection methods is best suited for tactical network environment, one could say that a compromise is the optimal solution meaning, the flexible aspect of the binary detector could be combined with the replacement factor of the foresight detector, since the foresight detector itself is not flexible enough. This could hypothetically be a viable solution to tactical networks, where the environment itself is quite dynamic.

References

[1] S. Kaviani, B. Ryu, E. Ahmed, K. A. Larson, A. Le, A. Yahja, and J. H. Kim, "Robust and scalable routing with multi-agent deep reinforcement learning for manets," 3 2021.

[2] J. F. Loevenich, P. H. Rettore, R. R. F. Lopes, and A. Sergeev, "A bayesian inference model for dynamic neighbor discovery in tactical networks," *Procedia Computer Science*, vol. 205, pp. 28–38, 2022. 2022 International Conference on Military Communication and Information Systems (ICMCIS).

[3] J. F. Loevenich, J. Bode, T. Hurten, L. Liberto, F. Spelter, P. H. L. Rettore, and R. R. F. Lopes, "Adversarial attacks against reinforcement learning based tactical networks: A case study," 2022.

[4] I. Ilahi, M. Usama, J. Qadir, M. U. Janjua, A. Al-Fuqaha, D. T. Hoang, and D. Niyato, "Challenges and countermeasures for adversarial attacks on deep reinforcement learning," 9 2021.

[5] DeepMind, "Alphago zero: Starting from scratch," 2017.

[6] DeepMind, "Muzero: Mastering go, chess, shogi and atari without rules," 2020.

[7] R. S. Sutton and A. G. Barto, "Reinforcement learning, second edition: An introduction," 2018.

[8] OpenAI, "Introduction to rl," 2018.

[9] S. Huang, N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel, "Adversarial attacks on neural network policies," 2 2017.

[10] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 3 2015.

[11] Y.-C. Lin, Z.-W. Hong, Y.-H. Liao, M.-L. Shih, M.-Y. Liu, and M. Sun, "Tactics of adversarial attack on deep reinforcement learning agents," 3 2019.

[12] A. Gleave, M. Dennis, C. Wild, N. Kant, S. Levine, and S. Russell, "Adversarial policies: Attacking deep reinforcement learning," 2019.

[13] N. Raza, M. U. Aftab, M. Akbar, O. Ashraf, and M. Irfan, "Mobile ad-hoc networks applications and its challenges," 1 2016.

[14] Z. Gong, W. Wang, and W.-S. Ku, "Adversarial and clean data are not twins," 4 2017.

[15] J. H. Metzen, T. Genewein, V. Fischer, and B. Bischoff, "On detecting adversarial perturbations," 2 2017.

[16] Y.-C. Lin, M.-Y. Liu, M. Sun, and J.-B. Huang, "Detecting adversarial attacks on neural network policies with visual foresight," 10 2017.

[17] L. Liberto, "Repository for Code."

[18] J. Raghuram, V. Chandrasekaran, S. Jha, and S. Banerjee, "A general framework for detecting anomalous inputs to dnn classifiers," 6 2021.