FAS Assignment 3: Adaptation Strategy Using Deep Q-learning in DingNet Exemplar

[redacted]

December 24, 2023

Contents

System Description

Our system, based on the DingNet architecture, optimizes LoRaWAN communication in IoT networks by dynamically adjusting mote spreading factors to mitigate packet loss. Central to our design is an adaptive mote, engineered to analyze and respond to the distribution of spreading factors in a densely populated mote environment.

The adaptive mote assesses the current distribution of spreading factors within its communication range. It then selects a spreading factor that is minimally used by neighboring motes, effectively reducing the probability of communication overlap and collision. The success of this approach is based on the principle that less frequent spreading factors are less likely to cause packet loss due to reduced interference and competition for bandwidth.

We tested our system's effectiveness with simulations, which allowed us to see how well it works under different conditions. These simulations, each initialized with different random seeds, present the mote with diverse communication challenges, assessing its capability to minimize packet loss adaptively. One of those seeds can be seen below. Success is measured by the system's ability to consistently lower packet loss rates, regardless of the spreading factor distributions in the simulated networks.

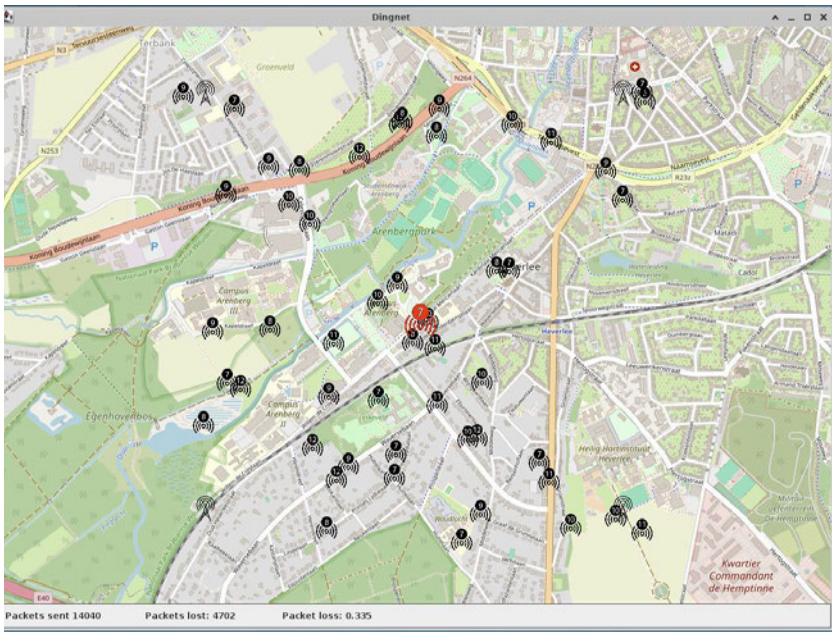

Figure 1. A snapshot of the simulation in DingNet.

Analysis of Uncertainties

Uncertainty analysis is critical for adaptive systems like DingNet. This section outlines the identified uncertainty — reliability — and its impact on system operations. Through a table, we detail how the Deep Q-learning model addresses this challenge, enhancing DingNet's adaptability and performance. The analysis provides a clear framework for understanding the trade-offs and decision-making processes inherent in the system's adaptation strategy. Additionally, there is an introduction on the rationale behind focusing on the uncertainty reliability, justifying its selection based on DingNet's operational needs and the capabilities of the Deep Q-learning model.

Considerations

There were multiple possible uncertainties that we considered to mitigate and which properties we would adapt to accomplish that. However, in the end, we decided on reliability, which is described in more detail below. See reflection for a summary of these considerations accompanied by a rationale for why they were not chosen for this project in the end.

Reliability

The focus on reliability in DingNet is crucial due to its direct impact on system consistency and data transmission accuracy. High reliability is essential to ensure continuous and effective communication between motes and gateways, a core function in IoT networks. Addressing reliability uncertainty is vital for maintaining system integrity, especially in dynamic environments where consistent data flow is key to operational success.

Name	U1-Reliability
Classification	This uncertainty occurs during execution. Reliability of the system depends on the consistency of data transmissions between the motes and gateways.
Context	The amount of packet loss directly impacts reliability. The higher the packet loss, the more packets that need to be sent between the motes and gateways.
Impact	Maintaining a low number of packet loss requires the self-adaptive system (SAS) to adjust the spreading factor of a mote based on the spreading factor of the motes around it. If a group of motes close to each other uses the same spreading factor, the number of packets lost increases.
Degree of severity	For any given mote, the SAS must be able to find the spreading factors of other motes in its communication range.
Sample illustration	The packet loss in the system increases. For a given mote, the SAS

	then first finds the spreading factors of other motes in its communication range. Using these values, it adjusts the mote's spreading factor to a value that is used the least among the spreading factors of the other motes. This reduces the number of motes with the same spreading factor, thus decreasing the packet loss.
Evaluation	A non-SAS will pick a random value for the spreading factor of a given mote. This value may be shared by multiple other motes within its communication range. This potentially increases the packet loss. A SAS adjusts the spreading factor of a mote based on the spreading factors of other motes. This ensures that whenever the packet loss increases, the SAS adjusts the spreading factor of a mote to differ as much as possible from the spreading factors of other motes, which in turn, decreases the packet loss.
Also known as	Accuracy, soundness.

Table 1. Uncertainty analysis of reliability.

Requirements

In adaptive software systems, requirements are critical for defining system functionalities and constraints (Lecture 2). For DingNet, facing the uncertainties inherent in IoT environments, well-defined requirements have been essential in implementing the Deep Q-learning model. This enables the system to effectively manage challenges related to reliability through careful adjustment of spreading factors, thereby reducing packet loss even amid fluctuating network conditions. This section of the paper takes inspiration from Alice's smart office example in slide 33 - Table 1 and 2 - Lecture 2, in formatting the requirements.

Traditional Requirements

This subsection outlines the core functionalities and constraints that DingNet is expected to meet. These requirements provide a foundational understanding of the system's capabilities and limitations before adaptation strategies are applied. Below, you will find each requirement clearly defined with specific actions that the system should take. "SHALL" in this context indicates mandatory requirements that are critical for the successful operation of the DingNet system, considering the uncertainty of *U1-Reliability*.

Reliability (U1)	R1: The system SHALL adjust the spreading factor of a mote to a value that is least used by the other motes in its communication range to minimize packet loss. R2: The system SHALL continuously monitor packet loss rates to inform decisions about spreading factor adjustments.

Table 2. The traditional requirement for U1-Reliability.

Requirements using RELAX language

This subsection translates traditional requirements into adaptive terms suitable for a self-adaptive system. It reflects how DingNet, through Deep Q-learning, can flexibly respond to its operational uncertainties and maintain system performance. It also introduces "MONITOR" and "ENVIRONMENT" elements to clarify what should be observed and under what conditions adjustments should be made.

Reliability (U1)	R1': The spreading factor SHALL be adjusted AS CLOSE AS POSSIBLE TO the least used spreading factor of other motes in its environment to minimize packet loss. MONITOR: spreading factor of other motes and packet loss rate. ENVIRONMENT: operation state of the mote. R2': The system SHALL MONITOR packet loss rates CONTINUOUSLY to inform spreading factor adjustments. MONITOR: current frequency of packet loss. ENVIRONMENT: operational state of the mote.

Table 3. The RELAX requirements for U1-Reliability.

Analysis of Potential Solutions

Introduction

In adapting DingNet, various solutions were explored to enhance the system's adaptability, focusing primarily on addressing the key uncertainty, reliability, through the adjustment of spreading factors and the minimization of packet loss. These solutions are extensions of the existing adaptation logic, tailored to respond dynamically to changing operational conditions in IoT environments.

Potential Solutions

The first considered solution was a rule-based adaptation logic, similar to the one presented in the paper (Provoost and Weyns, 2019). For Scenario 1, Provoost and Weyns proposed a signal-based and a distance-based approach that incremented or decremented the mote power settings in steps of 1 based on the strongest signal strength and the distance to the gateway, respectively. Transferred to our problem at hand, an adaptation strategy would be to set a threshold for the packet loss. If the *Monitor* step measures a packet loss that exceeds the predefined threshold, the spreading factor would be set to the next value in the 7 to 12 sequence. The advantage of this strategy is that the *Analyze* step is straightforward to implement. In fact, the packet loss value just has to be checked against the threshold. Yet, we decided against this strategy because it does not generalize well to different scenarios. Given different numbers and distributions of motes, the parameter of an acceptable threshold would have to be chosen very differently. Furthermore, while our approach also picks random spreading factors at the beginning in order to explore the action space, the rule-based adaptation logic is entirely deterministic. If all motes in the environment choose the same adaptation step in every iteration, the spreading factors of the motes would remain equal throughout the simulation, and the packet loss could not be reduced. We considered using the following machine learning based classifiers:

1. **Q-learning**: Unlike the rule-based approach, Q-learning is a model-free reinforcement learning technique. It can optimize the decision-making process by learning the value of taking specific actions in specific states. This approach would involve defining the states of the network based on factors such as current packet loss rates, mote density, and environmental conditions. The actions would be adjustments to the spreading factors. Over time, the Q-learning algorithm would learn the optimal action (i.e., spreading factor adjustment) for each state to minimize packet loss. The advantage here is the ability to adapt dynamically to changing network conditions without needing predefined rules.

2. **Learning Regression**: This method involves using regression models to predict the optimal spreading factor based on historical data. By collecting data on packet loss rates corresponding to different spreading factors under various network conditions, a regression model can be trained to predict the spreading factor that is likely to minimize

packet loss in similar conditions. The strength of this approach lies in its ability to use past experiences to inform future decisions, adapting to changing conditions while being guided by historical trends.

Motivation behind Deep Q-learning

Deep Q-learning was selected for its capability to optimize the choice of spreading factors in response to packet loss in IoT networks. This method allows our system to learn from and adapt to varying network conditions, leading to enhanced handling of spreading factors that directly contribute to reducing packet loss. The focus on spreading factor adjustment, rather than power transmission or other variables, provides a targeted approach to maintaining communication reliability within the DingNet framework.

- **Minimizing Packet Loss**: Packet loss in IoT networks can significantly degrade the performance and reliability of the network. It can be caused by a variety of factors, such as interference, signal attenuation, or congestion. Deep Q-learning is motivated by the need to dynamically adjust network parameters, in this case, the sampling rate of IoT devices, to minimize packet loss. By learning the optimal sampling rates under different network conditions, the system can ensure that the data transmission is reliable.

- **Learning from Network Dynamics**: IoT environments are characterized by their dynamic nature, where network conditions can change rapidly. Deep Q-learning is adept at learning from these changes. It can understand complex patterns and correlations between the sampling rate and packet loss, allowing the system to make informed decisions that adapt to the evolving network conditions.

- **Long-term Optimization Strategy**: Deep Q-learning is not just about making immediate decisions but also about considering the long-term impact of these decisions. By continuously interacting with the network and learning from the outcomes, the system can develop strategies that optimize packet loss and sampling rate over time, leading to sustained network performance improvements.

- **Handling Uncertainty and Complexity**: IoT networks, especially those like DingNet, can be complex and filled with uncertainties. Deep Q-learning is motivated by the need to operate effectively in such environments. It can handle uncertainties and complexities in the network by using its learning mechanism to continuously refine its decision-making process.

Design of the Proposed Adaptation Strategy

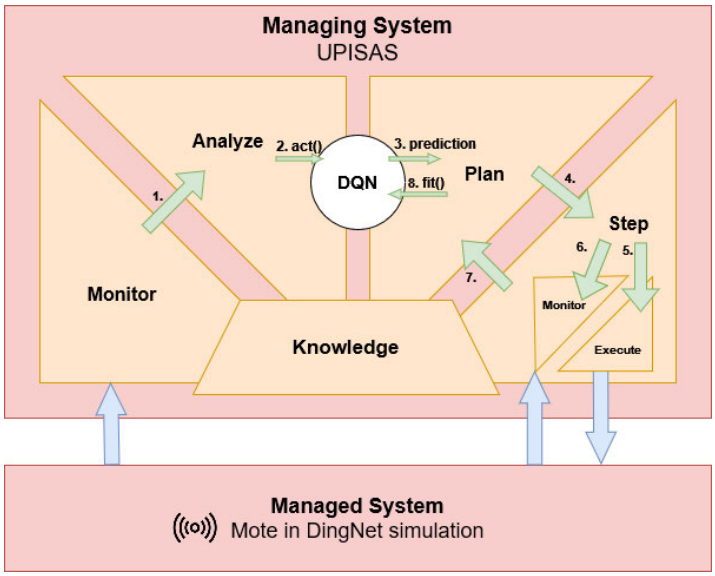

Figure 2. The model of our adaptation strategy.

The MAPE-K model is a conceptual framework for designing self-adaptive systems, which stands for Monitor, Analyze, Plan, Execute, and Knowledge. With the specifics of using a q-learning model, in our case it made more sense to deviate a bit from this model. Instead, our model's "plan" phase consists of an "execute" and a second "monitor". This way we are able to accurately observe the change in state as a result of the newly executed adaptation plan, which is required to compute the reward and fit the model. For our adaptation strategy we're assuming we only have access to whichever information is available to a single mote when we do a call to *monitor*. So, the environment consists only of the environment of a single mote.

- **Monitor**: Utilizing *run.py*, our system monitors the data in the DingNet simulation for each step in the simulation using the monitor endpoint.
- **Analyze**: The function *analyze* in the *dingnet_strategy.py* script constructs a state based on the monitored data. The state is used for the model to calculate a new value for the spreading factor, and this is achieved with the *act* function. If the newly calculated spreading factor and the old value are the same, no adaptation is needed and **Plan** will not be called.
- **Plan**: Upon recognizing a need for adaptation, our system performs a **Step** in the simulation using the newly calculated spreading factor. The reward we get back from **Step** is used to update our target model using the previous and new state, which is done by the *predict* function. This produces a reward from the target model. The rewards from the model and target model, and the new state are then used to fit and train the model.
- **Step (Execute + Monitor)**: After the new value for the spreading factor has been decided, a step in the simulation is executed with the new spreading factor. The monitor

endpoint is then called afterwards and a reward is calculated using the previous and new state. This reward is used to fit and train the model.

- **Knowledge**: Our knowledge repository is collectively maintained by *run.py*, *dingnet_strategy.py*, and *DQN.py*.
- **DQN**: DQN is the component that stands for all the logic behind the Q-learning model. It provides functions to create models and target models, to calculate rewards, and to train and fit the model.

Implementation

In this section, we will go over our implementation that realizes self-adaptation in DingNet. We split the implementation into two different parts: the managed system and the managing system. The managed system details what changes we had to make to DingNet to achieve self-adaptation. The managing system describes the implementation of the Q-learning model we use for self-adaptation.

Managed system

The managed system consists of the DingNet exemplar. We have made several changes to the exemplar so that the managing system is able to interfere while DingNet is running. The simulation in DingNet now runs indefinitely so that we can keep on monitoring data. By doing so, we also included a function that stops the simulation. This gives us the opportunity to stop the simulation and change its configuration to mimic changes in the environment. We also changed the manner in which the simulation state is updated. Instead of updating the simulation state in every step of the simulation, we now only do so when the monitor endpoint is called. This reduces the amount of redundant updates to the simulation state when there are no (significant) changes in the environment. Similarly, instead of calling *moteProbe* every iteration to obtain information about the motes, we now call this only when the main mote sends new packets, which is every 10 iterations. The result of which is stored in the mote itself for it to be retrieved later through the environment in the monitor endpoint.

Additionally, we created a new simulation besides the *MainSimulation* we had previously called *ScatteredSimulation*. This is the simulation we will be using for our evaluation in the evaluation. It sets up an environment programmatically based on a random *seed*. The seed can be provided by the user when calling the */start_run* endpoint as a query parameter, e.g. */start_run?seed=4* will run the simulation using seed 4. Since this simulation is used for the evaluation, more details on what the seed affects can be found in the evaluation section.

Furthermore, we added a GUI similar to the *MainGUI* or *MapView* provided initially by the exemplar but extended or modified the icons and functionality with our own requirements. We included new icons for motes with numbers from 7 to 12 to illustrate the current spreading factors of the motes (*GUI/MapViewer/mote-x.png*) as well as their “special” counterparts for highlighting a single mote (*GUI/MapViewer/mote-special-x.png*). This required some modification to the painter class(es) provided by the exemplar (e.g., *MoteWaypointPainter*). To show the GUI, we changed our docker file to use egalberts’s DingNet image as a base instead of OpenJDK, which boots up a Linux VM with a display that can be streamed over VNC to a web browser. Our updated docker image runs our .jar file and boots up the VM so the GUI can be shown.

Managing system

For the managing system, we use the reinforcement learning algorithm Q-learning. This algorithm learns by assigning rewards to actions in a certain state and aims to pick the action with the highest reward. Our implementation is spread over three files in the UPISAS framework: *run.py*, *DQN.py*, and *dingnet_strategy.py*.

The implementation starts in *run.py*. This file is responsible for maintaining the simulation in DingNet. When called, the *do_run* function starts the simulation in DingNet with a certain seed defining the configuration of the simulation. It enters a loop for a number of iterations, where it starts with calling the monitor endpoint. The function *analyze* (found in *dingnet_strategy.py)* is called to check the monitor results and derive whether adaptation is needed. If it is needed, the *plan* function is called (found in *dingnet_strategy.py*) which goes over the plan for adaptation. At the end of the loop, we make sure to wait so that the adaptation can take place before we monitor again. Once the loop has iterated 50 times, the simulation is stopped. The seed for the configuration is then changed, and the simulation continues again with this new seed. The function *do_run* can also be called without adaptation. This simply runs the simulation without any adaptation. *run.py* also plots the graphs of the packet loss rate and spreading factor over the number of iterations for both runs with and without adaptation.

The *dingnet_strategy.py* file is responsible for analyzing monitor results, executing the adaptation, and supplying the current state of the simulation to the Q-learning model. The function *analyze* is initiated from the file *run.py*. It starts off with obtaining the current state of the simulation. Based on this state, the model decides what the new value of the spreading factor should be. If this new value is equal to the previous spreading factor, no adaptation is needed. The *plan* function is called when adaptation is needed. The current state of the simulation is retrieved and a step in the simulation is taken using the *step* function, which calls the execute endpoint with the new spreading factor value calculated in *analyze*. This function then monitors the new state and returns this state along with a reward based off of the previous and new state. Aside from the model that predicts what actions need to be taken, there is also a target model that keeps track of the actions that we actually want the model to take. After we take the step in the simulation, we continue back in the *plan* function. The target model predicts a target using the current and new state. This target and the previously calculated reward are then used to fit and train the model. Subsequently, the target model is trained.

Finally, there is the *DQN.py* file, which contains the logic behind the Q-learning model. With this file, we can create the model and target model to realize the Q-learning model. The *act* function uses the model to calculate the new value for the spreading factor. Similarly, the *predict* function calculates the target the model should aim for using the target model. *fit* and *target_train* are used to fit the data to the model and to train the target modelby interpolating between its current weights and the weights of the model with a predefined interpolation weight. The Q-learning model's inner workings are inspired by the work of Yash Patel[1].

[1] https://towardsdatascience.com/reinforcement-learning-w-keras-openai-dqns-1eed3a5338c

Implementation process

A big part of what went into the start of the implementation process is explained in detail in the reflection, because we worked on a lot before having to scrap a lot of it. Nevertheless, for the implementation of the managed system (DingNet), we first of all wanted to create a new simulation scenario that would showcase the effectiveness of the adaptation strategy. However, it was quickly becoming confusing to see where the mote was moving and where the other motes were since we had removed the GUI for the first assignment. Therefore, one of the first things we did was bring the GUI back and upgrade it to where it can show the current value that a property of the motes is set to.

In the meantime, we started researching how to implement a Q-learning model in python and created the foundations for it in UPISAS. However, we would not be able to test it properly until we had completed the testing scenario in DingNet, so for a while we were working very much sequentially on the required parts of the project.

Finding a good scenario for adaptation took a while, however, with lots of back-and-forth between us and the professor to discuss what approach we should take. Eventually, we were at a stage to test the adaptation strategy, fix whatever bugs popped up, and collect results for the report.

Showcase and Evaluation

Simulation Scenario

To evaluate our implementation, we constructed a new simulation in DingNet where, in the construction of the environment, we randomly distribute n = 20 motes across a square map of 2000 meters by 2000 meters. These motes are assigned a random spreading factor between 7 and 12 inclusive, which is the allowed range for spreading factors in DingNet. We placed four gateways for the motes to communicate with - one at each corner of the map. Lastly, the environment consists of a “main” mote in the center of the map. This is the mote we will be adapting.

The simulation continuously performs simulation steps in a loop. Every tenth iteration, the main mote sends a packet to all of the gateways. Each of the other motes has a one in ten chance to send a packet each iteration. Therefore, the randomly placed motes, on average, send one packet every 10 iterations. The randomness was included to ensure that there is a consistent chance that they will send a packet at a similar time that the main mote does, to ensure that packet collisions are consistently happening. To ensure consistency between runs, we added a seed parameter that is used for the randomization function to ensure repeatable results. This parameter can be provided in the ‘/start_run’ endpoint such that we can test the same scenario with and without adaptation in UPISAS. Everything in the scenario that requires randomness uses the provided seed to determine the random values. This means that the positions of the randomly scattered motes around the map, as well as their spreading factor, are also determined by the seed. Therefore, by providing a different seed, we can construct an entirely new scenario. See Figure 3 for an illustration of an example scenario created using this method.

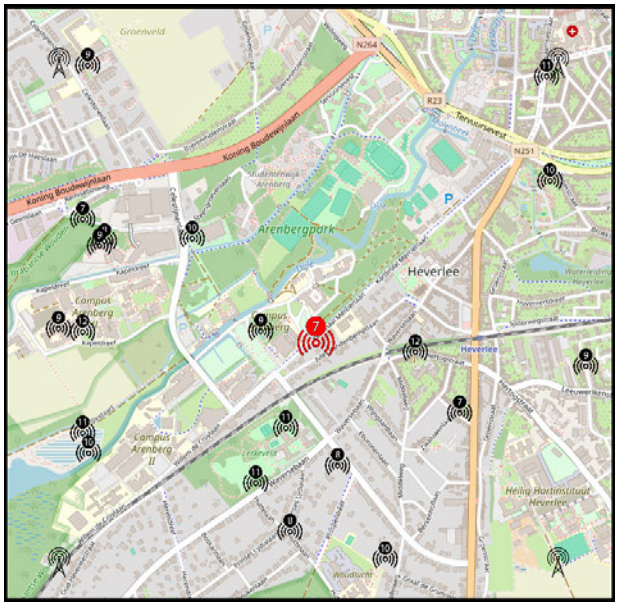

Figure 3. Example simulation scenario. The red mote is the mote to adapt, whereas the black motes are the randomly distributed motes with a randomly assigned static spreading factor.

Evaluation

Setup

For the evaluation, we decided on a baseline scenario where the main mote has a spreading factor of 11. Note that any static value would work for the baseline. We created an evaluation run in UPISAS that performs 100 iterations, where the duration of each iteration was normalized to 1.25 seconds to ensure they are the same length for the run with and without adaptation. In each iteration, the Q-learning-based adaptation goes through the steps of MAPE-K to determine the next spreading factor to select and train the model in the meantime. This is explained in more detail in the section on implementation. As for our baseline, for every iteration, we only call the monitor endpoint to create a history of the packet loss over time for analysis. To evaluate the adaptation strategy against multiple scenarios, after 50 iterations, we restarted the simulation with a different seed. As explained earlier, this creates an entirely new scenario, which we use to evaluate if our model can adapt to changing environments.

Results

In this section we present our results through a series of figures and numerical results. Figure 4 shows the packet loss and spreading factor over time for the baseline. Figure 5 shows the same but for the Q-learning adaptation strategy. Figure 6 directly compares the obtained packet loss of the baseline and the Q-learning adaptation approach. Lastly, Table 4 shows the final achieved packet loss by the adaptation strategies.

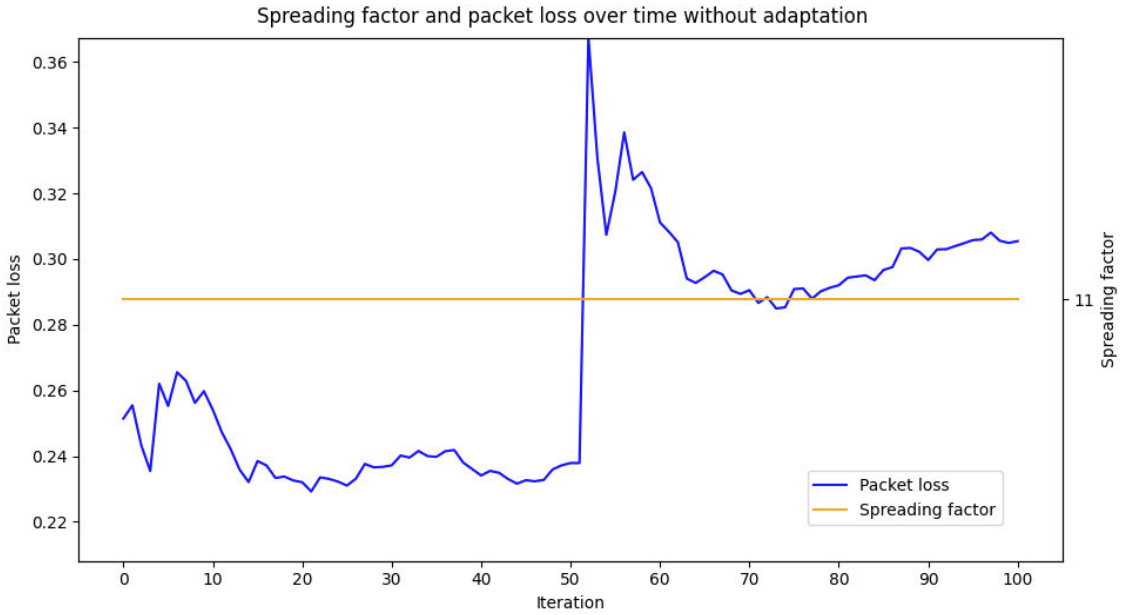

Figure 4. Packet loss and spreading factor over time for the base case (no adaptation).

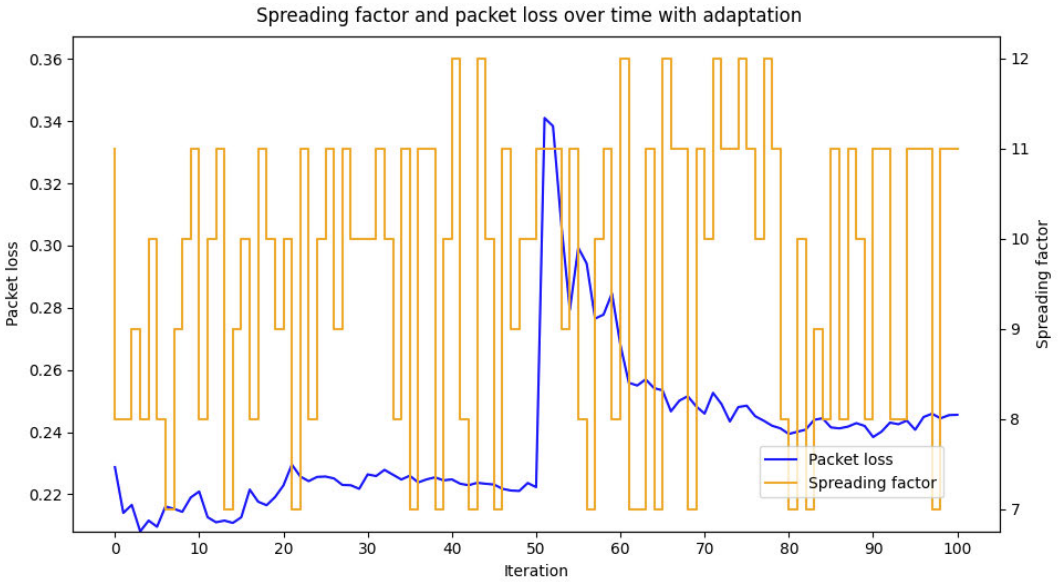

Figure 5. Packet loss and spreading factor over time performing the Q-learning adaptation strategy.

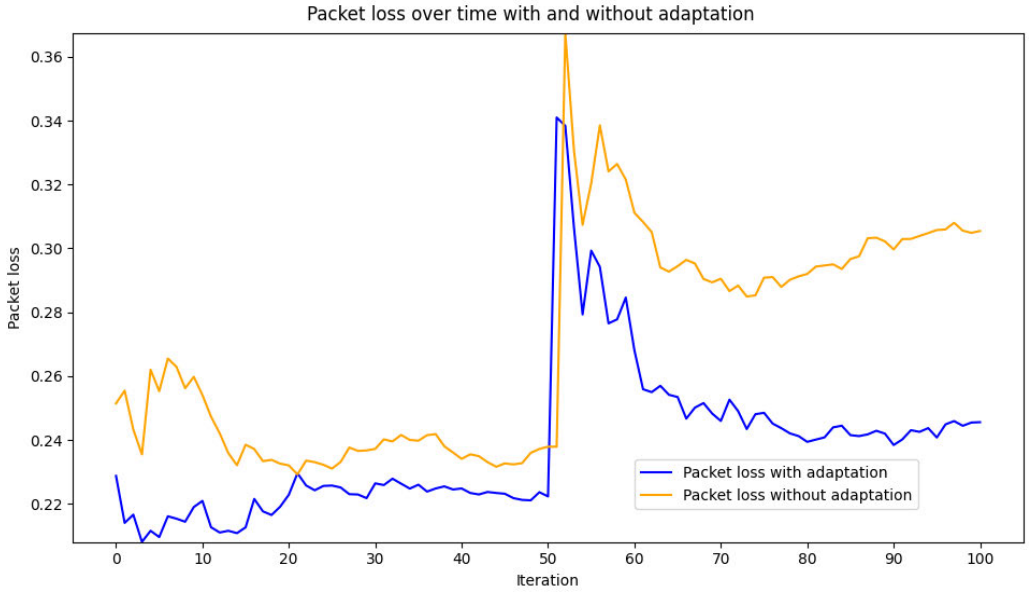

Figure 6. Packet loss over time with and without adaptation (base case).

Adaptation approach	Packet loss
No adaptation, spreading factor 11	≈ 30.5%
Q-learning, packet loss reward	≈ 24.6%
Theoretical best	≈ 11.3%

Table 4. The packet loss obtained by the adaptation strategies. Lower is better.

Results interpretation

As indicated by Table 4, the Q-learning approach reduces the packet loss by about 5.9% compared to the base case. As illustrated by the figures (Figure 4 and Figure 5), the Q-learning approach obtained a lower packet loss in both of the explored scenarios - before the 50th iteration and the newly generated scenario after that. The change in scenario is clearly visible in the figures because the packet loss jumps up significantly at iteration 50. This behavior happens for both the baseline and the Q-learning approach. This behavior can be attributed to the low amount of samples that the packet loss is calculated from at such an early stage in the simulation. After a couple of iterations, the packet loss settles down for both approaches due to there being a bigger sample size and, thus, a reduction in noise in this data (law of large numbers). However, what is interesting to note is that the Q-learning approach appears to reach a much lower packet loss in this scenario than the baseline, and it reaches it much more quickly.

Even though the Q-learning model appears to be performing better across the board, it is difficult to draw any concrete conclusions from these results. We state this primarily because the behavior of the Q-learning model, as shown in Figure 5, appears rather erratic. It appears the model is not able to converge to a specific value. This is partly due to the intentional random choice every couple of iterations to ensure the model jumps out of local minima, but the change in spreading factor occurs too frequently for that to be the primary contributor. We observed in DingNet that for this particular seed, the best spreading factor would have been 7 or 12 for the first scenario and 12 for the second scenario (post 50th iteration). We can see that the Q-learning model picked 12 more often in the second scenario than in the first, but the first scenario shows no particular preference over any of the possible spreading factors. The reason it performs better than the baseline is that the baseline value of 11 is suboptimal. While it is not the worst choice for the spreading factor in either of the scenarios, there are values that lead to lower packet loss. The erratic behavior of the Q-learning model means it attempts a wide variety of spreading factors, which results, on average, in a better result than the baseline value. In fact, had it picked the optimal value for both scenarios (7 in the first and 12 in the second), then the packet loss would only be around 11.3%, as summarized in Table 4. Obviously, it is not reasonable to assume a model will instantly pick the optimal value, and it is possible that had we let the model train for longer, it would converge. However, due to time constraints, we were not able to test that theory.

The theoretical best is an interesting case to consider. For our experiments, we adapted a mote exclusively from the mote's point of view". We assumed the mote could not observe, for example, the spreading factors of other motes. If a mote *would be* able to observe other mote's spreading factors, or if there were some overarching managing system that manages all the motes, then it would be possible for the motes or the overarching managing system to select the spreading factors for the motes such that there is the least amount of possible contention. We assumed this was not an assumption we could make here, and thus we focused on adapting a single mote. However, such a managing system could be interesting to experiment with in future work since it would still require adaptations when, e.g., motes come in and out of range.

Video

A video showcasing our adaptation strategy is available here: [redacted]

Reflection

Key take-aways from this project

In this project, we learned how to apply what we discussed in the lectures about self-adaptive systems. We went through the process of analyzing the adaptive system DingNet and turned it into a self-adaptive system. To do this, we analyzed the uncertainties present in DingNet and formed requirements about how the self-adaptive system should tackle these uncertainties. We learned how to analyze potential solutions for self-adaptation and express the most appropriate solution in terms of a MAPE-K loop. We learned how to incorporate a machine learning method into the self-adaptation of DingNet. Lastly, we managed to critically analyze the results of our self-adaptation implementation and form strong conclusions about them.

Challenges in adaptation strategy selection

In our adaptation strategy proposal, we proposed an adaptation strategy where we would attempt to minimize a mote's power usage while keeping the packet loss as low as possible. This idea was inspired by the work of M. Provoost and D. Weyns in their paper *DingNet: A Self-Adaptive Internet-of-Things Exemplar*. The authors presented two adaptation approaches that both adapted the power setting of the motes, namely a distance-based strategy and a signal-based one. The distance-based approach lowered the mote's power setting the closer it got to the gateway (in steps of 25 meters), whereas the signal-based one increased the power setting if the signal strength of the mote's packets fell below -48, and decreased the power setting once it got above -42. The baseline was a mote without adaptation and the maximum power setting of 14. Their results show that both approaches reduced both the energy usage *and* the packet loss.

After investigating how a packet loss can occur in the DingNet code, we found these two possible options:

1. The signal strength/power of the sent packet is sufficiently low that it is unable to be received by the gateway. This check is implemented in the *packetStrengthHighEnough* function of *NetworkEntity*.
2. Two packets collided. Collision is checked by the *collision* function of *NetworkEntity* and can occur when:
    a. Two packets have an equal spreading factor,
    b. they have similar transmission power, and
    c. they were "in the air" (i.e., traveling towards the destination) at the same time.

In all of our attempts in the DingNet exemplar, using various scenarios and various power settings, spreading factors, and other tests, we were unable to make a packet unable to be received by the gateway (option 1). The threshold for receiving a packet is so low that for a packet to have a sufficiently low signal when it arrives at the gateway that it cannot be received, the mote and the gateway would have to be multiple tenths of kilometers apart. Creating a map

of this size takes nearly a minute only to initialize, and running a simulation where we have to iterate multiple thousands of times per packet to reach the gateway is not feasible. Therefore, it seems probable that all of the packet loss in the paper's results can be attributed to collisions. If this is true, then it kind of defeats the purpose of doing a distance-based or a signal-based adaptation approach because no matter what you set the transmission power to, the gateway can receive the packet unless there is a collision. The only way a power-based adaptation approach can change the packet loss is by preventing collisions by ensuring that no two motes have a similar enough power setting that the *collision* function returns true, which appears to be what is creating the reduction in packet loss for the author's adaptation approaches: no matter which transmission power the adaptation approach chooses, as long as it is "far enough" away from the other mote's power setting of 14, it will reduce the packet loss. And, trivially, any setting different than the baseline causes a reduction in energy usage. In fact, with a power setting of -1 all of the packets would still reach the gateway with sufficient power remaining, and the packet loss would be 0% because the power setting is not similar to the power setting of any of the other motes (that aren't being adapted).

We only came to this conclusion after implementing the Q-learning model for adapting power usage with a reward based on the packet loss (lower is better) and transmission power (lower is better). As expected, our model started a picking transmission power of -1, because any transmission power different than the other motes is guaranteed to reduce the packet loss, and -1 is the lowest possible transmission power and thus reduces the energy cost to the minimum. To us, this was not a satisfying enough result, and thus, we did not choose to go with this as our adaptation approach. This did, however, take a considerable amount of time to discover and implement, which took away time from our actual implementation, which was a challenge.

As discussed in the previous sections, we ended up going with an adaptation strategy that adapts the spreading factor of the motes as a means to reduce the packet loss. The initial idea was to make a mote that moves along a track where a large amount of other motes are randomly distributed along the track with random spreading factors. The goal was that the mote would adapt its spreading factor as it was moving along the track to avoid contention with the spreading factors of the motes currently closest to it. Figure 7 shows the setup for this scenario during our testing phase. A single adaptive mote would run along the track and adapt its spreading factor. All the motes would send packets to the same gateway in the top-right of the map. Unfortunately, this idea did not work out either, because of a similar problem to the one discussed above. Namely, the motes on one end of the track would have collisions with motes on the opposite end of the track. Therefore, since every mote could collide packets with the mote moving along the track, the position of our adaptive mote had no impact. The only way to make a scenario where it would, would be to create a sufficiently large map that the distance between start and end point of the moving mote's track that they can not interfere. But creating a map of this size is too computationally expensive. This is why we went for a stationary mote where the mote can collide its packets with all other motes with the goal of finding the spreading factor that the last other motes have, which would reduce packet loss.

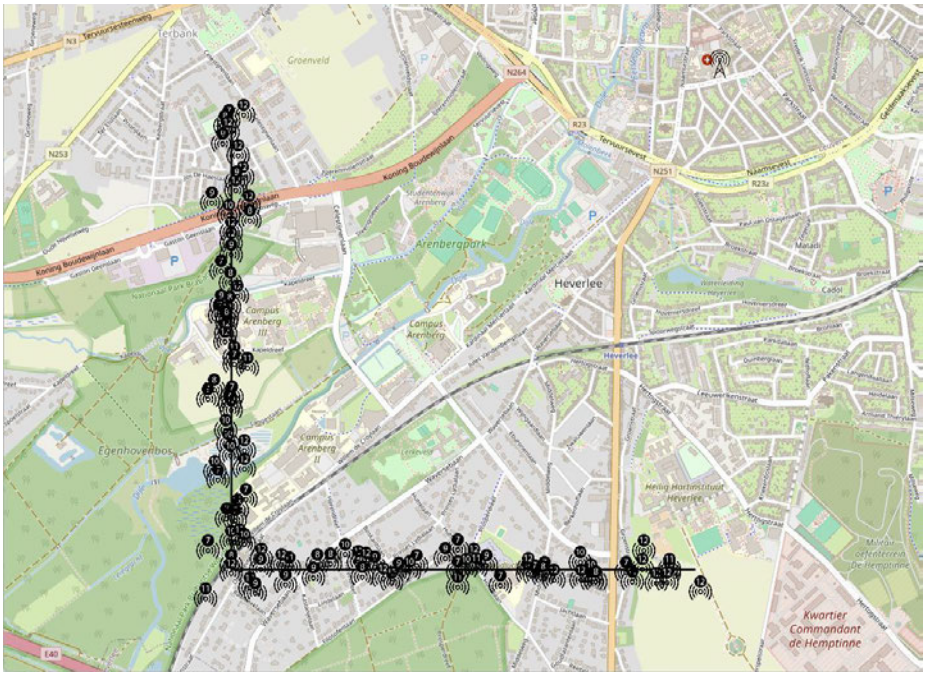

Figure 7: Initial idea for a spreading factor adaptation scenario

In conclusion, deciding on the adaptation strategy took a lot of time and was a major challenge for us during the project. Looking back, we could have prevented this by going with a simple rule-based adaptation strategy, similar to the work presented by the authors of the DingNet paper. However, we feel that reinforcement learning is much more interesting for this purpose.

Overall assignment challenge

Another challenge during the project was the requirements for the 3rd assignment itself. We felt that, for this assignment in particular, it was rather difficult to work on the project together in parallel. This is mainly because of the required implementation steps during the assignment. For this assignment we had a list comparable to the one below of our tasks at hand:

1. Decide on the adaptation goal after realizing adapting transmission power would not really work, as described in the section about our challenges in adaptation strategy selection.
2. Create a new simulation with a scenario that allows testing of the determined adaptation.
3. Implement a Q-learning model in UPISAS that implements the adaptation strategy.
4. Test and evaluate.

Almost none of the above steps could be performed in parallel. In addition, most of these steps are difficult to work together on. This made it quite challenging for us to collaborate on the implementation a lot of the time.

Alternative Implementation paths

Rule-Based Implementation

An alternative implementation path we considered was a rule-based adaptation strategy. In this approach, the system would operate based on predefined rules that trigger specific actions when certain conditions are met, such as adjusting spreading factors based on a set threshold for packet loss. For example, if packet loss exceeds a certain percentage, the system would automatically alter the spreading factors of motes to reduce interference.

However, we decided against this path due to scalability issues in diverse operational scenarios. The main challenge with a rule-based system lies in its lack of adaptability; a fixed threshold for packet loss that might be effective in one scenario could prove inadequate or excessive in another, particularly given the dynamic nature of IoT environments. In IoT networks, where conditions vary widely - from the density of motes to environmental interference - the inability of rule-based systems to adapt to these varying conditions limits their effectiveness.

Furthermore, maintaining and updating a rule-based system as network conditions evolve can be labor-intensive, requiring constant monitoring and manual rule adjustments. This lack of flexibility and the high maintenance cost led us to seek more adaptive and scalable solutions, like Deep Q-learning, which can learn and adjust to diverse and changing conditions without the need for predefined rules.

ML-based Predictive Model

Another alternative implementation path we considered was the ML-Based Predictive Model. Such models use historical data to train algorithms to predict future states and make decisions accordingly. The key advantage of predictive models is their potential to use past patterns to inform future outcomes. However, the implementation of ML-Based Predictive Models presented several challenges that influenced our decision to choose Q-learning instead.

Firstly, ML-Based Predictive Models are heavily dependent on the availability of large, historical datasets for training. Acquiring such datasets that are representative of the diverse and continually changing IoT environments presents a significant hurdle. In IoT settings, where the data can vary drastically due to numerous unpredictable factors, assembling a dataset that accurately captures this diversity is not trivial.

Furthermore, the effectiveness of predictive models degrades over time as the environment changes—a phenomenon known as model drift. This necessitates frequent retraining with new data to ensure the model's predictions remain accurate and relevant. Such retraining requires continuous data collection and processing, which can be resource-intensive and may not be feasible in real-time operational contexts (Evidently AI, 2021).

Q-learning, by contrast, offers a learning paradigm that does not rely on historical data. Instead, it learns from the environment in a real-time, online manner, making decisions based on immediate feedback from the system's current state. This approach is inherently more adaptable to the dynamic nature of IoT environments. It can continuously refine its strategy based on live data, leading to more accurate and timely adaptations.

The need for real-time adaptability and the challenges associated with data collection and model retraining in ML-Based Predictive Models made Q-learning a more practical and effective choice for our system. This is corroborated by the literature that discusses the necessity of retraining ML models to maintain their performance, especially in domains where the operational conditions can rapidly change.

Division of Work

[redacted]

References

- *M. Provoost and D. Weyns, “Dingnet: A self-adaptive internet-of-things exemplar,” in 2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and SelfManaging Systems (SEAMS). IEEE, 2019, pp. 195–201.*
- *Lecture 2 of the “Fundamentals of Adaptive Software” course taught at Vrije Universiteit Amsterdam (2023)*
- *Evidently AI. (n.d.). To retrain, or not to retrain? Let's get analytical about ML model updates. Evidently AI. Retrieved from https://www.evidentlyai.com/blog/retrain-or-not-retrain*
- *Wikipedia. Law of large numbers. Retrieved from https://en.wikipedia.org/wiki/Law_of_large_numbers*