NTNU
Innovation and Creativity

USING NEURAL NETWORKS FOR IOT POWER MANAGEMENT

Finn Juliuus Stephansen-Smith

Submission date: June 2020
Responsible professor: Frank Alexander Kraemer
Supervisor: Frank Alexander Kraemer

Norwegian University of Science and Technology
Department of Information Security and Communication Technology

Title: Using neural networks for IoT power management
Student: Finn Julius Stephansen-Smith

Problem description:

This project investigates whether neural networks can be used to realize intelligent power management in IoT devices. It takes externally provided trained models and attempts to implement them on resource-contrained IoT devices. Knowledge about real-world limitations discovered in this process, as well as steps for how to overcome them, are the desired results of the project.

Responsible professor:	Frank Alexander Kraemer, IIK
Supervisos:	Frank Alexander Kraemer, IIK, Abdulmajid Murad

Abstract

Most devices in the Internet of Things (IoT) operate with limited battery life. In order to still provide a reliable service, they need to make optimal use of their total available power. This project investigates whether the machine learning technique *neural networks* can be used to realize intelligent power management in IoT devices. In particular, we look at sensor nodes using energy replenishment strategies such as solar panels. Given previous weather data and predicted weather forecast, the neural network produces a policy for how often and how strongly devices should perform their function. The goal is maximizing achieved effect, usually measured as the amount of interesting data gathered, while avoiding battery depletion.

We examine how much energy can be saved using these neural networks. We then compare the energy saved to what is consumed by the implementation of, and inference from, the neural network. Finding the relationship between the total energy cost and the potential energy saved is the major goal of the project. Using these results, we are able to provide feedback to the designers of the neural networks so that they can re-train the network with improved parameters. Thus, our experiments and observations both affirm whether the existing neural networks are implementable on modern IoT devices, as well as providing import feedback for improving them.

Sammendrag

De fleste enheter i Tingenes Internett (IoT) har begrenset batterilevetid. For å likevel kunne være pålitelige, er de nødt til å utnytte batteriet på en så optimal måte som mulig. Dette prosjektet ser på hvorvidt maskinlæringsteknikken *nevrale nettverk* kan brukes for å oppnå intelligent strømforbruk i IoT-enheter. Vi ser spesielt på sensor-enheter som bruker solsellepaneler eller lignende teknikker. Gitt data om tidligere værforhold, samt batteritilstand og potensielt andre parametere, skal det nevrale nettet bestemme hvor ofte og hvor kraftig en IoT-enhet bør utføre arbeidet sitt for å oppnå optimalt batteriforbruk. Målet er å maksimere oppnådd resultat, gjerne målt i mengden data generert, samtidig som man unngår at enheten går tom for strøm.

Vi ser på hvor mye energi som kan spares ved å bruke disse nevrale nettene. Vi sammenligner så denne energien med det som forbrukes av enhetens operasjon, spesielt energien det koster å spørre det nevrale nettverket om svar. Å finne forholdet mellom total energi spart på den eneste siden, og energi tapt via det nevrale nettverket på den andre, er et av prosjektets viktigste mål. Ved å bruke denne kunnskapen kan vi gi tilbakemelding til de som designer det nevrale nettet. Dette vil forhåpentligvis la dem trene nettverket på nytt, og vi kan igjen observere hvordan deres modeller fungerer i praksis. Dermed vil vårt arbeid la oss si noe om hvorvidt eksisterende nevrale nettverk er realistiske å bruke i praksis i dag, samt generere essensiell informasjon for forbedringen og videreutviklingen av dem.

Contents

List of Tables

List of Figures

List of Equations

Chapter 1
Introduction

Chapter 1 gives a brief introduction to our work. Section 1.1 introduces the motivation for why studying power management in the Internet of Things (IoT) is interesting. Section 1.2 then briefly explains how neural networks can be applied to help achieve efficient power management in the IoT domain, specifying the scope of our project. Section 1.3 presents the concrete goals of the project, and gives a summary of the results we found. Finally, section 1.4 gives a brief summary of the chapters constituting the rest of the report.

1.1 Background and motivation

The Internet of Things is perhaps the most rapidly expanding technology today. The number of devices connected to the internet is projected to reach **34 billion** by 2025. It might be intuitive to assume most of these are regular user devices such as laptops or smart phones, which are obviously and visibly popular. However, even when completely disregarding all such everyday tools, the number of IoT devices in the world is 21 Billion – more than half of the total number. In fact, the number of IoT devices is expected to surpass the number of regular ones by 2022 [Lasse Lueth, 2018]. This surprising fact is illustrated in figure 1.1.

Given this explosive expansion, there is a growing need for technology

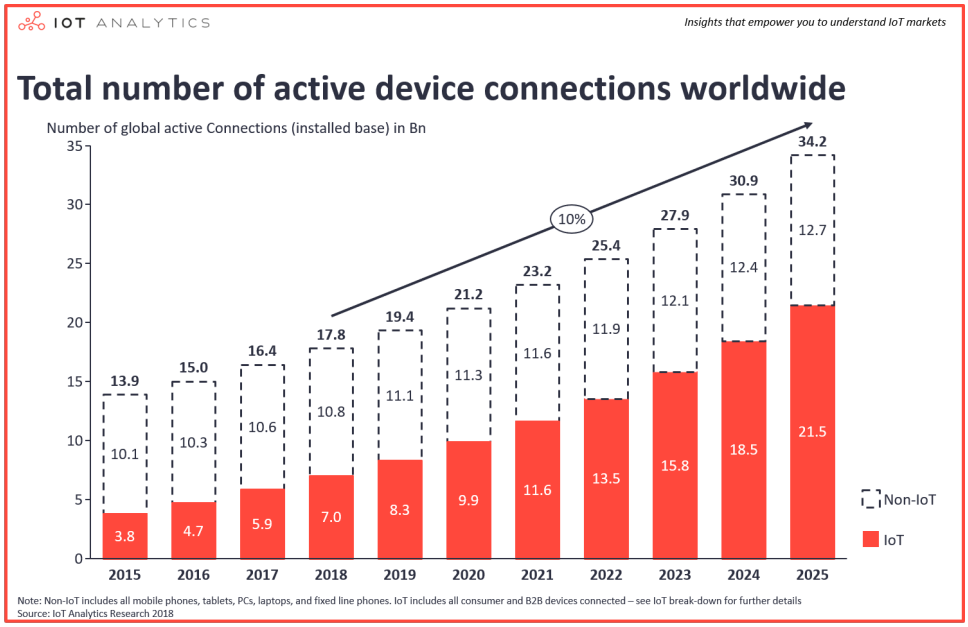

Figure 1.1: The number of devices connected to the internet. Taken from [Lasse Lueth, 2018].

able to cope with this new paradigm. IoT devices are largely heterogeneous in both hardware and software, and they present a range of novel challenges. It is one of these new frontiers we focus on in this report: IoT power management.

Power management means the strategy used to choose a balance between producing output and conserving energy for a device. Poor power management could mean utilizing too much energy too quickly, leading to rapid system failures due to battery depletion. It could also mean erring too strongly on the side of caution, producing much less output that the available power allows for. On the other hand, good power management strikes a balance between the two, maximizing output while maintaining a reasonable energy surplus.

In the traditional era of computers, power management was largely irrelevant. Being connected to a power grid meant a practically unlimited

supply of energy. With the transition into laptops and smart phones however, this changed. Optimizing both hardware and software so as to maximize battery life became essential. Power management became paramount, and the internet of things takes this one step further. Most IoT devices are not connected directly to power, nor do they have the option of temporarily charging that smart phones and laptops utilize. Some examples of such IoT devices include temperature sensors, wildlife monitoring, or even urban applications such as traffic sensors or parking weights. Unlike laptops or smart phones, it is not practical to plug these devices into the grid for charging in regular intervals. Instead, one of two main alternatives must be chosen. One is for manufacturers to simply supply the device with a large enough battery that it does not run out of power for its expected life time. This life time can typically be on the scale of a couple of years, at which point many devices need maintenance or replacement anyway.

Instead of relying on a large battery however, a more long-term, sustainable approach exists. Devices can be supplied with power from other sources than a power grid. Through energy harvesting methods such as solar panels, IoT devices can become fully autonomous even when deployed in difficult conditions. In addition, the sustainable power source means that the device might be able to afford more costly operations, leading to a higher quality of service. An example of this can be seen in figure 1.2.

Energy harvesting techniques come with their own challenges, though. There is a need to consider the variable nature of such techniques. With solar panels, weather starts playing a major role in how the IoT device should perform its functions. This is a largely unprecedented challenge.

The interesting case is the one where the balance between energy coming in through the solar panels and that being spent will be relatively close. Without this being the case, the IoT device consumes too much or too little power for any software or hardware decisions we make to really matter. When there is such a balance present, however, things change. If we let the device perform its function at 100% capacity at all

Figure 1.2: An example of solar panels being used to provide sustainable energy for a deployed IoT device. Taken from [OnL, 2017].

times, it would consume more battery than provided with and fail. Such power failures are undesirable. We also shouldn't turn the throughput down too much either, though; we want as much output from the device as possible given the available power.

In order to achieve this behaviour, various approaches have been suggested. They are mostly based on the idea of selecting appropriate *duty cycles*, meaning what level of intensity the IoT device should perform its function at. The obvious approach to power management is then to write a regular algorithm that produces such a duty cycle. It can for example generate some prediction of how future power input is going to look for a year, then produce a static duty cycle that leads to a net sum of zero power surplus throughout that year. That is, choose a constant level of operation so that surplus energy gathered in the summer is just enough to bring the device through the winter.

This approach has several demerits. For one, it assumes a battery capable of storing enough energy to last the device a long period of time. Second, it is unable to adapt to changing circumstances such

as a particularly dark or sunny year. To improve upon the inherent static, unadaptive nature of such algorithms, machine learning has been proposed as an alternative approach to power management. Specifically, the reinforcement learning technique has proven applicable to this domain [Hsu et al., 2009b]. The idea is to train a machine learning *policy* to accept input such as weather data, then output an appropriate duty cycle. This can be repeated in shorter regular intervals, providing adaptability without interference from developers. The result is a more efficient power management.

This is where neural networks come in. Using neural networks as the driving force behind reinforcement learning, we aim to enable more intelligent utilization of available power. This is different from previous reinforcement learning approaches, where neural networks were not utilized. They allow us to store the trained policy in a more sophisticated manner than the tables or similar data structures previously used in reinforcement learning. This project investigates whether this change leads to more efficient power management in the Internet of Things.

1.2 Problem scope

The memory– and runtime requirements of neural networks have made them unsuitable for the IoT domain for a long time. Only recent advances in the hardware being deployed at the edge has made this interaction possible. This progress is illustrated in figure 1.3.

However, literature on the subject published so far has focused on the more conceptual aspects of the integration. When work has done showing practical results, it has all been done through simulations. To the best of our knowledge, no work has been done showing an actual implementation of trained neural networks on IoT devices to achieve power management. This is the deficit this research aims to remedy. We pose the following Research Question (RQ):

Are we able to utilize neural networks on today's IoT devices

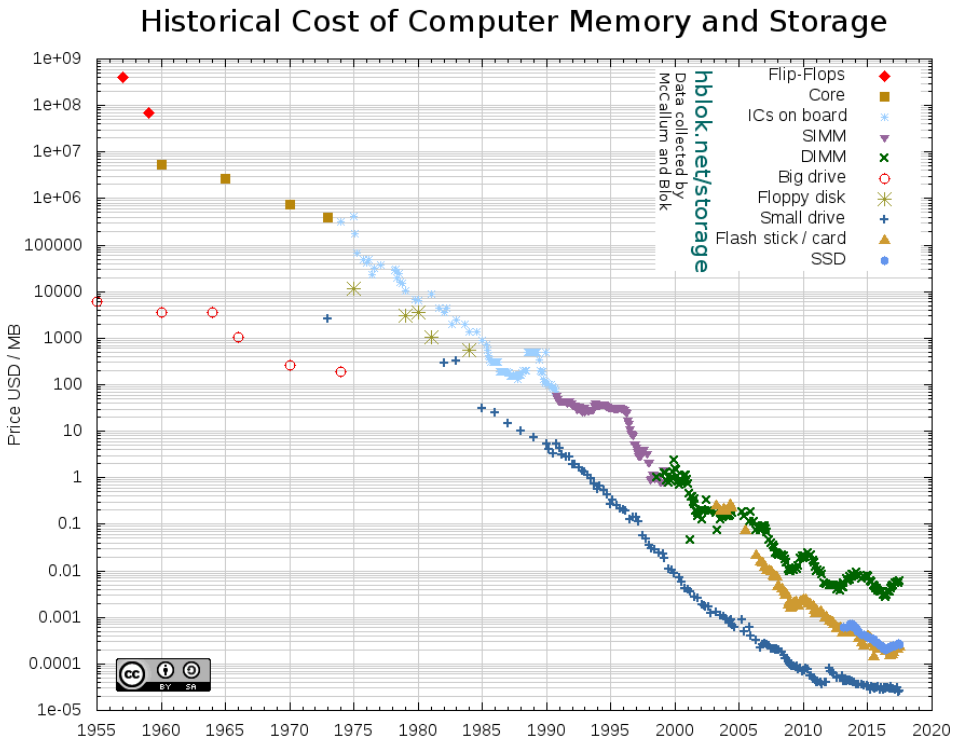

Figure 1.3: The cost of computer memory over time. Taken from [hbl, 2017].

in such a way that they utilize energy better than if they had used previous approaches?

By *utilize*, we mean transferring a neural network model to a device, then successfully performing inference from said model to make some decision. By *today's IoT devices*, we mean modern state-of-the-art devices widely applied in the IoT domain today.

The process leading to an answer to our research question poses a couple of main challenges. In the interest of clarifying exactly which part of the RQ we are attempting to answer at any given point in our report, we pose these implied challenges as their own *secondary* research questions. Secondary research question 1 (SRQ1) deals with the memory limitations of IoT devices.

Do neural networks representing power management policies fit on the restricted hardware of IoT microcontrollers?

In this context, *fit* means two things. First, the static memory size of the neural network should not exceed the flash memory size of a representative IoT device. Second, the runtime memory consumption must not exceed the device's RAM.

Being able to use the neural network is crucial, but it is not enough to answer our main research question. It also asks whether our approach *outperforms* previous approaches. There is a factor playing a role in this that might not be obvious: the transfer and inference from the neural network itself requires power. If our power management is to be efficient, the increased power utilization compared to other approaches must be greater than this consumption. We encapsulate this detail in secondary research question 2 (SRQ2).

Does the neural network-based power management policy consume less energy than it saves by outperforming static algorithms and regular reinforcement learning?

This question implies that our approach outperforms competing algorithms. If it doesn't, the answer to SRQ2 is clearly no. We thus consider the aspect of *being more efficient than competing approaches* from the main research question to be encapsulated here. By including the specific approaches we initially wish to outperform as a proof of concept, this secondary research question helps narrow down the scope of out problem. With these three research questions, the problem scope of our project is thus clearly defined.

1.3 Outcome

The desired outcome of our project is an efficient power management for IoT devices. The intent is for our approach of using neural networks

to outperform the competing power management approaches of static algorithms and regular reinforcement learning. By outperform, we here mean better performance in the following three aspects.

1. Utilize as much of the incoming power as possible.
2. Minimize the amount of resources consumed by the power management system such as memory, CPU, and power.
3. Avoid battery depletion.

The way we determine how well our approach performs in these categories is through **experiments**. Utilizing the design science framework, we perform scientific analysis of the development of an experimental setup. Through measuring the effect of our approach on real hardware, we gain indicative data about how our approach performs in each of the categories above. Comparing this data to the results of previous approaches, we gain knowledge about whether neural networks are a good fit for the IoT domain.

Our results indicate that neural networks indeed are applicable to IoT power management. The initial hurdle we needed to pass was to fit a program utilizing neural networks onto the limited memory capacities of a microcontroller. This challenge lead to the need for compression of the neural network, which brought with it a slight reduction in accuracy. Further, it became evident that special care needs to be taken when developing the surrounding program so as to avoid unnecessary inclusion of libraries or other wastes of memory. However, taking suchs steps eventually led to a successful utilization of neural networks on an IoT device. [TODO true?]

With the neural network in place, the next step was to measure its performance. We found that... [TODO more about results]

1.4 Outline

Chapter 2 is the result of a literature analysis. It provides the theoretical background necessary for our work, explaining key concepts and terminologies. It then looks at previous works in the field, mapping what has already been done and where our work fits in.

Chapter 3 describes how we propose to evaluate our research question. It goes into detail on the environment we wish to create as a testing ground, and it describes how this environment is envisioned to enable the examination of our research question.

Chapter 4 presents the actual developed system. It outlines the steps required to reach the goal outlined in chapter 3. The hardware and software used is presented, the purpose of each part of the system is explained, and the overarching choices made are the defended. In the interest of letting others learn from our mistakes, it also presents some of the challenges overcome during development.

Chapter 5 evaluates the system developed in chapter 4. We aim to conclude whether using neural networks for IoT management is a realistic, useful approach. Analyzing the parameters produced by the experiments will help us reach such a conclusion, and should also help us improve the network itself in the process.

Chapter 6 summarizes the problem and the main findings, and we suggest how to continue the research further.

Chapter 2
Background

Chapter 2 provides background material for the rest of the report, explaining key concepts and analyzing what has already been achieved in the field. Section 2.1 outlines the history of IoT power management, setting the stage for our contribution of neural networks as a new approach in this field. Section 2.2 begins the road to this contribution by explaining the concept of reinforcement learning, the machine learning technique we plan to use. It then looks at Q-learning in particular, a sub-category of reinforcement learning that is highly relevant. Section 2.3 gives a brief introduction to feed-forward neural networks, explaining what they are and how they can be integrated with reinforcement learning in our work. Finally, section 2.4 gives some background the main challenge we expect to face in our implementation: hardware constraints.

2.1 Power management in IoT

A large variety of approaches have suggested to achieve efficient power management in the internet of things. [TODO rewrite to avoid plagiarism from semester thesis] The naive solution, used in a multitude of devices deployed today, is to always "go". If the purpose of a device is to read temperature, for example, this *always-go* policy would involve performing a temperature scan every x seconds. There would be no consideration of current battery level, estimated cost of scan, the likelihood the reading being interesting, or other factors that might influence

whether performing a scan is actually a good idea. To improve upon this, several strategies have been suggested.

2.1.1 Static algorithms

When energy is scarce, more sophisticated methods than *always-go* are necessary. An obvious approach is to write a regular algorithm that takes parameters such as weather history and forecast as input, and produce a so-called *duty cycle* as an output. Duty cycle represents the idea of operating at different levels of intensity, where lower levels would be chosen to preserve energy. As an example, consider an IoT node whose purpose is to perform some scan of its environment. Duty cycle can then be represented as time between scans, the power level at which to run each scan, or other similar definitions. In the case of IoT devices, the choice of whether to perform its action is often a binary one, meaning that we cannot choose to go at, say, 70%. In these cases time between actions is typically chosen as the way to implement duty cycles.

In order to calculate the appropriate duty cycle of an IoT node, there are several approaches available. An intuitive one might write a regular algorithm. Take in historical data, assume the future is going to be similar to the past, and choose a duty cycle that ensures the IoT node does not consume more energy than it receives given this assumption.

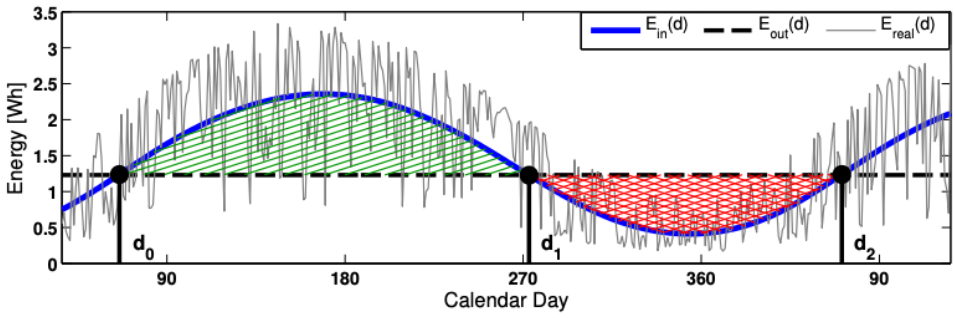

Figure 2.1: One year of solar power availability at a particular geographical location. Taken from [Buchli, 2014].

[Buchli, 2014] is one example of such an algorithm. They produce a mathematical algorithm with inputs as described above, outputting the desired duty cycle. The parameters they observed in one particular experiment is illustrated in figure 2.1. The blue line $E_{in}(d)$ is their expected solar input, extrapolated from historical data. Their algorithm used this to produce the dotted line $E_{out}(d)$, meaning their IoT device ran at a rate corresponding to a constant consumption of around 1.3 W/h. This is designed by the algorithm to build a buffer (green) during summer months that brings the device safely through the power deficit (red) of winter.

Of course, this approach has its demerits. The duty cycle is chosen as a constant value to be used throughout the entire period, meaning it cannot adapt to changing circumstances. It might be more interesting to operate at a high duty cycle during summer than winter for example, depending on what phenomenon the IoT device is actually trying to observe or affect. If the device is a temperature sensor, there might not be any value in performing frequent scans during the night, for instance. In addition, weather patterns might vary significantly from year to year, meaning historical data cannot be trusted. A static algorithm like this is poorly adjusted to dealing with these sorts of challenges. The desire for a more *dynamic*, *adaptive* power management is what inspired the exploration of **reinforcement learning** as an alternate approach. We momentarily diverge from the topic of IoT power management to look closer at this topic next.

2.2 Reinforcement learning

Reinforcement learning is one approach to machine learning. It is based on the idea that when training a machine learning agent, rewarding it for good behaviour should lead to good model. Obviously, this depends on an appropriate definition of what *good* means in the context of a particular machine learning scenario, and this is one of the main challenges faced in the field. A range of proposals for how to determine this have been proposed, but all depend on a common set of definitions. We introduce

these next.

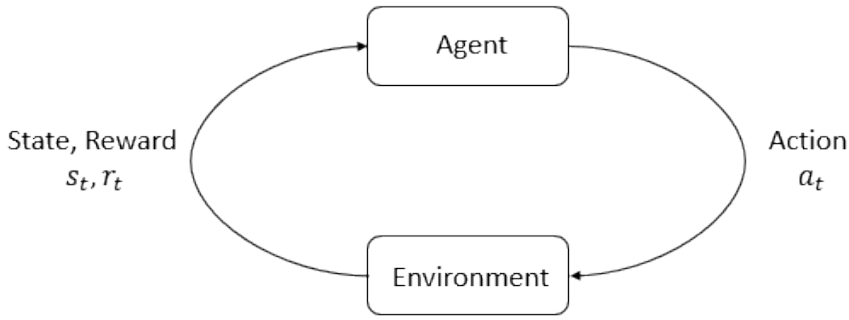

Figure 2.2: The basic structure of Reinforcement Learning. Taken from [**?**].

2.2.1 Key concepts and terminology

The world around the agent is given as state S. This state represents the environment in which the agent is supposed to perform. For example, if you wanted to use reinforcement learning to train an agent to play chess, the state S would represent — intuitively — the chess board. In addition, however, it would include the position of all pieces on the board, as well as which pieces have been taken, etc. In this sense, S can be thought of as containing all information about the world in which the agent exists. In some cases, the agent only sees a limited part of the state. We call this an *observation* of the world, or the agent's *observation space*.

In order to *affect* the world around it, the agent can perform *actions*. We denote this by saying that it performs an action a on state S. In response to an action, the agent typically receives an indication of how good the new world ends up being. By providing appropriate strategies for picking actions, as well as how the *goodness* of the world is calculated, we enable the agent to train. An action leading to a better world state means the agent becomes more likely to perform that action in the future. The result is an agent able to perform well in its given world — exactly the result we're aiming for. Hopefully, this behaviour that works well in our simulated world **also** works well when the agent is put to test in the real world. Only if it is have we successfully used

reinforcement learning to achieve a desirable real-world effect. It is thus we see the importance of what defines *goodness* in the simulated training world: it needs to match what's good in the real world. The element calculating this goodness is called the *reward function*, and choosing or designing a suitable reward functions is both immensely challenging and fundamentally essential in reinforcement learning [Ope, 2018c].

The strategy used by the agent to choose which action to try next is called a *policy*. There are two main categories of policies, *deterministic* and *stochastic*. We focus on deterministic policies, as these are typically more suitable when working with neural networks [Ope, 2018c]. As the policy can be thought of as the brain of the agent, the terms are sometimes used interchangeably. "Training an agent" and "training a policy" typically mean the same thing in the context of reinforcement learning. As the agent trains, its policy is adjusted, and the way it chooses actions adapts. This is achieved mainly by the fact that the policy decides how the agent responds to a reward function. if the reward is good, how fervently should the agent follow the parameters that led to that reward? In the beginning of training, how should the agent test different configurations in search of a good reward? How should it handle convergence? These are some of the questions addressed by the policy.

2.2.2 Q-learning

Q-learning is a category of reinforcement learning approaches that focuses on optimizing the so-called Q-function [Ope, 2018c]:

$$Q^{\pi}(s, a) = E_{\tau \sim \pi}[R(\tau) \,|s_0 = s, a_0 = a] \tag{2.1}$$

Here s represents the state of the world, and a is an action to be taken. R is the reward function, calulated with the given state and action. E_{π} gives the expected return of the term, given that after s_0 the agent chooses actions according to the chosen policy π.

The purpose of the Q-function is to calculate the cumulative reward of the world over time, given that the actor takes some specific action now. The Q-learning technique then uses that indication to evaluate the action it took, updating its policy to reflect how successful the action was. This approach is distinct because it uses an indirect evaluation of actions, looking at how they affect the big picture. The is different from the naive approach, where it simply compares the state directly before and after each action.

The cumulative reward – the result of the Q-function – is calculated as follows. The agent starts in state s_0, and it takes action a. It is this action we wish to evaluate. After the action is executed, the world transitions to state s_1. This state is some degree of better or worse than s_0, as defined by the reward function. After this initial action, the agent chooses all subsequent actions based on generic policy π until convergence. The taken action is then evaluated according to this cumulative reward, and the policy is updated. This is repeated for a user-defined number of steps, after which the agent has hopefully managed to produce a policy that is stable and well equipped to tackle real-world scenarios similar to that used in training.

There are many parameters that need to be specified and adjusted within the Q-learning framework. In particular, the policy used for selecting actions is of critical importance. Other parameters include reward function, number of steps, noise, and more. A large number of suggestions for how to specify many of these parameters exist. These sets of suggestions also often include more radical changes, such as using several Q-learning agents in parallel and comparing them each other for improved training. It is for this reason we call Q-learning a *category* of reinforcement learning. We have a closer look at some such specific approaches next.

2.2.3 Reinforcement learning algorithms

A strategy for how to apply the various Reinforcement Learning aspects and how to specify variables to achieve actual learning is called an

algorithm. An algorithm is no more than a series of steps to take in order to achieve a goal. In the context of reinforcement learning, the term also sometimes encapsulates specifications of the parameters mentioned earlier, such as the reward function.

We look at one algorithm in detail to better explore the concept. Twin Delayed DDPG, or TD3, is one such algorithm. It is a successor to the so-called *Deep Deterministic Policy Gradient* algorithm, or DDPG [Ope, 2018e]. Both algorithms are based on the idea of training both a Q-function **and** the policy directly. When using Q-functions, algorithms normally determine the final policy by using the optimal action in each step. This is given by equation 2.2:

$$a^*(s) = \arg\max_a Q^*(s, a). \tag{2.2}$$

Here a^* is the *optimal* action to be taken in a given state s. It is calculated by checking every possible action on state s, and choosing the one that results in the largest Q-value. All Q-learning algorithms deal with this optimization in some sense, but many do not do so directly. For instance, in many real world scenarios it takes an unfeasible amount of time to test every possible action in every single step of training. If the state is continuous, it is impossible. To combat this, some algorithms approximate a^* by techniques such as *gradient ascent* [Ope, 2018b].

DDPG is one such algorithm tailored for continuous action spaces. It diverges from the pattern of optimizing the agent's behaviour indirectly. Instead, it optimizes both for the Q-value **and** for the action directly in parallel. In fact, it uses one to train the other. The result is an algorithm that has been shown to outperform several competing Q-learning algorithms [Ope, 2018a].

TD3 is a direct successor of DDPG, and improves upon it in three ways. First, it uses so-called *clipped double Q-Learning*, which means

that the way the two trained networks are used against each other is adjusted. Further, it uses a *delayed* policy update. This means that instead of updating its policy immediately after learning the result of an actions, it stores the outcome in a buffer. After seeing the effect of **a couple** of actions, it uses the world view painted by the cumulative set of action results to finally adjusts its policy. Finally, TD3 implements *target policy smoothing*, which is another effort towards the same goal. The goal of all these "tricks" is to solve a single issue: overlearning. While DDPG has generally good results, it has shown a tendency to easily fall into the trap of overlearning. This means that if a certain action gives extremely good results, likely due to some error, the algorithm quickly discards all other options and single-mindedly chases the configuration that led to this erroneously good result. By lessening the importance of a single action's results and instead look at outcomes over time, TD3 improves upon this behaviour.

In summary, we see how a reinforcement learning algorithm can be specified not just as a selection of training parameters, but also as adjustments to the most fundamental aspects of the process.

2.2.4 Reinforcement learning in IoT

Returning to the realm of IoT power management, we look at how reinforcement learning can be used to aid this domain. There are existing works exploring the approach in this field already. These have largely focused on Q-learning algorithms. As a prominent example, Hsu et.al have published a series of work on the topic since 2009 [Hsu et al., 2009b]. Their work is based on the introduction of the term *energy neutrality*, defined as follows:

$$E_{distance\_from\_neutrality} = E_{harvest} - E_{consume} \tag{2.3}$$

That is, the difference from energy neutrality is 0 when the device

consumes exactly as much energy as it receives. We say that it is *energy neutral*. Achieving this means ideal power management. In reality you might want some buffer to ensure the battery doesn't die, but Hsu et.al. among others work with the slightly idealized situation that a perfectly energy neutral device is the perfect, unobtainable goal of power management. With this assumption, they are able to use the definition of energy neutrality to derive mathematical formulas. By attempting to minimize equation 2.3, they can pose the power management challenge as an optimization problem. Specifically, they formulate the reward function of their reinforcement learning algorithm so that a lower energy neutrality leads to a higher reward. With this pretext, they train an agent using basic reinforcement algorithms, iteratively improving their approach in various ways to try to further reduce distance from energy neutrality [Hsu et al., 2009a] [Hsu et al., 2014] [Hsu et al., 2015].

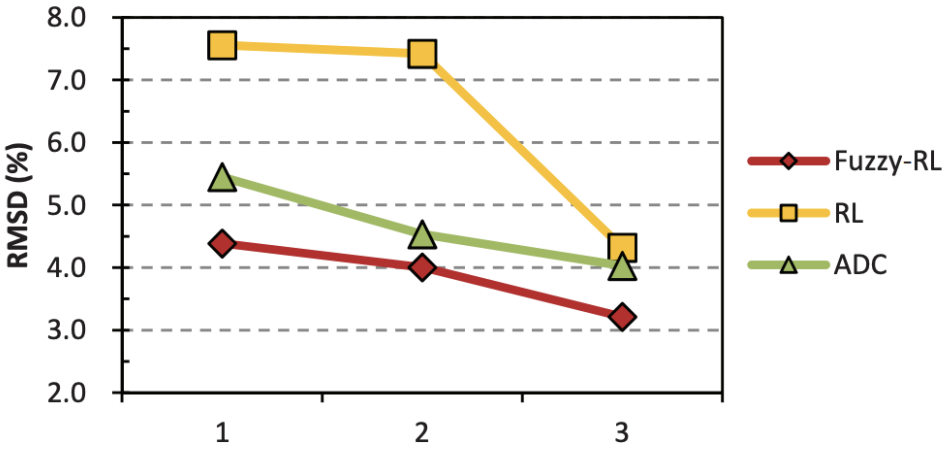

Figure 2.3: Comparison of root mean square deviation of energy neutrality of each month of spring for three comparing methods. Taken from [Hsu et al., 2015].

With each iteration, they show that their results improve compared to previous approaches. This is shown in figure 2.3, where *RL* and *Fuzzy RL* refer to two particular reinforcement earning algorithms they used and *ADC* (Adaptive Duty Cycle) is a static algorithm. We hope to continue and improve upon their work, outmatching them using **neural**

networks as the tool for training the agent. To explain how, we take a closer look at neural networks in section 2.3.

2.3 Feed-forward neural networks

A feed-forward neural network (FFNN) is a neural network in which all information flows in one direction [Schmidhuber, 2015]. This unidirectional nature is illustrated in figure 2.4. Section 2.3.1 introduces the necessary details of FFNNs, while section 2.3.2 describes how this can be used in conjugation with Reinforcement Learning.

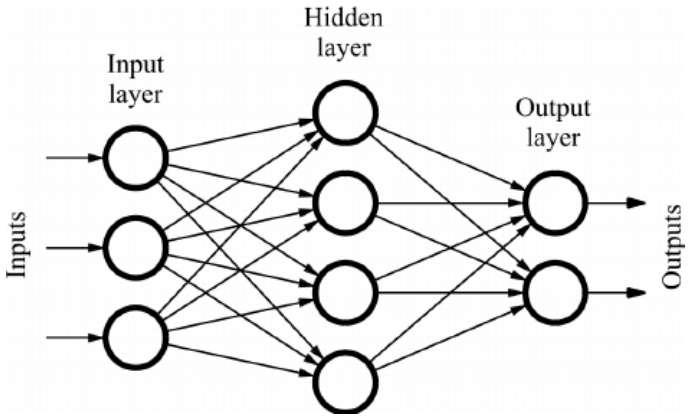

Figure 2.4: Illustration of a feed-forward neural network, in which connections never go backwards. Taken from [Res, 2020].

2.3.1 Neurons and layers

As can be seen in figure 2.4, a FFNN consists of an *input layer*, some amount of *hidden layers* where the training happens, and an *output layer*. The number of hidden layers can be zero. Each layer consists of a number of *neurons*, which act as the processing units of this architecture. The number of layers apart from the input layer is typically denoted *depth*, which would be 2 in the case of figure 2.4. Correspondingly, the largest amount of neurons in a single layer denotes the *width* of the network, in our case 4.

Intuitively, the input layer is where user-submitted parameters are accepted. These are fed forward to the neurons in hidden layers or the output layer. The arrows between nodes represent so-called *connections*, and each connection has an associated *weight*. In each hidden layer during training, several steps are taken to adjust these weights. The weighs of connections between neurons are what define how the NN makes decisions, and adjusting the strength of these in a manner that results in desired behaviour is the purpose of training.

When training a neural network, it is necessary to provide a so-called *activation function*. These are one of the steps taken when adjusting weights. They are typically chosen as a non-linear mathematical functions, a common example being $tanh(x)$. Their purpose is to provide *non-linearity* to the network, which is needed due to the fact that a machine learning model trained linearly has been shown to be no better than a regular linear model [TODO cite]. Choosing different activation functions also affects the actual training of the model, meaning it results in different weights. This makes choosing an appropriate activation function yet another important decision to be made by developers of neural networks.

2.3.2 FFNN in reinforcement learning

Feed-forward neural networks can be used in conjugation with reinforcement learning. When used in this setting, FFNNs are used as the tool for training the agent. The output of training becomes weights of neuron connections, as opposed to something like a simple table of data. These weights can then be used as a function that takes input parameters, runs them through the network with the given weights, and provides the final result of the network inference as output.

There are advantages and disadvantages to this approach. Training neural networks can be a heavier process computation-wise than other approaches, to name one. There is also a larger dependency on knowledge on the side of the developer; the math behind neural networks and the skill required to design an appropriate reward function is far from trivial.

However, there are advantages when compared to traditional approaches as well. By defining the output of training as a set of weights for a neural network, we are effectively able to handle a *continuous* spectre of input. This can be a crucial advantage over discrete outputs, providing increased accuracy and enabling a whole new field of real-world scenarios. This continuous nature also allows a whole range of mathematical tricks and optimizations to improve the training process. These are encapsulated in the practical specifications of reinforcement learning: algorithms.

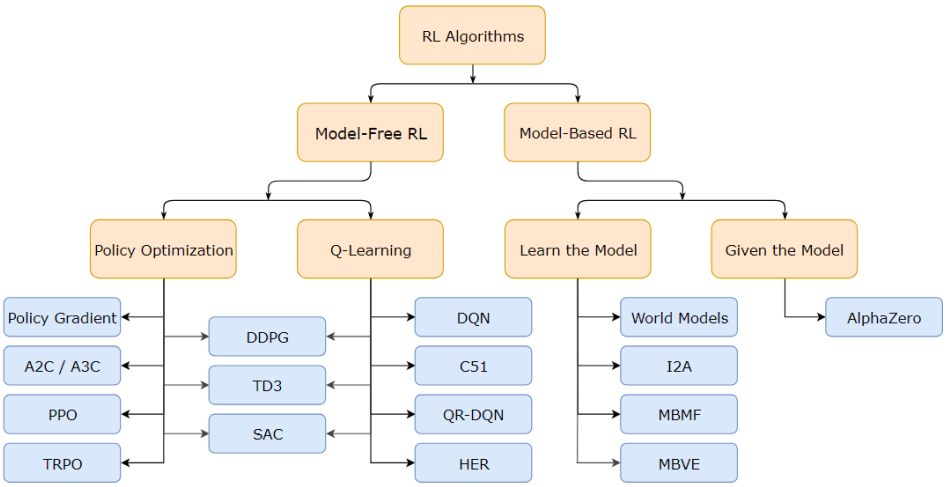

Figure 2.5: A taxonomy of some of the most popular algorithms used in modern RL. Taken from [Ope, 2018d]

A wide variety of algorithms in reinforcement learning have been proposed, and more are being developed every year. Figure 2.5 provides an overview of some of the most common ones used today. We'll look closer the TD3 algorithm, introduced in section 2.2.3. TD3 has several policy strategies available. One, based on convolutional networks ("CnnPolicy"), is mainly used for image processing and recognition. The other major option is caled Multilayer Perceptron policy, or MlpPolicy. Multilayer Perceptron is a class of feed-forward neural networks. It simply means that there at least a single hidden layer. If there are more than a single hidden layer, we say we're dealing with *deep* learning.

It is through the choice of this particular approach that our project becomes focused on neural networks. It is here we diverge from existing works such as [Hsu et al., 2009b], who have attempted achieving our particular goal with reinforcement learning, but without neural networks. To the best of our knowledge, no existing works have used a neural network-based policy to achieve power management in practise. Theory and simulations exist, but actual implementation and resulting real-world measurements do not. We wish to remedy this, and we introduce the main tool used to achieve this practical result next.

2.3.3 Tensorflow

It is necessary with a framework for actually setting up the training and usage of neural networks. One of the most commonly used today is *Tensorflow* [Ten, 2019]. It derives its name from *tensors*, the generalized version of vectors and matrices, because these are what's typically used as input and output to deep learning agents. Tensorflow includes built in code for setting up environments for an agent to training, specifying parameters such as number of training steps, width and depth of the neural network, and more. With their pre-made code, developers can easily convert neural network designs into live agents. In particular, their API for the Python programming language is widely used and well documented.

Figure 2.6: The flow of operation using Tensorflow Lite. Taken from [Ten, 2020].

Even Tensorflow is not a sufficiently specific framework for our pur-

poses, however. Seeing as the goal of our project is to achieve power management in **IoT microcontrollers**, we need ways to fit our trained Tensorflow agents onto the limited hardware capacities of microcontrollers. Tensorflow provides a sub-package for this purpose. *Tensorflow Lite* is a framework specifically made for using machine learning on mobile and IoT devices [Ten, 2020]. It compresses existing Tensorflow models, reducing both their Flash and Dynamic memory footprints. It then applies a technique called *quantization* to further reduce size. Figure 2.6 shows the general series of operations. These processes come at the expense of some model accuracy, but the granularity is chosen carefully so as to minimize the noticeable effect. The result is a machine learning agent with very nearly unchanged behaviour, but requiring vastly reduced hardware specifications.

We have mentioned the limited nature of IoT microcontrollers on various occasions, but we have not yet gone into detail on the limitations we must work with when considering the internet of things. We attempt to remedy this with section 2.4.

2.4 Hardware constraints

Devices used in the IoT are generally limited in terms of hardware capabilities. Memory storage, both static and dynamic, is often in the range of kilobytes. This is a stark difference when compared to modern computers, servers, or other common deployment targets. It is common for developers to be cautious about algorithm complexities, but these restrictive circumstances mean that normally negligible factors start mattering. Examples include which types are used for variables (i.e. float vs double), whether variables are unnecessarily copied due to inefficient function calls, etc. As a result, particular care needs to be taken when developing code for such platforms.

Unfortunately, neural networks are infamous for demanding a large amount of computational resources. This infamy comes largely from the *training* of neural networks, which can take days on even the most

powerful of supercomputers. Performing any sort of neural network training **on** IoT devices is completely infeasible with the sort of hardware specifications on State-of-the-art microcontrollers today. Luckily however, invoking responses from these networks **after** training consumes resources on a scale many orders of magnitude below. This is why it's potentially feasible to utilize neural networks on IoT devices with memory capabilities as small as most microcontrollers.

2.4.1 Memory consumption estimation

Given the restricted nature of microcontroller memories, it would be useful to have a framework for estimating whether our neural network fits. This is the pretense for [Berg, 2019]. In it, Berg provides a model for predicting applicability of neural networks in resource-constrained microcontrollers. Applied properly, this can be used in our work to get an idea of whether our neural networks fits on selected hardware prior to testing. After experimentation and measuring, it can provide insight into **expected** versus **observed** memory consumption. This can help identify outliers in our data and provide context for our results.

Specifically, Berg developed ways to predict three different hardware constraints: static memory, runtime memory, and CPU load. We are not overly concerned with CPU load, or *runtime* as Berg denotes it, as the sensing applications we consider are not particularly time-critical. The runtime matters where battery consumption is concerned, but estimation is largely irrelevant here as measurements of runtimes are simple to make. The same largely applies to static memory estimation: it definitely matter whether we are able to fit a neural network into the static memory of a device, but whether we can or not is easily measured when compiling the program. If we cannot, we know that reducing network depth is the way to reduce the static memory size. We can use Berg's results as an indication of how many Bytes each layer of neurons can save; table 2.1 contains one such reference. The hardware and architecture used in his experiments don't necessarily transfer to our work, which means that there might not be a lot of value in the absolute numbers of bytes presented. However, the difference induced by addition or removal of

layers can be a good reference for ballpark estimation.

Depth	Static size [B]
2	417 752
3	424 168
4	427 800
5	434 088

Table 2.1: Memory static size in Bytes for architectures of depth $2 \leqslant \mathrm{L} \leqslant 5$. Taken from [Berg, 2019].

Out of the three aspects of Berg's work, it is thus mainly the runtime memory estimation that is directly relevant to our work. It can be challenging to measure dynamic memory consumption during runtime [TODO citation on this]. This is especially true for microcontrollers, where the OS is often simplistic enough that there is no explicit indication of a memory overflow. Other architectures might trigger errors such as *stack overflow* or *segmentation fault*, but many microcontrollers simply start producing incoherent output – or none at all [Ard, 2020a]. Thus, it is a useful approach to estimate runtime memory consumption beforehand instead of through measurements. This is where Berg's work comes in. Through experimentation, he finds that the runtime memory consumption of a neural network with a single hidden layer is given by the formula

$$Y = 5x + 5554B \tag{2.4}$$

Y here represents the total memory consumption as output, with input x being the number of hidden layers. With Berg's particular setup, he found that his total available RAM was $\sim$216 KB. Inserting this into the formula and solving for x, he concluded that the maximum number of hidden layers possible was $x = 42183$. We don't intend to push the limits of layer size, but this gives us a solid foundation from which to

investigate runtime memory limits on real hardware.

2.4.2 Energy consumption estimation

In addition to estimating memory footprints, we need to look at the battery consumption imposed by invocation of a neural network. A 2017 paper with the title "Energy Consumption Estimation for Energy-Aware, Adaptive Sensing Applications" [Tamkittikhun, 2019] is of particular interest given our goal of IoT power management. One conclusion we can draw from their work is the following. Given an action for the CPU to perform, the energy consumed by the action depends almost solely on the amount of *time* spent on it by the CPU. In other words, given the time an action takes, we can usually calculate the amount of power the action drains. The accuracy of this calculation depends heavily on whether network transmissions are part of the picture. Such transmissions are often energy-heavy processes. As such, the assumption that power consumption is dependant only on time might not hold if they are a prevalent part of the IoT node's life cycle. The paper looks at this scenario, developing a formula for energy consumption given different power consumption rates for different operations. When these differences are accounted for, energy consumption prediction accuracy can reach levels as high as 97% [Tamkittikhun, 2019]. The formula they pose is as follows:

$$E = \sum_{i=1}^{I} P_i \Delta t_i \tag{2.5}$$

Here i represents a *phase* of an IoT node's *life cycle*. For example, a phase can consist of making some observation through a sensor, or it might be the transmission of a message. P_i denotes the power consumption rate if a given phase. Visually, this rate is indicated by the height of each column in figure 2.7. t_i is simply the amount of time spent in each phase. As a result, the term $P_i \Delta t_i$ is the total power consumption

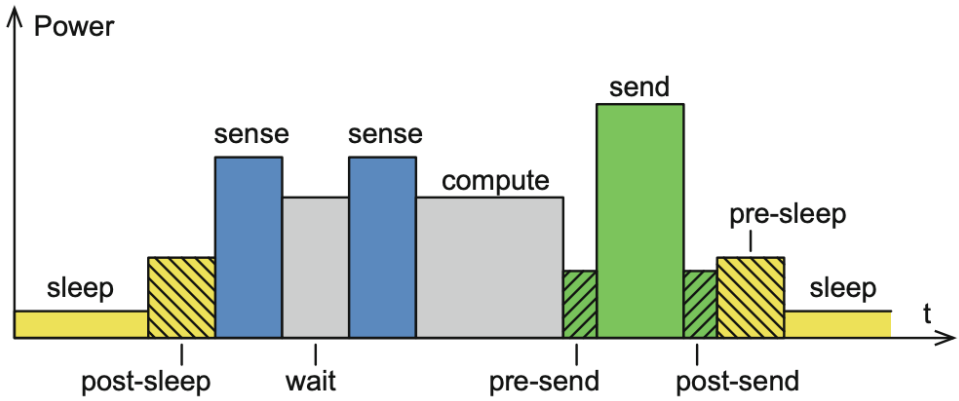

Figure 2.7: An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019].

of phase i. This can be thought of as the area of each column in figure 2.7. E, then, gives us the total energy consumption of the node's entire life cycle by summing the consumption of each phase.

It is worth noting that in the case of near-uniform power consumption per phase, P_i can be considered a constant. In this case, equation 2.5 simplifies to

$$E = P\Delta t \tag{2.6}$$

where P is the energy consumption rate shared by all phases, and Δt is the total amount of time elapsed by the cycle. This is the conclusion we drew earlier about a direct relation between time spent and energy consumed. P can typically be observed as the steady power consumption of a device during regular operation, and we determine it prior to experiments. Consequently, we gain the tools necessary to determine estimated energy consumption of a process using nothing more than the time taken by the process.

These formulas are directly useful for our work. When implementing our neural network with the goal of achieving power management, it will be of crucial importance to know the extra energy consumed by inclusion of the network. This factor will act as a sort of reality check – our neural network solution obviously needs to save more energy than it consumes. For the purpose of learning this consumption value accurately, it will be useful to have methods for predicting and modelling analytically. Measuring energy consumption of a single part of a system directly is challenging, and these formulas allow us to substitute measurements with estimations of high confidence.

2.4.3 Applicability of neural networks in the IoT domain

From the background provided so far, we have two main conclusions. First, we've seen that reinforcement learning has been used for IoT power management previously with good results. Second, we are reasonable sure that neural networks can fit on resource-constrained microcontrollers. If they aren't, we have to tools to find out why and to work towards a fit. The inspiration for our work is the combination of these two conclusions. Our goal is to use neural networks as the force driving reinforcement learning on an IoT device, hopefully leading to better results than previous approaches have achieved.

However, we are not the first to consider this approach. [Murad et al., 2019a] is a work in which a reinforcement learning agent is trained using neural networks, then deployed in a simulated IoT scenario. Their setup is shown in figure 2.8. Specifically, they consider sensing IoT nodes attempting to minimize their distance from energy neutrality. As opposed to using a real device with a solar panel and incoming power, they simulate a power buffer and use historical weather data to provide varying input to the buffer. One example of this is shown in figure 2.9. This way, they are able to analyze how well their network performs in an ideal scenario. Through experiments, they found that the NN approach indeed outperformed competing algorithms in the given scenario. With this,

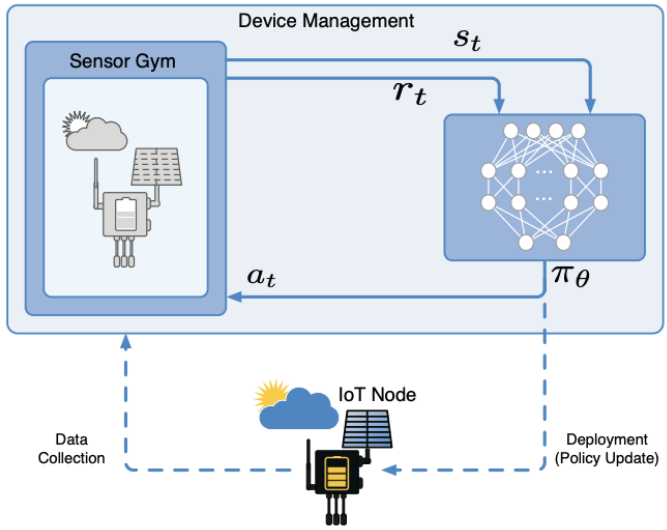

Figure 2.8: The intended agent/environment setup of [Murad et al., 2019a]. The upper parts represent training and invocation from a neural network, while the lower is the updating of the policy of an actual IoT device. This lower part was only simulated in their work. Taken from [Murad et al., 2019a].

they concluded that neural networks are appropriate for the IoT domain, but that further work was necessary in the field [Murad et al., 2019a].

We intend to be part of that further work. Their simulation-based approach has some inherent shortcomings, not least of which is the absence of any actual hardware. By simulating every part of the NN implementation, their works leaves out crucial aspects we have discussed such as memory constraints. What's more, despite the goal being IoT power management, their approach is unable to account for the energy consumed by actually invoking from the neural network. Thus, we can take their work as a reassuring sign that neural networks are indeed applicable to the IoT domain, while leaving plenty of holes for us to fill in our work.

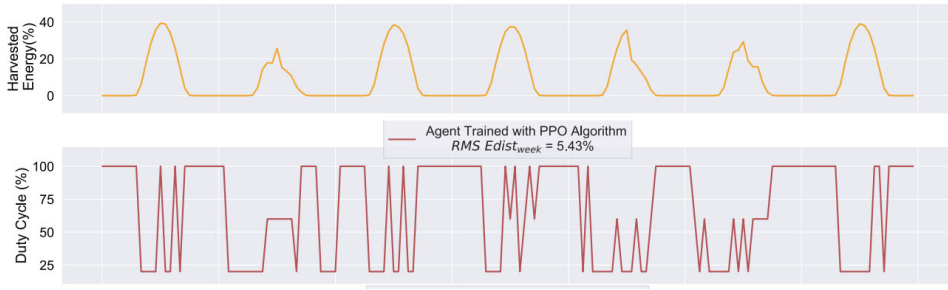

Figure 2.9: Graphs showing simulated solar power and corresponding duty cycle chosen by an agent trained using neural networks. Taken from [Murad et al., 2019a],

Chapter 3
Methodology

Chapter 3 provides an outline of how and what we wish to achieve with this report. Section 3.1 introduces our research question, and provides our reasoning for our selection of parameters. Section 3.2 goes on to provide a framework for how we intend to conduct our research in a manner that enables us to answer our research question. Section **??** does c.

3.1 Research question and context

The goal of our project is to investigate the interaction of neural network and the IoT. Specifically, we want to look at whether existing neural networks can be deployed on State-of-the-art hardware to achieve efficient power management. We pose the following Research Question (RQ):

Are we able to utilize neural networks on today's IoT devices in such a way that they utilize energy better than if they had used previous approaches?

By *implement*, we mean two things. First, the neural network has to be transferred to a microcontroller without exceeding the device's static memory capacity. Second, the neural network must be possible to run inference from at runtime without exceeding the device's runtime memory

(RAM). The process of investigating this part of the research question is quite distinct, and it is useful to clearly distinguish whenever we are concerned with this particular aspect of our work. We thus encapsulate it in a *secondary* research question, secondary research question 1 (SRQ1):

Do neural networks representing power management policies fit on the restricted hardware of IoT microcontrollers?

Going back to the main research question, *previous approaches* refers mainly to the other methodologies presented in section 2.1. These consist of *static* and *dynamic* algorithms, where static means regular algorithms and dynamic are the ones using reinforcement learning. By *utilizing energy better than [those approaches]*, we mean achieving a power management that achieves a lower average distance to energy neutrality. Some other factors such as avoiding battery depletion to avoid burning it out also play a role, and we consider these encapsulated in the term utilize energy better. As explained in chapter 1, we also need to make sure that the energy consumed by inference from the neural network does not outweigh the benefit it grants. To distinguish this process of measurements and comparisons from SRQ1, we introduce secondary research question 2 (SRQ2).

Does the neural network-based power management policy consume less energy than it saves by outperforming static algorithms and regular reinforcement learning?

With that, the final unexplained aspect of our research question is what we mean by *today's IoT devices*. This requires a more in-depth explanation, presented in section 3.1.1.

3.1.1 Choice of hardware

We define *today's IoT devices* as State-of-the-art microcontrollers deemed applicable for the IoT domain. Table 3.1 shows a comparison of a couple of common, highly relevant IoT microcontrollers. It is clear

from the list that the technical specifications of these device are quite similar, indicating that any could be used as a relatively representative device. We wish to avoid hardware-specific conclusions in our report, and we take care to note whenever we do something that would not be directly applicable to other State-of-the-art devices. In the pursuit of this goal, we performed initial testing of setup of neural networks on each of these microcontrollers. It quickly became evident that ARM's **mbed-os** [ARM, 2019] was the prevalent operating system on modern IoT microcontrollers. As a result, we made mbed-os compatibility a requirement for qualification for being a *State-of-the-art* microcontroller.

Device name	**CPU**	**Flash**	**RAM**	**mbed-os**
nRF52840 (Berg)	64 MHz	1MB	256kB	Yes
nRF9160 (NB-IoT)	64 MHz	1MB	256kB	No
nRF52-DK (BLE)	64 MHz	512KB	64KB	Yes
Arduino Nano 33 BLE	**64 MHz**	**1MB**	**256kB**	**Yes**

Table 3.1: Comparison of the most important specifications of various state-of-the-art IoT microcontrollers. Taken from [Semiconductor, 2019], [Berg, 2019], and [Ard, 2020b].

Further, it's worth noting that a wide range of microcontrollers with lower specs than those presented are available. For some use cases, these cheaper, smaller devices can be suitable. However, as presented in section 2.4, neural networks are infamous for have large memory footprints. As such, the intuitive choice was to select a microcontroller among those in the upper end of hardware specifications. To establish whether this intuition was reasonable, we performed initial tests with a microcontroller of lower memory capacities. The nRF52-DK was chosen for this purpose. As can be seen in table 3.1, this microcontroller has half the Flash memory and less than half as much RAM as the other, upper-end microcontrollers. As can be seen in figure 3.1, initial tests indicated a RAM requirement above even the upper end devices' capacity. We knew at this point that a wide range of optimizations were possible, and later chapters of this report show that we were indeed able to get the memory footprint down significantly. However, we used this early

indication to conclude that our intuition that upper-end microcontrollers such as the other three presented were indeed more applicable to this project.

Module	.text	.data	.bss
[fill]	517(+517)	11(+11)	37(+37)
[lib]/c.a	69431(+69431)	2548(+2548)	127(+127)
[lib]/gcc.a	7616(+7616)	0(+0)	0(+0)
[lib]/m.a	968(+968)	0(+0)	0(+0)
[lib]/misc	252(+252)	16(+16)	28(+28)
[lib]/nosys.a	32(+32)	0(+0)	0(+0)
[lib]/stdc++.a	173167(+173167)	165(+165)	5676(+5676)
main.o	1565(+1565)	24580(+24580)	237(+237)
mbed-os/cmsis	1033(+1033)	0(+0)	84(+84)
mbed-os/components	170(+170)	4(+4)	4(+4)
mbed-os/drivers	1040(+1040)	0(+0)	0(+0)
mbed-os/hal	1550(+1550)	4(+4)	66(+66)
mbed-os/platform	3789(+3789)	260(+260)	228(+228)
mbed-os/rtos	7961(+7961)	168(+168)	6289(+6289)
mbed-os/targets	9957(+9957)	4(+4)	856(+856)
models/cifar10_cnn.o	130030(+130030)	4(+4)	1(+1)
uTensor/uTensor	6737(+6737)	28(+28)	7(+7)
Subtotals	415815(+415815)	27792(+27792)	13640(+13640)

Total Static RAM memory (data + bss): 41432(+41432) bytes
Total Flash memory (text + data): 443607(+443607) bytes

Figure 3.1: Screenshot of the output of attempted compilation and transfer of a basic neural network to the nRF52-DK microcontroller.

With mbed-os and approximate RAM and Flash storage capabilities established as requirements, we narrowed down our choice of hardware. THe similar work done in [Berg, 2019] relied on the nRF52840 microcontroller, and we considered this at first. However, the configuration of this device did not work smoothly out of the box, and although we eventually made it work, the modifications necessary were quite hardware-specific. In addition, the device is physically large, making it inapplicable for many real-world use cases. As a result, we looked for a smaller device of similar specs whose setup was known to be relatively hassle-free. The result of the search was the **Arduino Nano 33 BLE**, shown in figure 3.2.

Figure 3.2: Arduino Nano 33 BLE, the physical IoT device we plan to use.

Arduino microcontrollers are known to be designed for ease of use programmatically. Few design decisions should need to be hardware-speficic. In addition, the Arduino Nano 33 BLE specifically is equipped with a range of sensors, making it well suited to be an IoT sensing node. These sensor are fit onto the smallest form factor available: 45x18mm [Ard, 2020b]. Last but not least, it has the memory capabilities and mbed-os compatability established as requirements for being a State-of-the-art microcontrollers. We have thus chosen the Arduino Nano 33 BLE as the hardware for our research.

3.1.2 Choice of parameters

The choice of Arduino as our hardware platform comes with several advantages. First, the Arduino IDE provides tools for easily compiling C++ code into a runnable bundle complete with the underlying mbed-os included [Ard, 2020b]. This makes the path from a high-level program to code runnable on a microcontroller short, and it helps alleviate hardware dependency. As such, we explicitly specify the Arduino framework as one chosen parameter for our project.

The Arduino framework does not include tools related to neural networks, however. In order for a neural network to become small enough to be usable on a microcontroller, we need to perform whatever optimizations we can. The Tensorflow Lite framework is made for exactly this purpose [Ten, 2020]. Tensorflow is one of the most common

frameworks for training and usage of neural networks, and it allows us to load the trained networks provided by

Table 3.2 summarizes our choice of parameters.

IoT microcontroller	Arduino Nano 33 BLE
Development framework	Arduino
OS	mbed-os
Embedding technology	Tensorflow Lite
RL algorithm	TD3
RL policy	Feed-forward neural networks

Table 3.2: Chosen parameters for our project.

3.2 Research method

Before continuing to the specifics of our work, it is worth taking a moment to discuss the method we intend to apply to make sure our project produces scientific knowledge. As we will see, our work is poorly suited to the hypothesis-testing model of the natural sciences. The alternative we use instead is based on **producing** something, and in that kind of work it is easy to lose track of what new knowledge is actually being produced. One might instead fall into the trap of spending an unjustified amount of time making sure the product is as polished as it can be, while the underlying interesting questions being answered are sidelined. Those questions might turn out to already have been answered by previous works, or there might not have been an interesting question there to begin with. In the following, we formulate a plan to avoid this trap.

3.2.1 Iterative design

Our work is based on practical experiments, trying to *produce* something of value. This is different from natural sciences, in which the goal is to

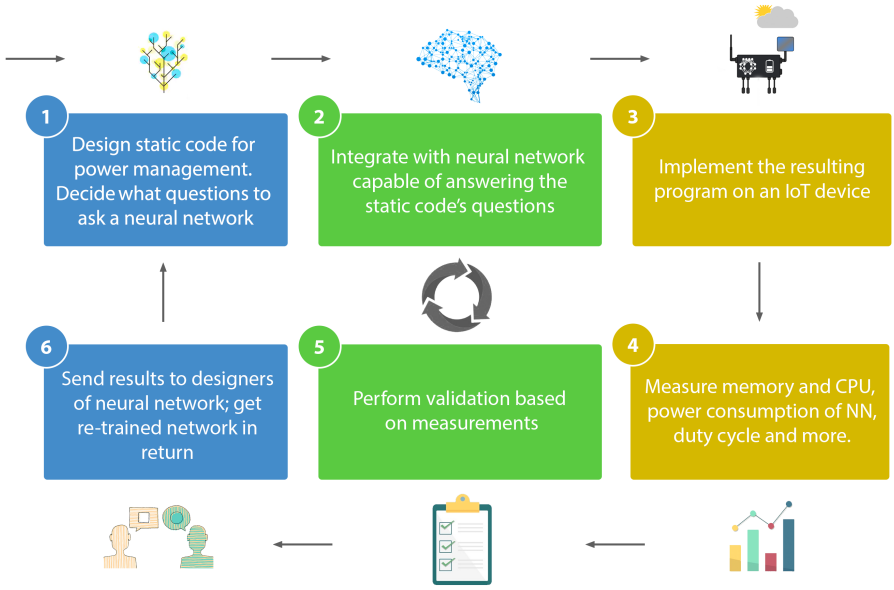

Figure 3.3: The iterative process we will follow for the design and validation of the neural network.

observe the world and figure out how it works – without changing it. Our work is also heavily iterative, reviewing our approach and parameters whenever we hit roadblocks such as a memory shortage. Figure 3.3 reflects this nature. We wish to be constantly iterating so as to provide our external designers time to perform re-training, and we wish for the goal to be a functioning system providing value if possible. If we are to talk about our research method in a generic sense, we need a framework that encapsulates these differences from natural sciences. We also need it to reflect the iterative nature of our work.

Design science is one such framework [Wieringa, 2014]. In it, they define the production of something by humans as *design*. This term is meant as a clear distinction from the passive, non-interfering nature of the natural sciences. Design science, then, is a formal framework how we we can approach this kind of design in a scientific manner.

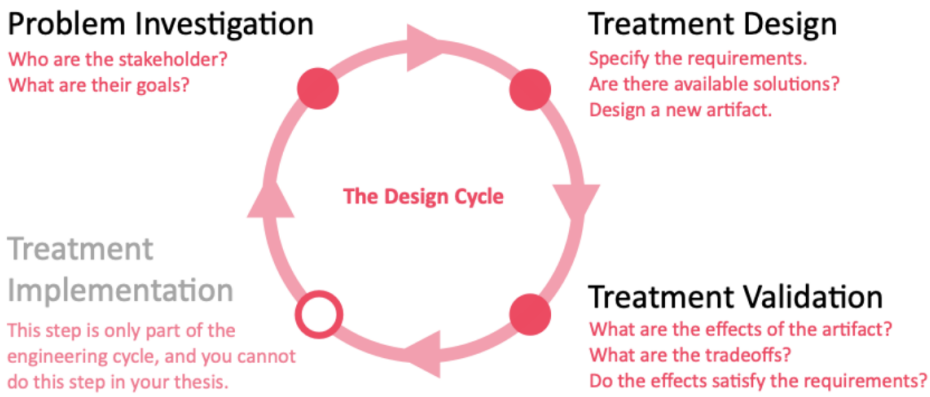

Figure 3.4: The iterative nature of design science. Taken from [Des, 2019].

Figure 3.4 shows the main principles of design science. As is obvious from the figure, the design science cycle consists of four main phases. First, one must identify the goal of the project. What lack of knowledge exists that we can fill? Who does this knowledge create value for, and why? This phase is where the aforementioned "interesting questions" are investigated. Although it is one of the shorter phases, it might be the most important one. Continuing to the next step of the cycle without having properly investigated the problem at hand is a recipe for disaster.

We wish to perform a proper problem investigation. In our work, we identify the *stakeholders* as owners of IoT devices. Specifically, those with a need to deploy their IoT in a setting that does not allow for regular electrical charging, but instead using energy harvesting methods such as solar panels. Their goal, the one we wish to help achieve, is to provide their IoT devices with efficient power management. This means making sure the devices perform as strongly as possible given available harvested power while avoiding frequent battery depletion. This task has been considered by previous works, and incremental improvements have been made. The new scientific knowledge we wish to provide is whether using neural networks can lead to a system performing *better* power management than those approaches. With this, our goal is clearly

established, and the initial problem investigation is complete. We stay mindful of the possibility that the defined goal will need adjustment as new aspects of the work are explored.

The next step is *treatment design*. A *treatment* is defined in design science as a proposed solution to the identified problem. The cyclic nature of design science comes largely from this definition. Each treatment is thoroughly investigated, first analytically and then in practise to whatever extent is realistic. Shortcomings are typically identified. This creates a need for a new proposal – a new treatment. This process repeats, seldom reaching perfection but instead moving closer to a good solution with each iteration.

As for treatment design in our project, we identified the *requirements* and *existing solutions* described in figure 3.4 during problem investigation. Namely, treatments need to be usable on State-of-the-art microcontrollers, and they must provide better power management than previous approaches – existing solutions. These are likely to stay constant throughout our work. Designing a new *artifact* is how the design science framework phrases development of the individual parts that make up a treatment. An artifact might in our case be the program that enables inference from a neural network on a microcontroller, for example. Going back to figure 3.4, steps 1 through 3 encapsulate our main artifacts. The design of these artifacts, as well as investigation of how they interact with their environment and form a treatment, is the main practical contribution of our work. We go into detail on this in chapter 4.

Finally, there is treatment *validation*. This is the step that makes sure we avoid the trap of polishing our product at the expense of scientific knowledge. It is defined as the *prediction* of how treatments would likely perform if deployed in a real-world scenario. This definition is meant to highlight the difference from what's done in step 4 in the cycle, *evaluation*. According to the framework, proper evaluation requires stakeholders to make use of the treatment in real scenarios, at scale. That is, the treatment needs to move from a development phase to

a production phase. As this is unfeasible in many scientific works, validation is introduced as a method of still reflecting on the process and measuring treatments' applicability. You emulate the real-world scenario as best you can, typically through software simulations, and you measure what performance parameters you can. What you cannot measure, you provide insight into or you provide steps for exploring it in future works. This is how a large number of scientific works are built up.

We adapt these definitions slightly. The neural networks we intend to work with have already been validated in a general sense. That is, they have been shown to perform as intended in simulated scenarios [Murad et al., 2019a] [Murad et al., 2019b]. If we were to work with other neural networks as inputs, our predefined parameters would still imply pre-validation as a requirement. We wish to take those networks one step closer to evaluation with our work. Although we will not be able to scale up to the level implied by the design science framework's definition of evaluation, we introduce a large amount of new factors by the inclusion of hardware. We go from a simulated test environment to a practical experiment using real hardware, measuring performance using equipment similar to that used in the market. This allows us to gain confidence about whether this approach to IoT power management is a reasonable one. We perform detailed measurements, making sure to be observant of whenever real-world performance differs from simulated results. With this we are able to observe the effects of taking the neural network solution one step further, identifying trade-offs that become necessary as new constraints are introduced. This is in line with what's outlined in the validation step in figure 3.4. Finally, with a result in hand, we reflect on the initial requirements. In the case that they are not fully satisfied, we might embark on another cycle of the design process. In the end, design science thus lets us not only end up with a product well matched with the initial requirements, but also a detailed trail of the process used to get there. Used together, these form the basis for our report.

3.3 Experiment setup

The foundation for our experiments presented so far are summarized as follows. We wish to investigate whether neural networks can be applied to the IoT domain in order to achieve efficient power management. We imagine a use case in which a stakeholder owns one or more IoT devices, and the devices are in need of a system to let them make intelligent decisions about power output. Importantly, we assume that *the user already has a trained neural network* applicable for this task. Whether this network can fit on a microcontroller is the question we wish to enlighten. We define our *context* as the following:

- The predefined parameters defined in table 3.2
- An externally provided neural network, assumed to have shown efficient power management behaviour in simulations
- Data used as input to the neural network. In the case of the neural network being based on solar input, this data could be historical weather data

We phrase our imagined scenario as a *user story*:

«As a user, I would like to know whether my neural network trained for power management can actually achieve the desired power management behaviour on real IoT hardware, using real data.»

We work with the assumption that our chosen predefined parameters yields representative behaviour of IoT hardware in general. Given this, we wish to perform experiments and measurements that yield whether the inclusion of a neural network for power management purposes actually reduces distance from energy neutrality overall. We formulate these wishes as so-called *feasibility criterion*.

(i) **Feasibility criterion 1:** Verify that a given neural networks can be transferred to, and ran inference from, a given microcontroller.

(ii) **Feasibility criterion 2:** Verify that the resulting behaviour yields a lower distance from energy neutrality than competing approaches.

To check whether these feasibility criterion are satisfied, we produce expressions and conduct experiments that result in *boolean* expressions. That is, they are satisfied or they are not. If we reach the conclusion that any are **not**, we wish to provide detailed descriptions of both why and of what must change in order to reach the desired result. We formulate this wish as the following *proposition criterion*.

(iii) **Proposition criterion 1:** If the neural network does not fit on the given microcontroller, identify what specific constraint is being overridden. Provide explanations for what would have to change, either on the end of the hardware or of the neural network, for transfer and inference to work.

(iv) **Proposition criterion 2:** If the resulting behaviour does not yield sufficiently low distance from energy neutrality, identify why. In particular, differentiate between weaknesses in three distinct aspects: the neural network itself, hardware effects, and our own testing framework. Identify which of these is the culprit, and provide steps for improving it.

In order to provide a comprehensive look at whether each of these criterion is satisfied, and alternatively why not, we need to break them down. Specifically, we apply them to each individual aspect of our work. For each aspect, such as the basic transfer and inference from a neural network, we introduce *sub-criterion* that indicate the status of that particular part of the system. The feasibility criterion are satisfied if, and only if, each part of the system's sub-criterion are satisfied. In the case that they are not, this partitioning helps us give insight into what particular part of the process failed and why. This lets us provide

answers to the proposition criterion. We discuss each part of our system in its own following subsection.

3.3.1 Sense cycle

Sense cycle. Include feasibility / proposition criterion (?).

3.3.2 Neural network on a microcontroller

NN micro. Include feasibility / proposition criterion.

3.3.3 Power management

Power management. Include feasibility / proposition criterion.

Chapter 4
Experiments

Section 4.1 does A. 4.1.1 does B. 4.2 does c.

4.1 Memory consumption

4.1.1 Static sensing cycle

The Arduino Board we've chosen has a program storage space of 983040 Bytes, or approximately 1MB. Initial tests show that the static part of our program, the sensing cycle, takes 292kB. This is approximately 29% of the total available space.

As for dynamic memory (RAM), the maximum is 262kB. Some of this RAM is allocated at runtime, as the compiler is able to identify the size of so-called global variables. In our basic code exmaple, 67 kB (25%) of the RAM is consumed in this manner. This leaves 194kB available for memory allocation during the program's run time. This memory consumption is shown in figure 4.1.

4.1.2 Neural network invocation

The memory imprint of the neural network is largely dependent on the effects of the actual invocation. Thus, the data concerning memory computed at compile time are of limited value when viewed alone. It is

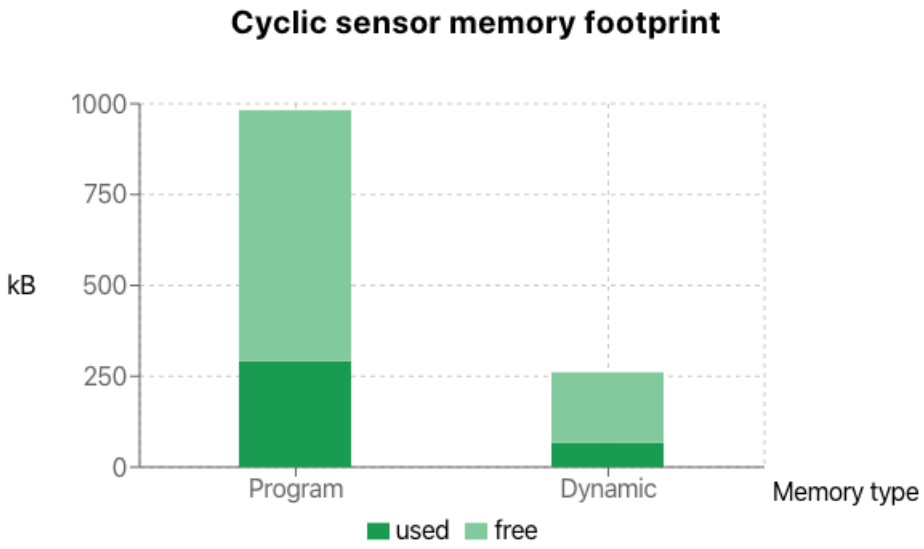

Figure 4.1: The memory consumption of our static program.

still an important starting point, however. The data is shown in figure 4.2.

4.1.3 Total memeory

We want to evaluate how much memory the finished program consumes at runtime. Simply adding the two previous statistics is insufficient, as there is some common overhead when transferring any kind of runnable code. To analyze this, we compile and transfer an entirely empty program with no depenencies:

```
// Empty program used to evaluate memory overhead

void setup() { }

void loop() { }
```

This empty program still has a memory footprint: 76kB (7%) program

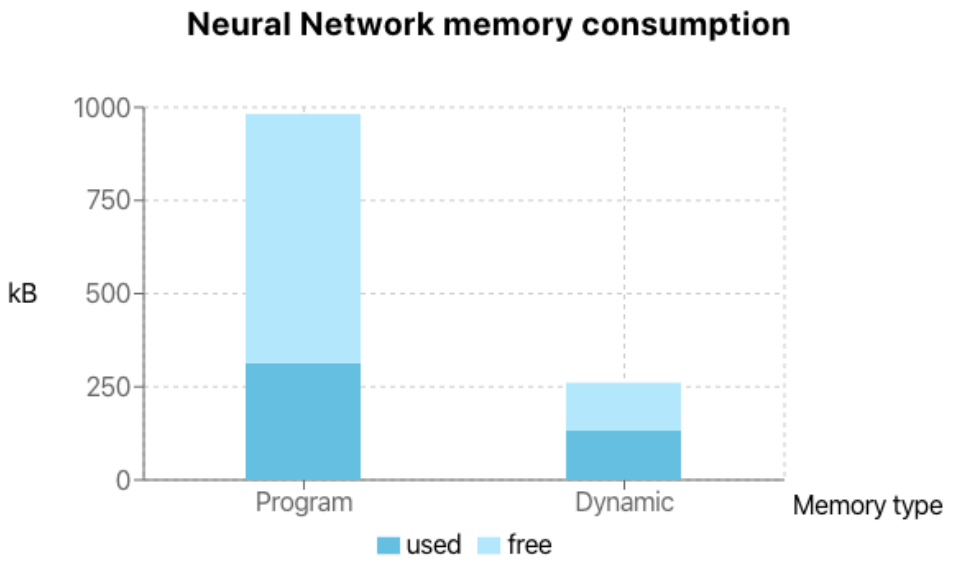

Figure 4.2: The memory consumption of the neural network.

storage space, as well as 42kB (16%) of dynamic memory. We consider this the common, unavoidable memory overhead. Denoting each memory set as M_i and taking the union of this base case and the two previous sizes, we get the following compile-time memory consumption:

$$M_{tot} = M_{sense} \cup M_{nn} \tag{4.1}$$
$$= M_{sense} + M_{nn} - M_{sense} \cap M_{nn} \tag{4.2}$$

Inserting our collected data about program memory into formula 4.2 gives us:

$$M_{program} = 292kB + 314kB - 76kB$$
$$= \mathbf{530kB}$$

Likewise, the collected data for the dynamic gives us:

$$M_{dynamic} = 67kB + 132kB - 42kB$$
$$= \mathbf{157kB}$$

The findings are summarized in table 4.1.

Memory	**Overhead**	**Sense cycle**	**Neural network**	**Total**
Program	76kB	292-76 =216kB	314-76 = 238kB	530kB (54%)
Dynamic	42kB	67-42 = 25KB	132-42 = 90kB	157kB (60%)

Table 4.1: Memory consumption of the various parts of our experimental program.

This percentage of total memory used is shown in figure 4.3. As we can see, around half of the program memory is consumed, while a bit more than half of the dynamic memory is allocated. To be precise, 54% of the program memory and 60% of the dynamic memory is spent.

Further, the distribution of the memory allocation is shown in pie charts. Respectively, figure 4.4 for the program memory and figure 4.5 for the dynamic memory.

This preliminary data seems promising in regards the amount of free memory available. More than 40% free memory is indicates a good likelihood of our successful execution of a neural network in combination with some static code and the inherent overhead. However, there is one crucial parameter not yet analyzed: the runtime memory.

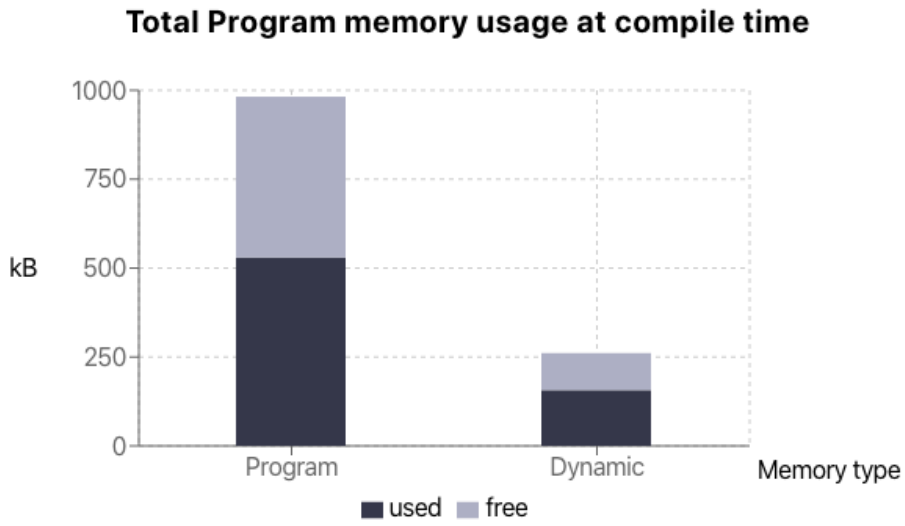

Figure 4.3: Total memory consumption at runtime.

4.2 Runtime memory

The contents of this subsection

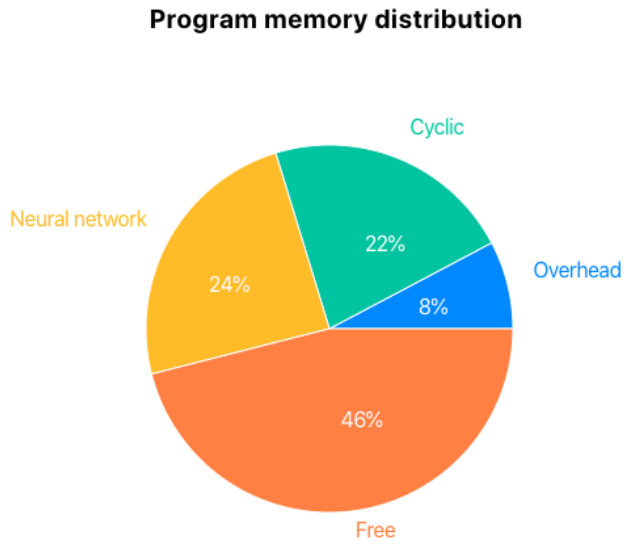

Figure 4.4: The distribution of program memory at compile time.

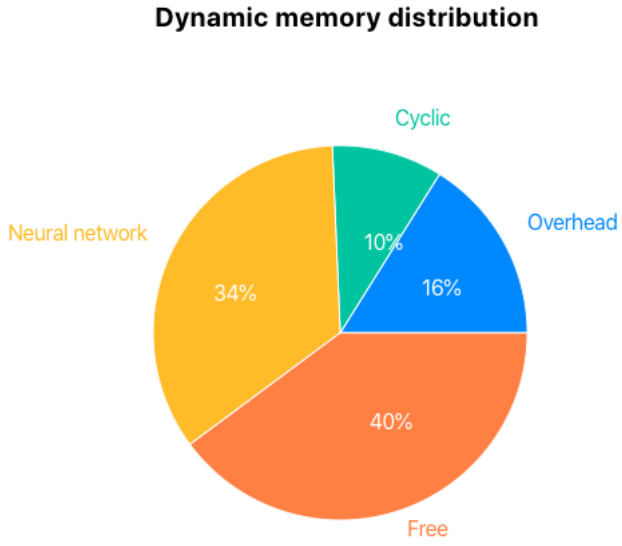

Figure 4.5: The distribution of dynamic memory at compile time.

Chapter 5
Results

Section 3.1 does A. **??** does B. **??** does c.

5.1 Some section

Introduction to the topic

5.1.1 Some subsection

The contents of this subsection

Chapter 6
Concluding remarks

Concluding remarks!

Bibliography

[hbl, 2017] (2017). Historical cost of computer memory and storage. *https://hblok.net/blog/posts/2017/12/17/historical-cost-of-computer-memory-and-storage-4/*.

[OnL, 2017] (2017). Onlogic. *https://www.onlogic.com/company/io-hub/extrovert-iot-contest-winner-lensec-remote-surveillance/*.

[Ope, 2018a] (2018a). Deep deterministic policy gradient (ddpg). *https://spinningup.openai.com/en/latest/algorithms/ddpg.htmlbackground*.

[Ope, 2018b] (2018b). Openai spinning up: Intro to policy optimization. *https://spinningup.openai.com/en/latest/spinningup/rl*$_i$*ntro*3.*html*.

[Ope, 2018c] (2018c). Openai spinning up: Key concepts in rl. *https://spinningup.openai.com/en/latest/spinningup/rl*$_i$*ntro.html*.

[Ope, 2018d] (2018d). Openai spinning up: Kinds of rl algorithms. *https://spinningup.openai.com/en/latest/spinningup/rl*$_i$*ntro*2.*html*.

[Ope, 2018e] (2018e). Twin delayed ddpg (td3). *https://spinningup.openai.com/en/latest/algorithms/td3.html*.

[ARM, 2019] (2019). Arm mbed. *https://www.mbed.com/en/*.

[Des, 2019] (2019). Design science seminar. *https://falkr.github.io/designscience/preparation.html*.

[Ten, 2019] (2019). Tensorflow. *https://github.com/tensorflow/tensorflow*.

[Ard, 2020a] (2020a). Arduino memory. *https://www.arduino.cc/en/tutorial/memory*.

[Ard, 2020b] (2020b). Arduino nano 33 ble. *https://store.arduino.cc/arduino-nano-33-ble*.

[Res, 2020] (2020). Feed-forward neural network overview. *https://www.researchgate.net/figure/Feedforward-neural-network$_f$ig1$_3$29586439*.

[Ten, 2020] (2020). Tensorflow lite. *https://www.tensorflow.org/lite*.

[Berg, 2019] Berg, A. V. (2019). Implementing artificial neural networks in resource-constrained devices. *NTNU Master thesis (?)*.

[Buchli, 2014] Buchli, B. (2014). Dynamic power management for long-term energy neutral operation of solar energy harvesting systems. *SenSys'14*.

[Hsu et al., 2009a] Hsu, R. C., Lin, T.-H., Chen, S.-M., and Liu, C.-T. (2009a). Qos-aware power management for energy harvesting wireless sensor network utilizing reinforcement learning. *IEEE Transactions on Emerging Topics in Computing*, pages 537–542.

[Hsu et al., 2015] Hsu, R. C., Lin, T.-H., Chen, S.-M., and Liu, C.-T. (2015). Dynamic energy management of energy harvesting wireless sensor nodes using fuzzy inference system with reinforcement learning. *IEEE Transactions on Emerging Topics in Computing*.

[Hsu et al., 2009b] Hsu, R. C., Liu, C.-T., and Lee, W.-M. (2009b). Reinforcement learning-based dynamic power management for energy harvesting wireless sensor network. *IEEE Transactions on Emerging Topics in Computing*, pages 399–408.

[Hsu et al., 2014] Hsu, R. C., Liu, C.-T., and Wang, H.-L. (2014). A reinforcement learning-based tod provisioning dynamic power management for sustainable operation of energy harvesting wireless sensor node. *IEEE Transactions on Emerging Topics in Computing*, 2(2):181–194.

[Lasse Lueth, 2018] Lasse Lueth, K. (2018). State of the iot 2018: Number of iot devices now at 7b – market accelerating. *https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/*.

[Murad et al., 2019a] Murad, A., Kraemer, F. A., Bach, K., and Taylor, G. (2019a). Autonomous management of energy-harvestingiot nodes using deep reinforcement learningabdulmajid. *2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*.

[Murad et al., 2019b] Murad, A., Kraemer, F. A., Bach, K., and Taylor, G. (2019b). Iot sensor gym: Trainingautonomous iot devices with deepreinforcement learning. *Proceedings of International Conference on Internet of Things (IoT2019)*.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117.

[Semiconductor, 2019] Semiconductor, N. (2019). nrf9160 dk product brief. *Taken from https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF9160-DK-product-brief.pdf?la=enhash=C37A8EFD5E8CB6DC82F79F81EC22E1473E6447E7*.

[Tamkittikhun, 2019] Tamkittikhun, S. (2019). Energy consumption estimationfor energy-aware, adaptive sensingapplications. *S. Bouzefrane et al. (Eds.): MSPN 2017, LNCS 10566*.

[Wieringa, 2014] Wieringa, R. J. (2014). Design science methodology for information systems and software engineering.