Finn Julius Stephansen-Smith

Using Neural Networks for IoT Power Management

Master's thesis in Communication Technology and Digital Security
Supervisor: Frank Alexander Kraemer
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Dept. of Information Security and Communication Technology

Using Neural Networks for IoT Power Management

Finn Julius 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-constrained 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
Supervisors: Frank Alexander Kraemer, IIK, Abdulmajid Murad

Abstract

Most devices in the Internet of Things (IoT) operate with limited battery life. To still provide a reliable service, they need to make optimal use of their total available energy. This project investigates whether *neural networks* can be used to implement intelligent power management in IoT devices, specifically those using energy harvesting techniques such as solar panels. Given input such as previous weather data and predicted weather forecast, the neural network helps save energy by adjusting the frequency of devices' operations based on available energy.

We investigate a wide range of such neural networks, looking at how network structure affects both memory footprint and power consumption. Our first finding is the limit at which neural networks become too large to fit in the memory of a typical IoT device. We found that these limits were well beyond the size of existing neural networks designed for IoT power management. We next examine how much energy can be saved using these neural networks. Of course, running an inference from a neural network is itself an operation that costs energy. By comparing the energy spent to the energy saved, we get an idea of when a neural network is worth including. We ran experiments on our neural networks to find when, and why, they were able to break even on energy. Applying this insight to our specific power management neural network, we again found that the network fell well within our estimated bounds. We thus conclude that neural networks indeed seem applicable to the IoT power management domain.

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 *nevrale nettverk* kan brukes for å oppnå intelligent strømforbruk i IoT-enheter, spesifikt de som høster energi via solcellepaneler. Gitt data om tidligere værforhold, samt batteritilstand og potensielt andre parametere, reduserer det nevrale nettverket totalt strømforbruk ved å justere hvor ofte enheten kjører sin funksjon basert på hvor mye energi som er tilgjengelig.

Vi ser på en lang rekke nevrale nettverk, med fokus på hvordan nettverksstørrelse påvirker både minne- og energiforbruk. Vårt første resultat er grensen for hvor stort et nevralt nettverk kan bli før det ikke lenger passer i minnet til en IoT-enhet. Disse grensene ble observert som langt større enn størrelsen til et eksisterende nevralt nettverk trent for smart strømforbruk. Vi ser deretter på hvor mye energi som kan spares ved bruk av disse nevrale nettverkene. Naturligvis representerer bruken av et slikt nevralt nettverk også et energiforbruk. Ved å sammenligne energi spart med energi brukt kan vi få en idé om når et nevralt nettverk er verdt å inkludere. Vi kjørte eksperimenter på våre nevrale nettverk for å finne når, og hvorfor, de klarte å gå i null energimessig. Ved å bruke denne innsikten på vårt spesifikke strømforbruks-nettverk fant vi igjen at nettverket falt godt innenfor de estimerte grensene. Vi konkluderer dermed at nevrale nettverk virker passende for bruk i IoT-strømplanlegging.

Preface

Thank you, Frank Alexander Kraemer, for frequent and excellent guidance throughout the project.

Thank you, Abdulmajid Murad, for your technical input.

A special thank you to Amund Askeland, without whom several core technical challenges of the project would still stand unresolved.

Contents

List of Tables XII

List of Figures XV

List of Equations XIX

Symbols XXI

Acronyms XXIII

1 Introduction 1
- 1.1 Background and Motivation . . . 1
- 1.2 Problem Scope . . . 4
- 1.3 Results . . . 7
- 1.4 Outline . . . 9

2 Background 11
- 2.1 Power Management in IoT . . . 11
  - 2.1.1 Static Algorithms . . . 11
- 2.2 Reinforcement Learning . . . 13
  - 2.2.1 Key Concepts and Terminology . . . 13
  - 2.2.2 Q-learning . . . 14
  - 2.2.3 Reinforcement Learning Algorithms . . . 15
  - 2.2.4 Reinforcement Learning in IoT . . . 16
- 2.3 Feed-forward Neural Networks . . . 17
  - 2.3.1 Neurons and Layers . . . 18
  - 2.3.2 FFNN in Reinforcement Learning . . . 19
  - 2.3.3 TensorFlow . . . 20
- 2.4 Hardware Constraints . . . 21
  - 2.4.1 Memory Consumption Estimation . . . 21
  - 2.4.2 Energy Consumption Estimation . . . 23
  - 2.4.3 Applicability of Neural Networks in the IoT Domain . . . 24

3 Methodology 27
- 3.1 Research Question and Context . . . 27
  - 3.1.1 Choice of Hardware . . . 28
  - 3.1.2 Choice of Parameters . . . 29
- 3.2 Research Method . . . 29
  - 3.2.1 Iterative Design . . . 30
- 3.3 Experiment Setup . . . 33
  - 3.3.1 Sense Cycle . . . 34
  - 3.3.2 Neural Network on a Microcontroller . . . 35

3.3.3 Power Management 36

4 Experiments **39**
4.1 Sense Cycle Implementation 39
4.1.1 Memory Consumption 40
4.1.2 Energy Consumption 42
4.2 Resource Consumption of Neural Networks 45
4.2.1 Procedure 46
4.2.2 Memory Consumption of a Single Network 47
4.2.3 Memory Consumption Boundaries 50
4.2.4 Compile-Time Memory 52
4.3 Power Management Implementation 58
4.3.1 Total Memory 58
4.3.2 Total Energy Consumption 62

5 Discussion **67**
5.1 Fitting Neural Networks into Device Memory 67
5.1.1 Memory Required by a Sense Cycle 68
5.1.2 Fitting a Neural Network in the Remaining Memory 68
5.2 Power Management Performance 70
5.2.1 Energy Budget of an IoT Device 70
5.2.2 Energy Saved by the Neural Network 72
5.3 Case Study 76
5.3.1 Evaluation of Externally Provided Neural Network 76
5.4 Research Question Revisited 79

6 Concluding Remarks **83**

List of Tables

1.1 The width limits of a neural network given different depths, assuming they are required to fit onto a 1024 KB Flash memory. 7

2.1 Memory static size in Bytes for architectures of depth $2 \leqslant L \leqslant 5$. Taken from [Berg, 2019]. 22

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]. 28
3.2 Chosen parameters for our project. 30

4.1 Memory consumption of our sense cycle application in isolation. 41
4.2 Runtime of the different parts of our developed sense cycle, measured over 5 iterations. 42
4.3 Runtime of the different parts of our developed sense cycle, this time when integrated into the larger project. 44
4.4 Final estimations of CPU runtime of the various parts of a sense cycle program. 45
4.5 The parameters used in the neural network. 47
4.6 Memory consumption of our initial neural network in isolation. 48
4.7 Memory consumption of our initial neural network in isolation, measured during runtime embedded on an Arduino Nano 33 BLE microcontroller. 50
4.8 The selected limits of neural network size throughout our experiments. . 51
4.9 Memory consumption of some important network configurations. 52
4.10 Memory consumption of some important network configurations. Cells without entries denote network configurations for which compilation or transfer was impossible. 52
4.11 The width limits of a neural network given different depths, assuming they are required to fit onto a 1024 KB Flash memory. 55
4.12 The unavoidable memory overhead of an Arduino sketch when compiled for the Arduino Nano 33 BLE. 59
4.13 Memory consumption of the various parts of our experimental program. 60
4.14 Measured invocation runtime of neural networks of various sizes. 63

5.1 The definitions of our various parameters. 73
5.2 The value of ψ given different invocation ratios ρ and energy consumption ratios ϕ. 75

5.3 The amount of energy a neural network power management system needs to save in order to break even ψ, given observed ϕ and selected ρ. 78

List of Figures

1.1 The number of devices connected to the internet. Taken from [Lasse Lueth, 2018]. 2
1.2 An example of solar panels being used to provide sustainable energy for a deployed IoT device. Taken from [OnL, 2017]. 3
1.3 The cost of computer memory over time. Taken from [hbl, 2017]. 5
1.4 Arduino Nano 33 BLE, the physical IoT device we plan to use. 6
1.5 An overview of how our report is structured. 6
1.6 ψ, the percentage of energy that a neural network has to save in order to break even with its consumption. ϕ is the proportion of energy input going to the neural network, and ρ is the frequency of invocation. Calculated using formula 1.1. 8

2.1 One year of solar power availability at a particular geographical location. Taken from [Buchli, 2014]. 12
2.2 The basic structure of Reinforcement Learning. Taken from [Ope, 2018c]. 13
2.3 Comparison of root mean square deviation from energy neutrality of each month of spring (x-axis) for three competing methods. Taken from [Hsu et al., 2015]. 17
2.4 Illustration of a feed-forward neural network, in which connections never go backward. Taken from [Res, 2020]. 18
2.5 A taxonomy of some of the most popular algorithms used in modern RL. Taken from [Ope, 2018d] 19
2.6 The flow of operation using TensorFlow Lite. Taken from [Ten, 2020a]. . 20
2.7 An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019]. 23
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]. 25
2.9 Graphs showing simulated solar power and corresponding duty cycle chosen by agents trained using neural networks. The final graph shows the variance of each agent, resulting from the factor ζ indicating how much an agent is punished for variance. Taken from [Murad et al., 2019a], 26

3.1 Arduino Nano 33 BLE, the physical IoT device we plan to use. 29
3.2 The iterative process we will follow for the design and validation of the neural network. 31
3.3 The iterative nature of design science. Taken from [Des, 2019]. 32

4.1 The memory consumption of our static program. 41
4.2 The runtime of a sense cycle program in isolation over 30 iterations. . . . 43
4.3 The runtime of a sense cycle program in isolation over 30 iterations, this time when integrated into the larger project. 44
4.4 Box and Whisker chart displaying the mean and outliers of the runtime of sensor scans. 45
4.5 The memory consumption of the neural network. 48
4.6 The behavior of our initially received neural network. The network takes eight values as input, but for the sake of visualization we sample two and then repeat those. 49
4.7 The file size of neural networks of various configurations when stored as compressed TensorFlow Lite files. 53
4.8 The tflite file size (blue) and final flash memory requirements (green) of neural networks with depth = 3 and various widths. 54
4.9 The Flash memory requirement of neural networks as a function of network width given five different network depths. The black line indicates the Flash memory limit imposed by our chosen microcontroller, 1024 Kilobytes. The point at which each network configuration exceeds this limits is indicated. 56
4.10 The Flash memory requirement of neural networks as a function of network width given depth = 1. The black line indicates the Flash memory limit imposed by our chosen microcontroller, 1024 Kilobytes. Note that the x-axis needs to extend significantly further than in figure 4.9 to reach the point where the lines meet. 57
4.11 Total memory consumption at runtime. 60
4.12 The distribution of flash memory at compile time. 61
4.13 The distribution of RAM at compile time. 61
4.14 The console output produced by compiling the final combined power management application. 62
4.15 The runtime of the invocation of neural networks of various configurations. Each data point represents the mean of a sample size of 30 runs for that width / depth combination. The corresponding variance, expressed as standard deviation, is indicated through grey vertical lines. 64

5.1 An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019]. 71
5.2 A further abstracted model of the energy consumption of different phases in an IoT sensing node's life cycle. Power and time consumption are merely indicative. Based on [Tamkittikhun, 2019]. 71
5.3 ψ, the percentage of energy that a neural network has to save in order to break even with its consumption. ϕ is the proportion of energy input going to the neural network, and ρ is the frequency of invocation. Calculated using formula 5.2. 75
5.4 Power Consumption of various nodes. Taken from [Ferry et al., 2011]. . . 78

5.5 The percentage of energy the neural network needs to help save in order to break even with its consumption, ψ. Plotted for all ρ and the two observed values of ϕ, 0.02 and 1.86. 80

List of Equations

2.1 Q function 14
2.2 Getting optimal action from a Q-function 15
2.3 Energy neutrality 16
2.4 Berg's runtime memory estimation 22
2.5 Total energy consumption 23
2.6 Simple energy consumption 24
4.1 Calculated formula for TensorFlow Lite memory size 52
4.2 Formula for final Flash Memory Requirement of a neural network. 55
4.4 Union of sets 59
4.5 Neural network runtime 64
5.0 Neural network power consumption 74
5.1 Total power consumption 74
5.2 Energy percentage 74

Symbols

μ	Statistical population mean
σ	Standard deviation
$\cup$	Union of sets
$\cap$	Intersection of sets
Π_t	TensorFlow Lite file size
Π_f	Compiled neural network file size
Π_e	Runtime of the invocation of a neural network
A	Ampere
μA	Microampere
W	Watt
mW	Milliwatt
mWs	Milliwattsecond
μWs	Microwattsecond
E_{sense}	Energy Consumption of a Sense Cycle
E_{nn}	Energy Consumption of a Neural Network
E_{tot}	Energy Consumption of a loop iteration
ϕ	Ratio of NN and Sense Cycle Energy
ρ	Rate of Neural Network Invocation
ψ	Energy Consumption Percentage of Neural Network

Acronyms

IoT	Internet of Things
NN	Neural Network
FFNN	Feed-Forward Neural Network
PM	Power Management
DS	Design Science
RAM	Random Access Memory
BLE	Bluetooth Low Energy
OS	Operating System
RL	Reinforcement Learning
ENN	Externally Provided Neural Network

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 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 smartphones, 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 devices 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 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 less output than the available power allows for. Good power management strikes a balance between the two, maximizing output while minimizing energy consumption.

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

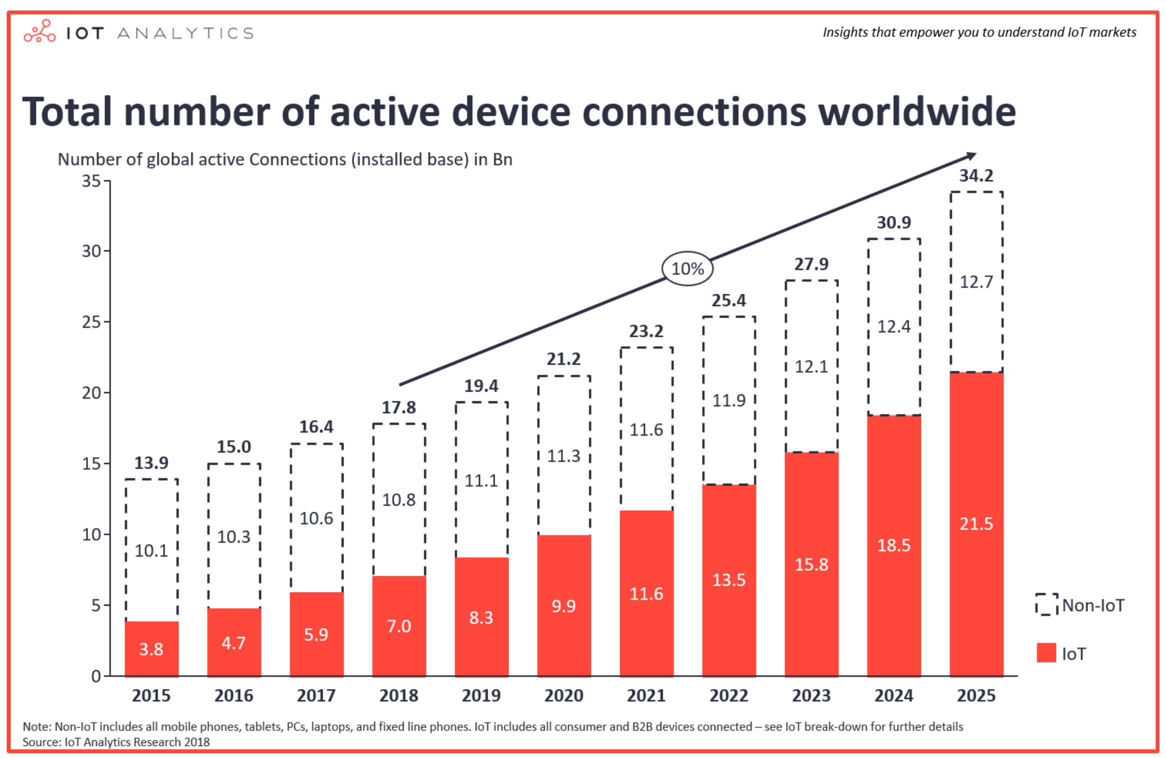

Figure 1.1: The number of devices connected to the internet. Taken from [Lasse Lueth, 2018].

transition into laptops and smartphones, this changed. Optimizing both hardware and software 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 smartphones 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 smartphones, it is not practical to plug these devices into the grid for charging at 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 lifetime. This lifetime can typically be on the scale of a couple of years, at which point many devices need maintenance or replacement anyway. This is a workable solution for many use cases.

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 challenges of their own, however. There is a need to consider the variable nature of such techniques. With solar panels, weather

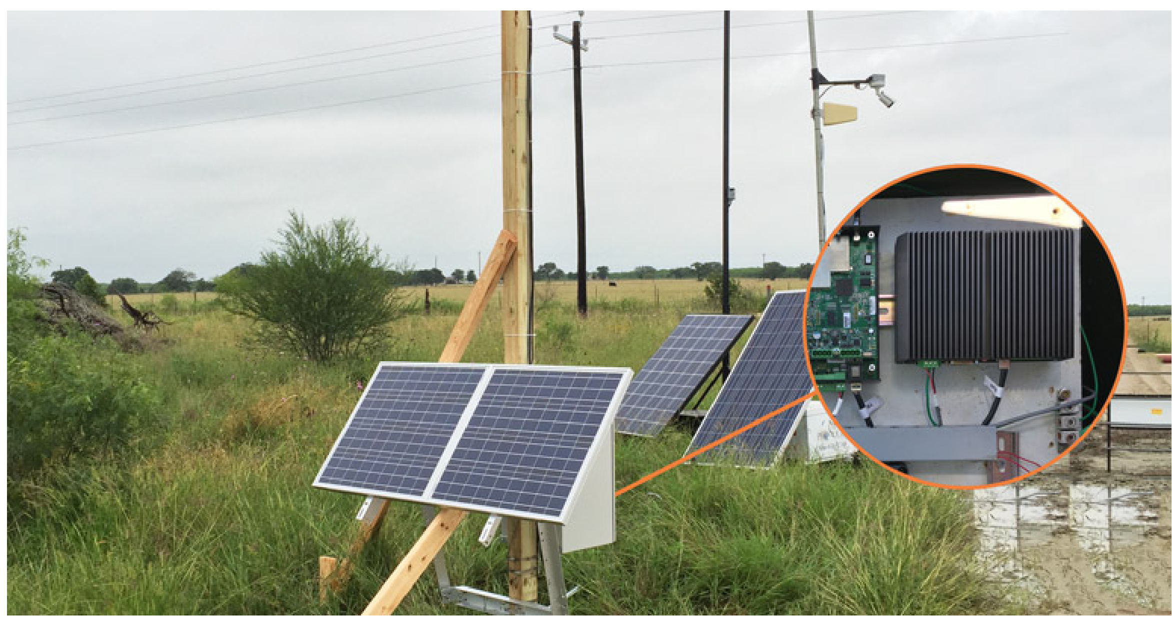

Figure 1.2: An example of solar panels being used to provide sustainable energy for a deployed IoT device. Taken from [OnL, 2017].

starts playing a major role in how the IoT device should perform its functions. The IoT device might have to start hoarding energy during the summer if it is to make it through the winter, for instance. On a smaller scale, the day-to-day throughput might be tuned up or down based on weather reports, if parsed intelligently. These sorts of decisions provide a range of tuning knobs that can affect both the life time and the performance of an IoT device drastically.

Before delving into the details of how to tune a given power management, however, one must first consider whether the approach is correctly *dimensioned*. Deploying solar panels to a region where the sun does not appear for months, for instance, will inevitably lead to disaster. On the other hand, if deployed in a desert with constant, powerful sun, there might not be much need for the parsing of weather reports or intelligent use of energy. The interesting case is the one where there is a *balance* between energy coming in through the solar panels and energy being spent. Without this being the case, the IoT device consumes too much or too little power for any software or hardware decisions we make to matter. When there is such a balance present, however, things change. If we let the device perform its function at 100 % capacity at all times, it would consume more energy than provided with and fail. We also shouldn't turn the throughput down too much either, though, as we want as much output from the device as possible given the available power.

To tune the imaginary knobs in an intelligent, 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 for a long period. If no such battery is available, the approach doesn't work. 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 provide an appropriate duty cycle as output. This can be repeated in shorter regular intervals, providing adaptability without interference from developers. The result is a more adaptive and 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. Neural networks 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 our project aims to remedy. 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 help save more energy than they consume?

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.

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

We chose the Arduino Nano 33 BLE as a representative device for our experiments [Ard, 2020b]. The microcontroller has 1 MB of Flash memory and 250 KB of RAM, and it is shown in figure 1.4

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 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 can save energy. The reason

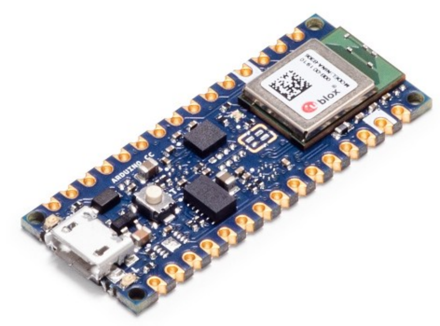

Figure 1.4: Arduino Nano 33 BLE, the physical IoT device we plan to use.

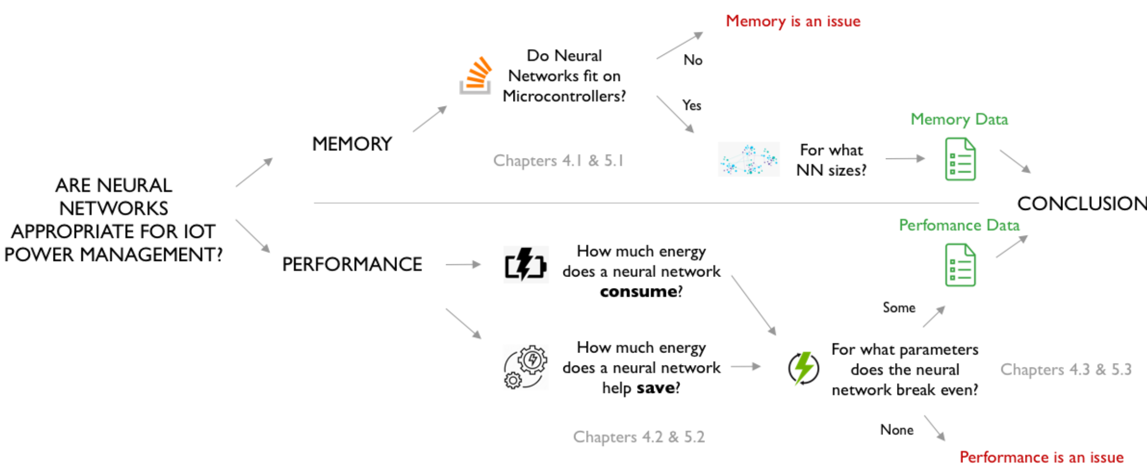

Figure 1.5: An overview of how our report is structured.

we need to ask this question is that 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.

In what circumstances is the neural network-based power management able to help save more energy than it consumes?

This secondary research question helps specify how we intend to go about answering the main research question. If the neural network approach to IoT power management is usable at all, there will still be limits to the runtime configuration for which the system is able to save power. By *circumstances*, we mean these limits. With these three research questions in place, the problem scope of our project is thus clearly defined.

An overview of how we go about our research is presented in figure 1.5.

1.3 Results

The result of our project is insight into when the neural network approach to IoT power management makes sense. The goal of such power management is to achieve the following behavior.

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 experimentation. Utilizing the Design Science framework [Wieringa, 2014], we perform a 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. This data can then be used to discuss our research questions.

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 our microcontroller. This challenge led to the need for compression of the neural network, and we developed the steps required to end up with a neural network runnable on IoT microcontrollers. With this method in place, we generated neural networks for a wide range of widths and depths, attempting to establish when the neural networks grew too large for the IoT device's memory. The result is presented in table 1.1. We thus concluded that all neural networks within these size limits indeed fit in an IoT device's memory.

Network Depth	**Width Limit**
1	23 752
2	482
3	342
4	280
5	242

Table 1.1: The width limits of a neural network given different depths, assuming they are required to fit onto a 1024 KB Flash memory.

With the neural network in place, the next step was to measure its performance. When measuring the neural network's energy consumption, we assumed a direct dependency on CPU runtime. We again found a link between network size and this runtime, eventually producing a mathematical formula that predicted runtime given network size. We then used these results to discuss which range of parameters the neural network approach is appropriate for in the IoT field. These parameters mainly

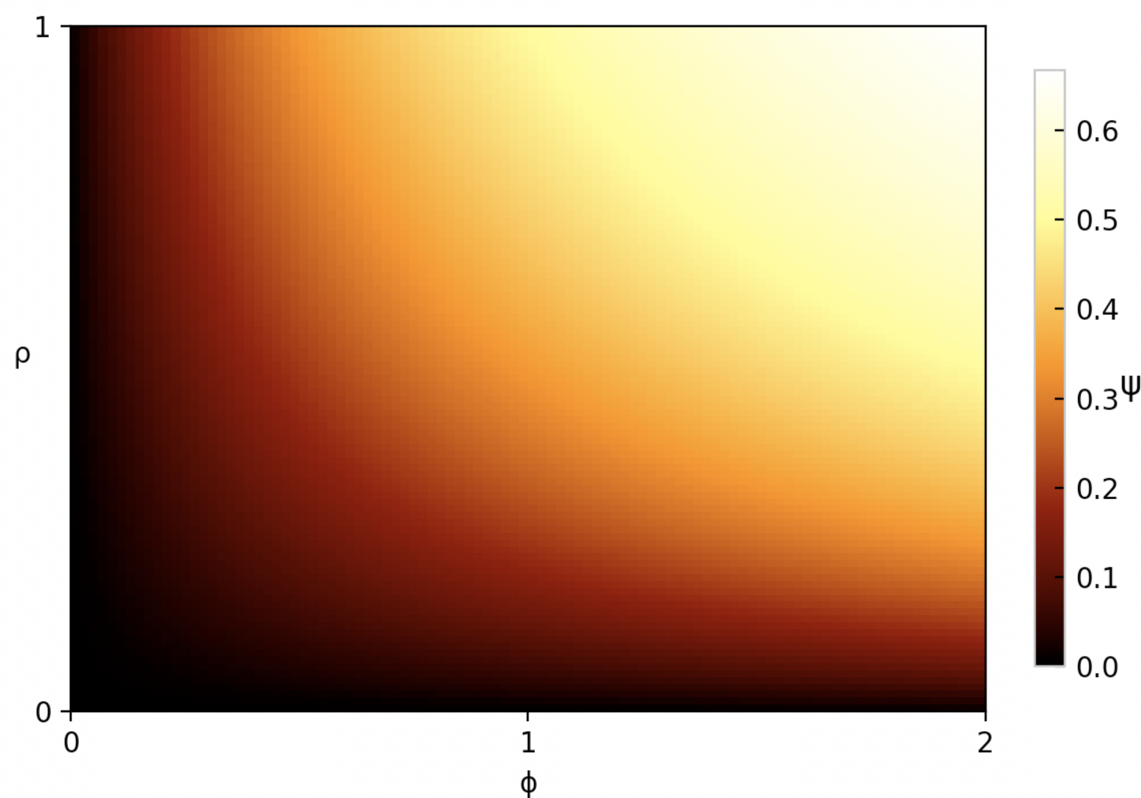

Figure 1.6: ψ, the percentage of energy that a neural network has to save in order to break even with its consumption. ϕ is the proportion of energy input going to the neural network, and ρ is the frequency of invocation. Calculated using formula 1.1.

include two variables: one, how much energy the neural network consumes compared to the rest of the device's functions. The neural network helps reduce energy spent by adjusting how often these surrounding functions are run, meaning the amount of energy saved is directly dependent on this energy relationship. We call this ϕ. The second crucial parameter how often the neural network is invoked. The network could be asked to update the device's policy every iteration of its loop, but if circumstances haven't changed much from iteration to iteration, there might not be much benefit. Invoking it less frequently reduces the amount of energy spent on power management, and the loss of performance might be minimal. We call this invocation rate ρ. These were combined in the following derived formula:

$$\psi = \frac{\rho}{\frac{1}{\phi} + \rho} \tag{1.1}$$

The result of this formula, ψ, is the percentage of energy a neural network needs to help reduce in order to break even with its consumption. Lower values for either parameter lead to the neural network needing to help save less energy to be worth

including. If both are high, the neural network would have to help save an unrealistic amount of energy to break even with its consumption, meaning it would likely be poorly suited for the IoT. We can use the formula to validate whether a given neural network-based power management is appropriate for the IoT field, thus answering our research question. This is one of the **main results** of our work, and it is illustrated in figure 1.6 for all ρ and ϕ from 0 to 2.

We conclude by applying these methods to a neural network developed specifically for IoT power management [Murad et al., 2019a]. We do not consider the design or construction of this neural network to be within the scope of our work. However, through testing it with our developed methods on real hardware, we were able to provide novel insight into it. First, we found that the network consumed approximately 25 % of both the Arduino's RAM and Flash memory. Memory constraints were thus not violated. We then looked at the neural network's *performance* when coupled with a representative IoT application. We tested with two different such applications: one consuming significantly *more* energy than the neural network per iteration, and one where the neural network was the heavier consumer. If the neural network was invoked every iteration, we observed that it would need to help save approximately 2 % or 86 % energy to break even, respectively. If it is invoked less often, we can use formula 1.1 to calculate this break-even point. Asking the neural network for a new policy every tenth iteration, for instance, we get a ρ of 0.1. The formula then yields that the neural network would have to help save **0.2 %** or **15.7 %** energy to break even, respectively. These results show how our developed methods can be applied to provide insight into neural networks used for IoT power management.

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 experiments performed and their resulting data. It shows the concrete steps taken in order to set up and perform these experiments, first for the static application, then for the neural network, and finally combined as a power management system. The generated data is presented through a series of tables, figures, and formulas.

Chapter 5 discusses the implications of the data produced in chapter 4. It follows the research methodology outlined in chapter 3, discussing each of our posed research

questions in turn. Their level of validation is considered, and where the data allows for it, answers to our research questions are presented.

Chapter 6 summarizes the project's problem and the main findings, as well as providing an outline for which topics are interesting to pursue as further work.

Chapter 2 Background

Chapter 2 provides background material for the rest of the report, explaining key concepts and terminology. It also analyzes what has already been achieved in the field through related works. 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 use. 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. 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. One common approach is to always "go". Ignoring energy efficiency altogether, this approach simply performs the device's function as often as possible. Obviously, this *always-go* approach does not account for whether performing a scan is actually a good idea. Often, this means energy is wasted – virtually the same amount of data could have been produced with fewer scans if chosen tactically. To achieve this sort of improvement, 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. A 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. The duty cycle can then be represented as the 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.

There are several approaches available for calculating the appropriate duty cycle of an IoT node. 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.

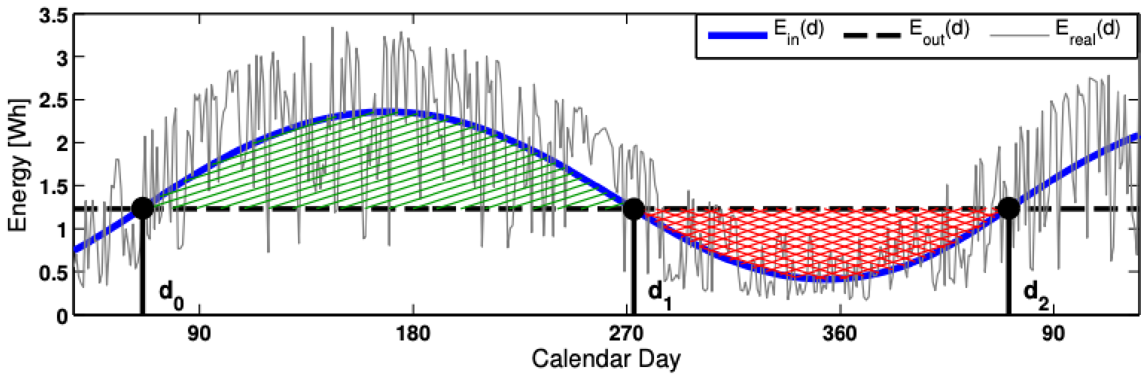

Figure 2.1: One year of solar power availability at a particular geographical location. Taken from [Buchli, 2014].

[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 bring 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 simple idea that when training a machine learning agent, rewarding it for good behavior should lead to a 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.

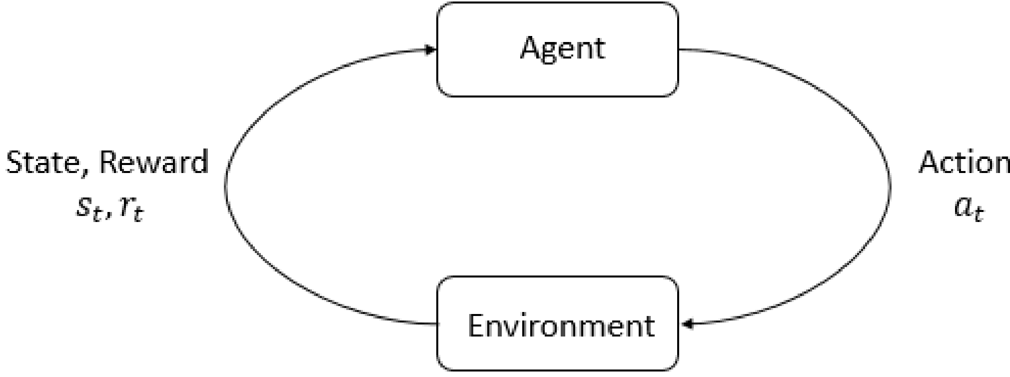

Figure 2.2: The basic structure of Reinforcement Learning. Taken from [Ope, 2018c].

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 chessboard. 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*.

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 receives an indication of how good the new world becomes. An action leading to a better world means the agent becomes more likely to perform that action in the future. By letting the agent choose a myriad of actions and adapting its strategy according to perceived *goodness*, we are letting it *train*. The result is an agent able to perform excellently in its given, simulated world. Hopefully, this behavior **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 desired 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.

With these basic definitions in place, we now turn to some of the main challenges faced by a RL agent. If the reward of a given action is good, how fervently should the agent follow the parameters that led to that reward? At 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 concepts we introduce next.

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]:

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

The exact meaning of the terms in formula2.1 are not important, but we explain them briefly for completeness. s represents the state of the world, and a is an action to be taken. R is the reward function, calculated 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 evaluate a given action α by calculating the cumulative reward of the world *over time*, given that the actor takes the given action *now*. The Q-learning technique then updates 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. This 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 to 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 to achieve a goal. In the context of reinforcement learning, the term also sometimes encapsulates specifications of the parameters such as the reward function.

We look at one algorithm in detail to better explore the concept. Twin Delayed DDPG, or TD3, is an example. 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, for example, it is per definition 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 behavior 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. Without going into detail, the result is an algorithm that has been shown to outperform several competing Q-learning algorithms [Ope, 2018a].

To understand some of the parameters a RL algorithm might tune, we look at why TD3 was proposed as a replacement for DDPG. 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 action, 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 behavior.

In summary, we see how a reinforcement learning algorithm can be specified not just as a selection of training parameters, but also as adjustments of 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 works 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 the distance from energy neutrality [Hsu et al., 2009a] [Hsu et al., 2014] [Hsu et al., 2015].

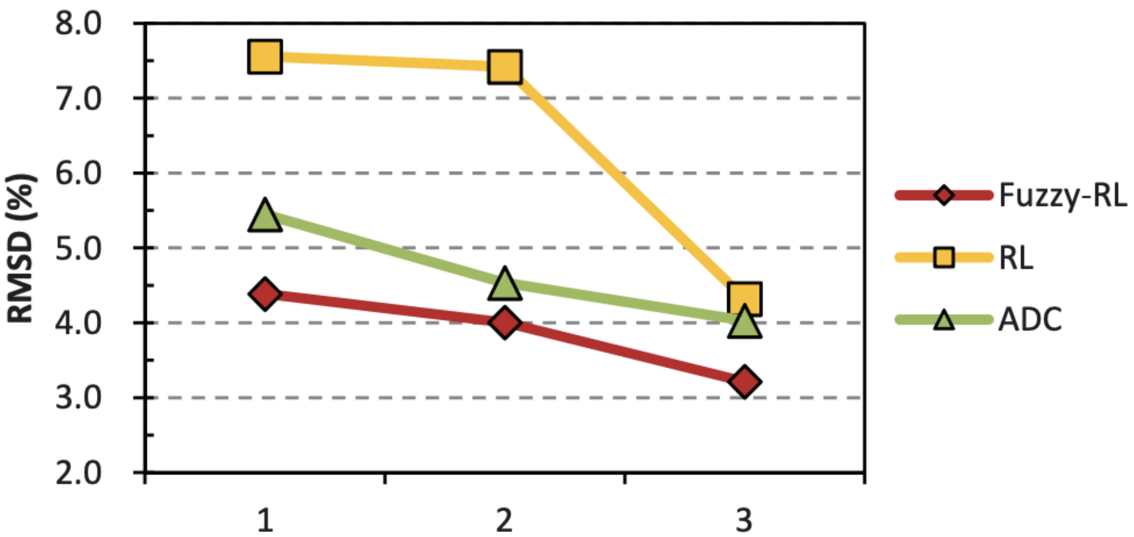

Figure 2.3: Comparison of root mean square deviation from energy neutrality of each month of spring (x-axis) for three competing methods. Taken from [Hsu et al., 2015].

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* refers to two particular reinforcement learning 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.

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.

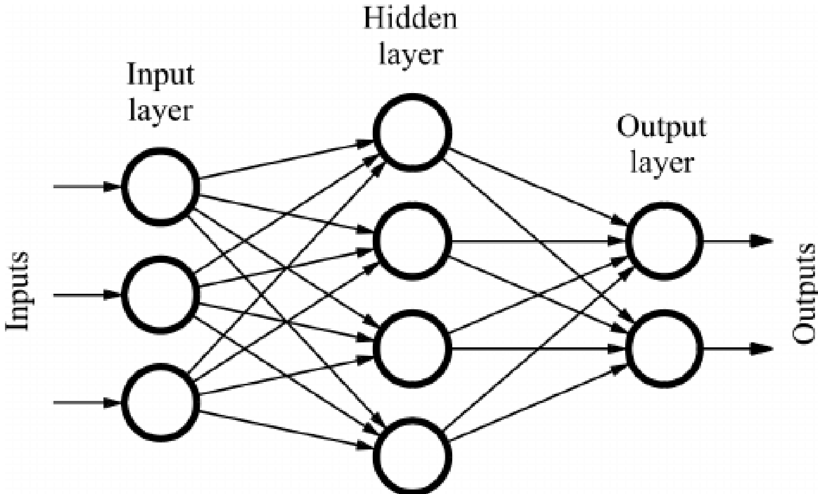

Figure 2.4: Illustration of a feed-forward neural network, in which connections never go backward. Taken from [Res, 2020].

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 behavior 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 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 [NG:, 2020]. 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* spectrum 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.

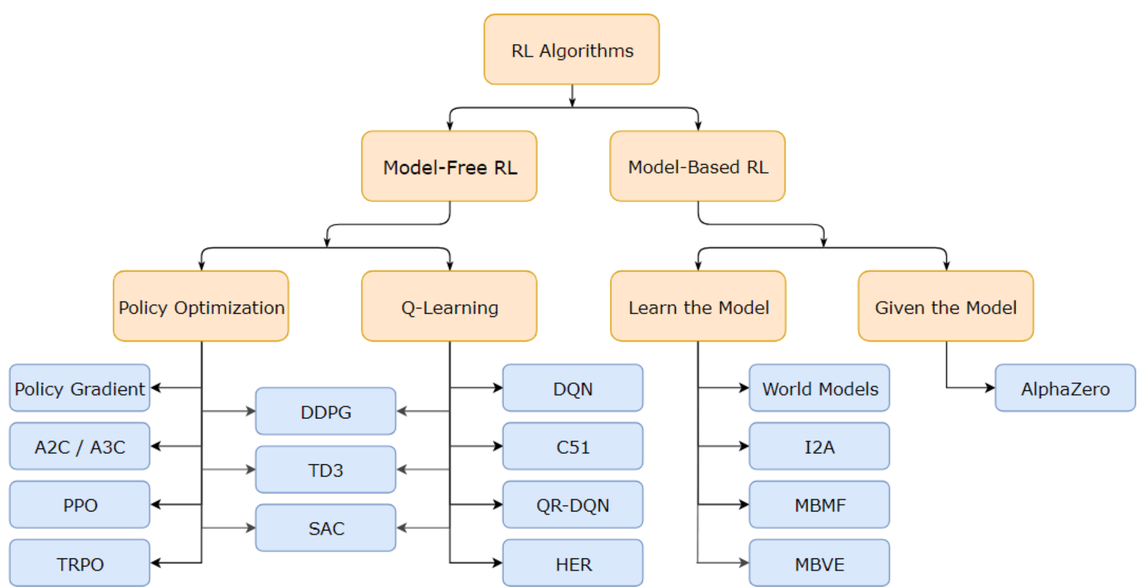

Figure 2.5: A taxonomy of some of the most popular algorithms used in modern RL. Taken from [Ope, 2018d]

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 at 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 called 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 to achieve 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 practice. 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, 2020a].

Even TensorFlow is not a sufficiently specific framework for our purposes, 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, 2020a]. 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 behavior, 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 fit 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 $2 \leqslant L \leqslant 5$. 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 [Ard, 2020a]. This is especially true for microcontrollers, where the OS is often simple 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 before-hand 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 width of the widest hidden layer. With Berg's particular setup, he found that his total available RAM was ~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

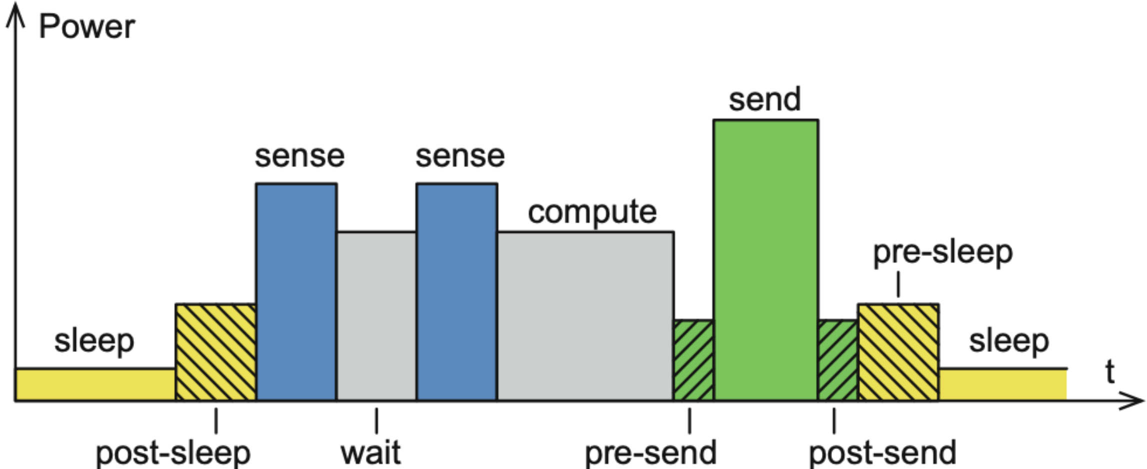

Figure 2.7: An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019].

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 the 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 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 relationship 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 the 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 modeling 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 reasonably 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

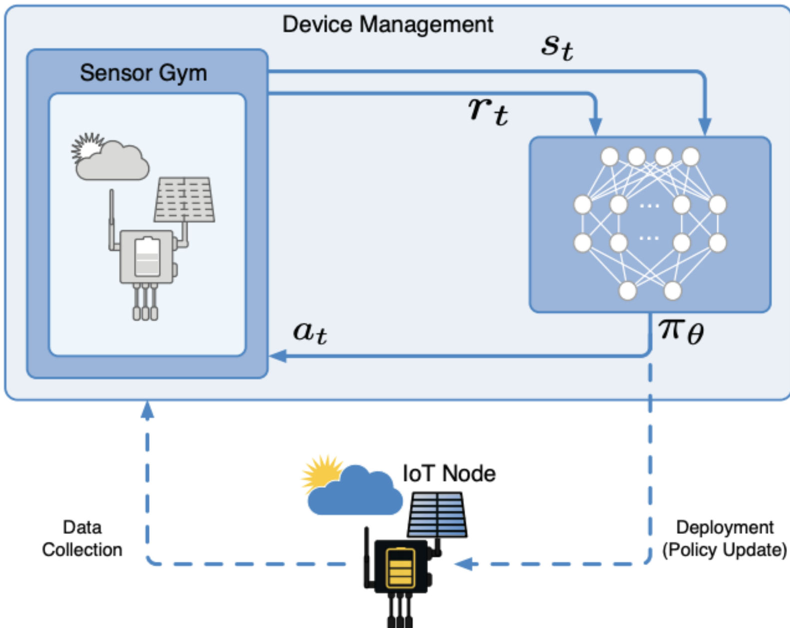

Figure 2.8: The intended agent/environment setup of [Murad et al., 2019a]. The upper parts represent training and invocation from a neural network, while the lower is the updating of the policy of an actual IoT device. This lower part was only simulated in their work. Taken from [Murad et al., 2019a].

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, 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 leave 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.

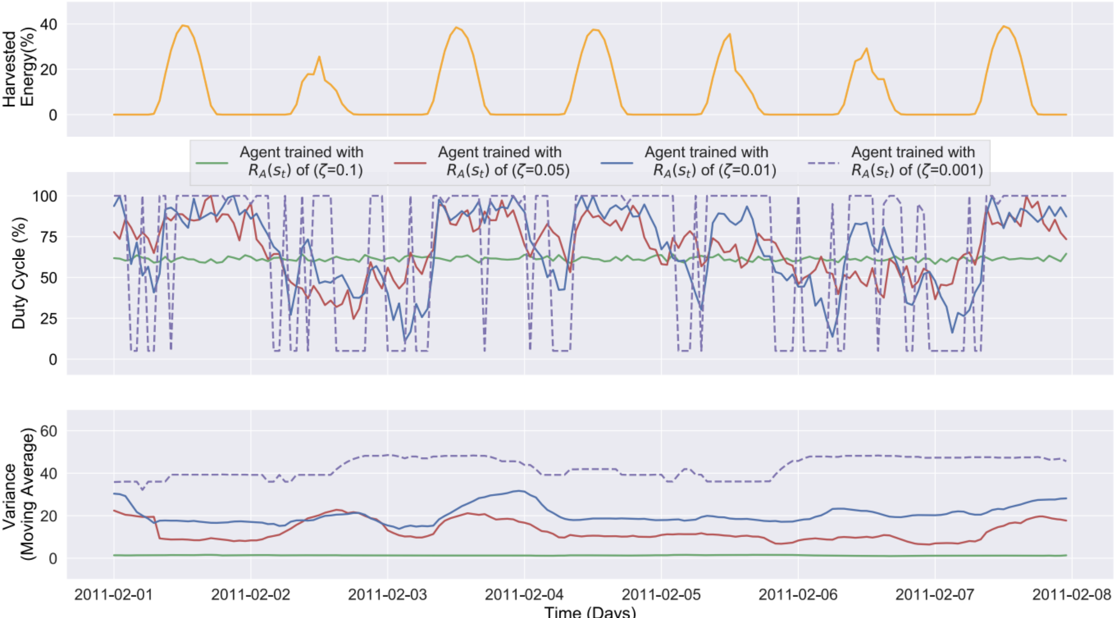

Figure 2.9: Graphs showing simulated solar power and corresponding duty cycle chosen by agents trained using neural networks. The final graph shows the variance of each agent, resulting from the factor ζ indicating how much an agent is punished for variance. Taken from [Murad et al., 2019a],

Chapter 3 Methodology

Chapter 3 provides an outline of how we wish to achieve our goals. Section 3.1 introduces our research question, and provides the reasoning for our selection of parameters. Section 3.2 goes on to present how we ensure our work is scientific, focusing on reproducibility and verification rather than just the implementation. Finally, section 3.3 provides a framework for how we intend to conduct our research in a manner that enables us to answer our research question.

3.1 Research Question and Context

The goal of our project is to investigate the interaction of neural networks 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 help save more energy than they consume?

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 memory-based part of the research question is quite distinct from measuring its performance, 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?

The remaining part of the Research Question is concerned with the performance of the neural network-based power management. For the approach to make sense, we

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).

In what circumstances is the neural network-based power management able to help save more energy than it consumes?

With that, the Research Question has been almost entirely dissected for analysis. There is a detail so far gone unmentioned, though: what is meant by *today's IoT devices*. We discuss this next.

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 devices 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 the 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	1 MB	256 KB	Yes
nRF9160 (NB-IoT)	64 MHz	1 MB	256 KB	No
nRF52-DK (BLE)	64 MHz	512 KB	64 KB	Yes
Arduino Nano 33 BLE	**64 MHz**	**1MB**	**256 KB**	**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].

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 3.1.

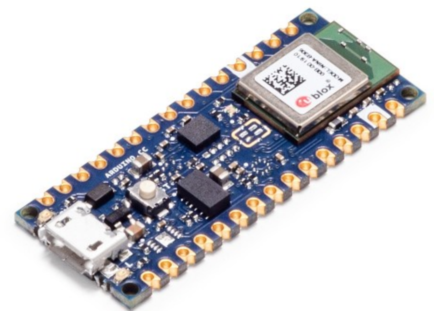

Figure 3.1: Arduino Nano 33 BLE, the physical IoT device we plan to use.

Arduino microcontrollers are known to be designed for ease of use programmatically. Few design decisions should need to be hardware-specific. 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 sensors are fit onto the smallest form factor available: 45x18mm [Ard, 2020b]. Last but not least, it has the memory capabilities and mbed-os compatibility 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, 2020a]. 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.

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.

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.

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 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 3.2 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 can approach this kind of design in a scientific manner.

Figure 3.3 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

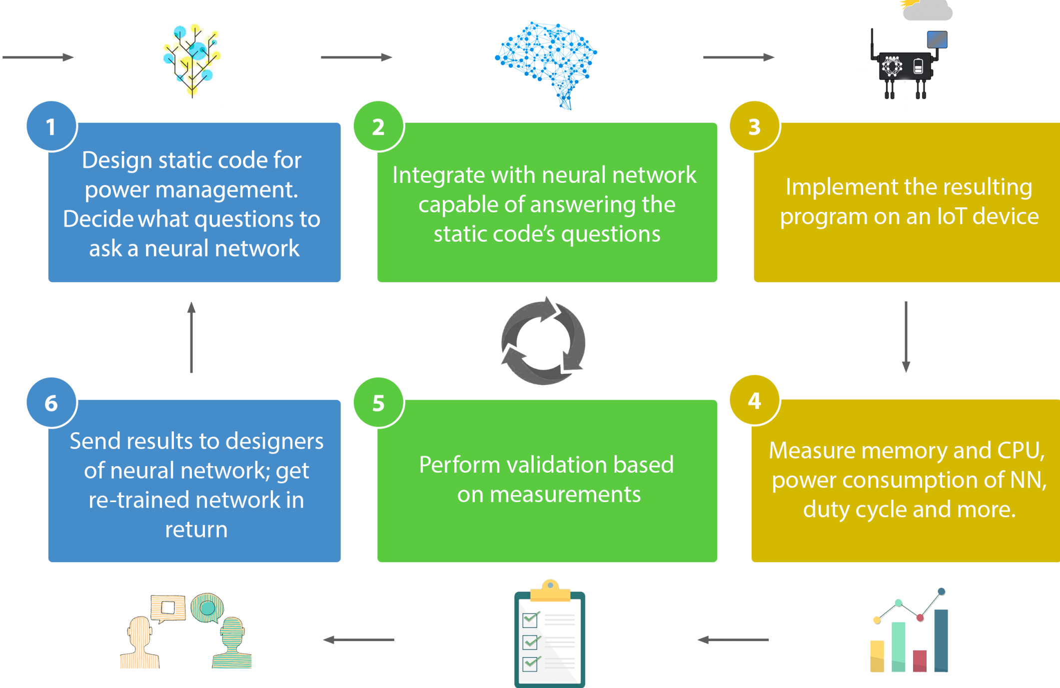

Figure 3.2: The iterative process we will follow for the design and validation of the neural network.

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 be applied in this field, potentially leading to a new step forward in IoT power management. 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 practice 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*

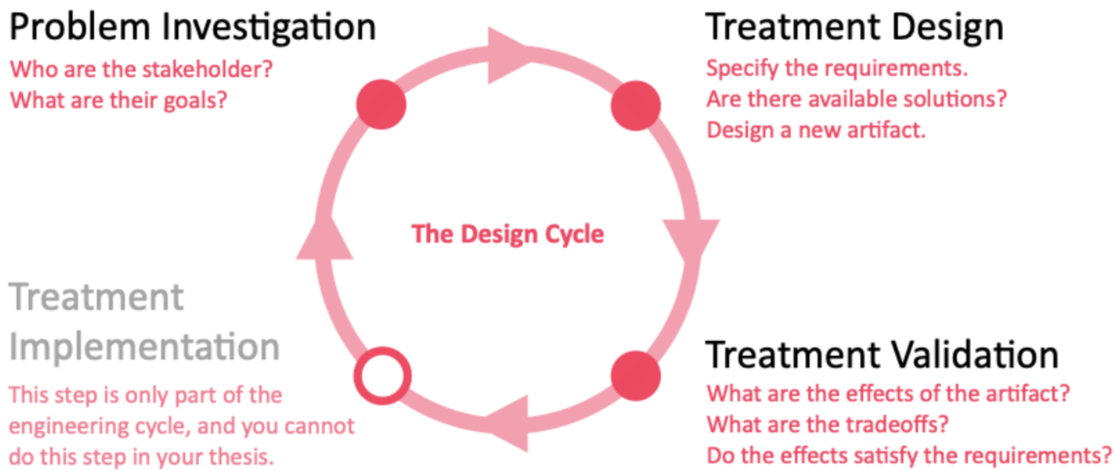

Figure 3.3: The iterative nature of design science. Taken from [Des, 2019].

solutions described in figure 3.3 during problem investigation. Namely, treatments need to be usable on State-of-the-art microcontrollers, and they must be able to save more energy than they consume. 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 3.3, 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 3.3. 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 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 behavior in simulations.

– Data used as input to the neural network.

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 behavior on real IoT hardware, using real data.»

We work with the assumption that our chosen predefined parameters yield representa-

tive behavior of IoT hardware in general. That is, the Arduino board and mbed-os chosen give us behavior that reflects what we would see on most other possible choices. Given this, we wish to perform experiments and measurements that yield whether the inclusion of a neural network for power management purposes actually reduces the distance from energy neutrality overall. Inspired by our Secondary Research Questions, 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 neural network-based power management helps save more energy than it consumes.

If we reach the conclusion that any of these feasibility criterion are **not** satisfied, we wish to provide detailed descriptions of both why and of what must change 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 violated. 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 behavior 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 criteria 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 indicates the status of that particular part of the system. The feasibility criteria are satisfied if, and only if, each part of the system's sub-criteria 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

In order to facilitate the invocation of a neural network on an IoT microcontroller, we set up a surrounding ecosystem. We need this system to perform some action representative of an IoT device's task, and it needs to use our neural network in some way for power

management. We implement this as follows. We create a program surrounding our neural network called a *sense cycle*. The name is intended to encapsulate the specific kind of IoT device we've chosen for our project, a *sensing* IoT node. That is, a device whose purpose it is to make some observation about the world such as current temperature. This typically needs to be performed regularly, hence the *cycle* part of the name.

We implement the interaction between the sense cycle application and the neural network as a selection of **duty cycle**, described in chapter 2. The application chooses to perform its sensor scan with some selected frequency, realizing different levels of power management based on how long it sleeps between each sense cycle. The length of this delay is our chosen implementation of a duty cycle. In some less frequent interval, the sense cycle can then ask the neural network for a new duty cycle, which is subsequently used for one interval. In this way, power management is implemented through the invocation of a neural network.

In order to scientifically analyze the contribution of our sense cycle to the overall project, we formulate the following sub-criterion.

(i) **Feasibility criterion 1.1:** Verify that the static code constituting a sense cycle can be transferred and run on an IoT microcontroller without significant resource consumption.

(ii) **Proposition criterion 1.1:** If it cannot, reduce the scope of the program so as to less accurately reflect a real IoT node's function, but taking less space and computation time.

By *significant resource consumption*, we mean memory and runtime requirements that make it infeasible to use a neural network alongside it. As this depends on the requirements of the neural network, we let this definition stand without further specification for now. If feasibility criterion 1.1 is fulfilled, we consider the sense cycle part of our experimental setup complete. We explain in section 4.1 the specific steps we take to reach this goal. Before that, though, we have a closer look at the part of the experiment our sense cycles is meant to enable: the neural network.

3.3.2 Neural Network on a Microcontroller

The goal of the sense cycle is to facilitate the use of a neural network. To formalize this goal, and also to fully encapsulate the overarching Feasibility criterion 1, we formulate the following:

(i) **Feasibility criterion 1.2:** Verify that neural networks of appropriate size fit on an IoT's device flash memory. Further, verify that invocation does not require more memory than available in the device's RAM.

(ii) **Proposition criterion 1.2:** If either memory constraint is violated, specify either how much the hardware would need to improve or the network size to be reduced for a fit.

By *appropriate* size, we here mean a network depth and width that is in relatively close proximity to those shown to have produced efficient power management behavior. We show the steps taken to verify this feasibility criterion in section 4.2. These criteria ensure we validate the use of the neural network itself properly, but it does not put it into context and validate its usefulness. We remedy this in section 3.3.3.

3.3.3 Power Management

The final aspect we need to cover is the total behavior we wish to end up producing – power management. Feasibility criteria 1.1 and 1.2 are sub-criteria of Feasibility criterion 1, asking whether "a given neural network can be transferred to, and ran inference from, a given microcontroller". Thus, with sub-criterion 1.1 and 1.2 fulfilled, we can consider the corresponding main criterion fulfilled. Power management deserves the same treatment. We split Feasibility criterion 2 into two sub-criterion:

(i) **Feasibility criterion 2.1:** Verify that there are possible configurations in which a neural network is able to help save more energy than it consumes.

(ii) **Feasibility criterion 2.2:** Verify whether the *externally provided* neural network saves more energy than it consumes.

Feasibility criterion 2.1 is a reality check. We do not have any guarantee that the provided neural network acts in a productive way when deployed onto real hardware. Factors such as memory limits or additional energy **consumption** by the neural network might alter the sum effect drastically. If the resulting behavior does not actually save energy for the device, we have not achieved any sort of power management, let alone an efficient or intelligent one. The sub-criterion makes sure we are cognizant of this fact.

Feasibility criterion 2.2 is similar to the original encompassing criterion, but more specific. It specifies a single particular neural network the we wish to test. The neural network in question is one provided by the research of the encompassing larger project that sparked our project. This externally provided neural network was shown to exhibit promising power management behavior in simulations [Murad et al., 2019a]. If we are able to utilize this neural network on real hardware, observing the same behaviour shown in simulations, we will have helped the larger project take a major step forward in verifying the applicability of the approach.

To ensure we have considered the potential for our feasibility criterion to be unfulfilled, we propose the following proposition criterion:

(i) **Proposition criterion 2.1:** If invocation consumes more energy than it is able to save, identify why. If it is because the neural network does not behave properly, obtain a more fittingly trained network. If the neural network consumes too much energy, reduce its parameters. If it because of hardware aspects, identify which. Either experiment with other hardware, or conclude the approach ill-suited.

(ii) **Proposition criterion 2.2:** If the externally provided neural network does not save more energy than it saves, identify whether it is because of some property of the microcontroller or if the neural network itself simply consumes too much power. The feedback will yield useful information to the designers of the neural network.

Notice that for the first time, we have included the option of concluding the approach as ill-suited a valid conclusion to make. From the outset of this project, we have wanted to validate whether the approach of using neural networks in IoT devices makes *any sense at all*. It is possible that spending the considerable amount of time and effort it takes to train and utilize a neural network, all for the purpose of saving some energy on sensing devices, is like using a sledgehammer to crack a nut. The energy consumed by transferring and inferring from the network might be greater than what we're able to save with the increased intelligence. It is also possible that they simply will not fit on current state-of-the-art hardware, and that we need to wait another 5 or 10 years before the approach becomes applicable. These kinds of overarching reality-check questions are intended to be encapsulated in these proposition criteria. If we are able to successfully transfer and infer from a neural network, then observe that the resulting behavior is efficient power management, we can conclude that the approach makes some sense. If it also out-competes other approaches, as it has in simulations [Murad et al., 2019a], we have a promising new direction in the field. We continue to the experiments that provide real data and test these hypotheses in chapter 4.

Chapter 4 Experiments

Chapter 4 describes the concrete steps we take to produce and implement the various parts of our IoT Power Management system. Section 4.1 deals with the static application surrounding our neural network, the sense cycle. It presents the memory footprints of this application, then looks at estimated energy consumption for various configurations. Section 4.2 follows this pattern, describing the setup and execution of our neural network in isolation. The memory requirements and energy consumption of various network sizes are presented. Section 4.3 finally brings this all together, presenting the memory and energy data produced by combining a sense cycle with a neural network to realize power management on an IoT device.

4.1 Sense Cycle Implementation

We wish to create a program that represents IoT applications in general. Although we mainly wish to test the neural networks used to achieve power management, we need some such regular program as a reference point. The reason for this can best be illustrated by an example. If we find that the neural network consumes 5 KB of flash memory, for instance, that has little value without also providing the context of *how much more than normal* this number is. If we consider some arbitrary IoT device, the pre-existing IoT application will naturally have some memory and battery consumption. If this memory consumption is, say, 500 KB, then the 5 KB we found earlier is a trivially tiny amount. We could thus conclude that the power management system does not put significant strain on the device's memory. If, on the other hand, the IoT application only consumes 5 KB, then inclusion of the neural network suddenly leads to a memory consumption increase of 100 %. This is why it's crucial to first develop a general IoT application for our experiments. We focus on the domain of cyclic energy-harvesting applications [Murad et al., 2019a], and we call our application a *sense cycle*.

The first step of our experiments was thus to develop a sense cycle application. The program consists of two parts: a setup phase, and a loop phase. During setup, global variables are initialized, runtime parameters are established, and logging is started.

Then, without delay, the first iteration of an infinite loop is started. In each iteration, the device performs a set series of tasks. First, it scans its environment. As an example, we have chosen scanning for temperature and humidity. This scan is typically the most important part of an IoT device's programming; it is only this scan that actually provides data and enables the node to generate value. The rest of the program is in place mainly to facilitate this scan, and possibly to transfer or store the results. Because of this, it is also the least generalizable aspect of our program. Depending on what the device intends to scan, it might need to spend several minutes preparing its sensors. Alternatively, it might not be making traditional scans, instead relying on network calls or other heterogeneous functions. Due to these factors, the amount of energy spent on the scanning part of the process varies greatly. We make sure to account for this when modeling the total energy consumption in section 4.3. For now, though, we stick with a temperature and humidity scan as a compromise to end up with somewhat representative sense cycle behavior.

IoT devices that perform scans need to somehow handle the resulting data in a way that makes it useful to the system at large. This mainly consists of either storing it locally, or more commonly, transmitting it to a central server that takes care of storage and processing. This latter approach is the one we implement in our experiment. The Arduino Nano 33 BLE is, as the name suggests, equipped with Bluetooth Low Energy (BLE) equipment. While not as tailored for the IoT domain as protocols like NB-IoT, BLE is a relatively common way to communicate in the internet of things [Haukland, 2019]. Thus, we go with this as the representative handling of data in our program. This concludes the bulk of the sense cycle. The only thing remaining before going to sleep until the next cycle is deciding whether to ask the neural network for updated configuration. How often to ask the network, and for what, is discussed in section 4.2. Before that, though, we look at the data produced by implementing our sense cycle.

For a neural network to work on an IoT board, it needs to fit in the board's memory. Further, for it to provide actual power management, it needs to fit *alongside* an application containing the actual logic for whatever the node is supposed to do. For us that is the sense cycle, and that is why we need to consider the sense cycle's memory consumption. In other words, even though the main thing we want to measure is the **neural network**'s memory consumption, we start by identifying how much room it has to work with by finding the memory consumed by a typical surrounding application.

4.1.1 Memory Consumption

Table 4.1 shows the measured memory consumption of our developed sense cycle application.

The data presented in table 4.1 is only useful in the context of our chosen hardware.

Memory type	Total	Taken	Free
Flash	983 KB	292 KB (29 %)	691 KB (71 %)
RAM	262 KB	67 KB (25 %)	194 KB (75 %)

Table 4.1: Memory consumption of our sense cycle application in isolation.

The Arduino Board we've chosen has a program storage space of 983040 Bytes, or approximately 1MB. Further, it has a RAM of 262 KB. We found that our application takes approximately 29 % of the Flash memory. This is trivially found by the compiler at compile time. The compiler also identifies the RAM requirements of so-called global variables, giving us that 25 % of the RAM is pre-occupied. As long as we avoid mistakenly writing our application in a manner that requires a lot of memory allocation at runtime, this should be the sum RAM consumption of the sense cycle. We thus have a starting point to work with when starting measurements for our neural network. The portion of available memory consumed is illustrated in figure 4.1.

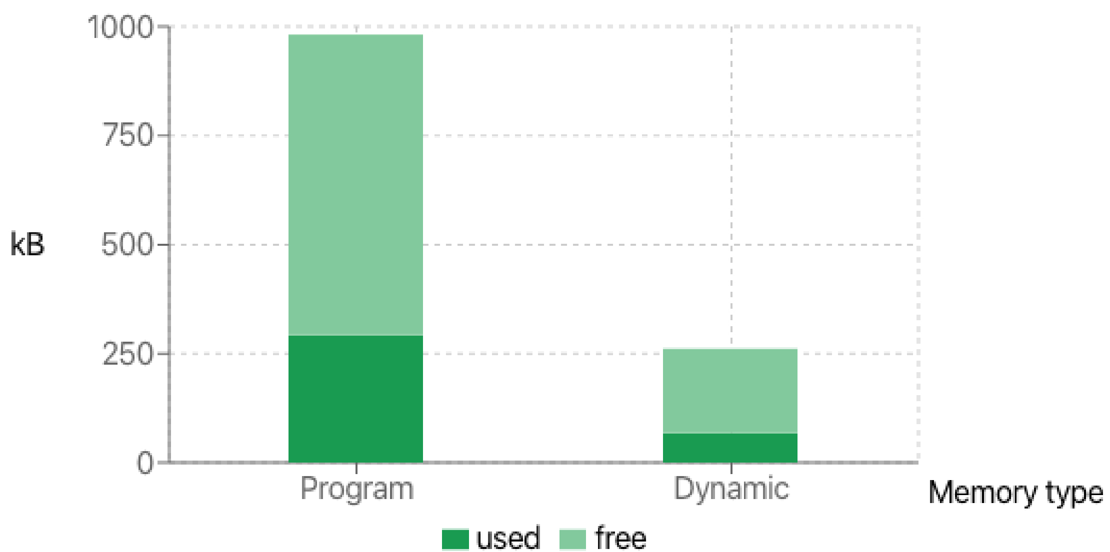

Figure 4.1: The memory consumption of our static program.

It is useful to ask ourselves at this point whether it is reasonable for 25 to 29 percent of the memory to be allocated to a static application. The implication would be that with around 30 % of the memory going to the application, the remaining 70 % would be allocated to the neural network and power management. However, it is important to realize that these numbers do not represent *limits* for how much memory the sense cycle can consume. They are more of a ballpark estimate of how big we can roughly expect a surrounding IoT application to be. Further, it should be noted that the hardware we have chosen is on the upper end of IoT device specifications. This could imply that

most IoT applications would fit on smaller hardware, and that one might acquire such a large device *specifically* for the purpose of including secondary functionality such as power management. Thus, we make a preliminary statement that the presented memory consumption is within realistic, reasonable bounds.

4.1.2 Energy Consumption

To accurately measure the performance of our neural network as a power management tool, we need to know how much energy it consumes. For that to be possible, we must first get an idea of how much the surrounding system consumes when there is no neural network present. It is for this reason we wish to look at the energy consumed by our sense cycle.

As presented in section 2.4.2, the energy consumption is strongly dependant on the runtime of a given process. In fact, we go forward with the assumption that the two are *linearly* correlated. With this assumption, the task of finding out how much energy the various parts of our sense cycle reduces to measuring their runtime on the CPU. We do this trivially by comparing timestamps before and after each process during runtime. The results for five cycles of such measurements are presented in table 4.2.

Task	**Cycle 1**	**Cycle 2**	**Cycle 3**	**Cycle 4**	**Cycle 5**	**Average**
Scan	4ms	4ms	4ms	4ms	4ms	**4ms**
BLE	1ms	0ms	0ms	0ms	1ms	**0.4ms**
Total	6ms	6ms	6ms	6ms	6ms	**6ms**

Table 4.2: Runtime of the different parts of our developed sense cycle, measured over 5 iterations.

It is important at this point to make a comment about the level of accuracy used in these experiments. Several of the observed values vary strongly – the BLE scan goes from 0 ms to 1 ms instead of from, say, 0.83 ms to 0.78 ms. One possible explanation is that our measurements are too rough – the variance of each scan is on the scale of nanoseconds, and our accuracy level of milliseconds causes each to be rounded to the same value. To combat this, the intuitive choice would be to measure the function time on some more precise scale. However, the Arduino Nano 33 BLE has a clock rate of 64 MHz. This means that each clock cycle takes 16 nanoseconds. It is thus impossible to get an accuracy of individual nanoseconds, and $millis()$ is the highest accuracy time-measurement function available on the platform [Ard, 2020c]. As a point of further study, counting individual clock cycles to achieve a time accuracy of multiples of 16 nanoseconds could be of interest. For now, though, the given accuracy is sufficient as an indication for how long the sense cycle will take in comparison with a neural network invocation.

The main tasks of the sense cycle are Environment Scans and BLE communication. The device first creates the data point, in our case a temperature and humidity sensor scan. It then sends this result for processing. We achieved this by connecting to the

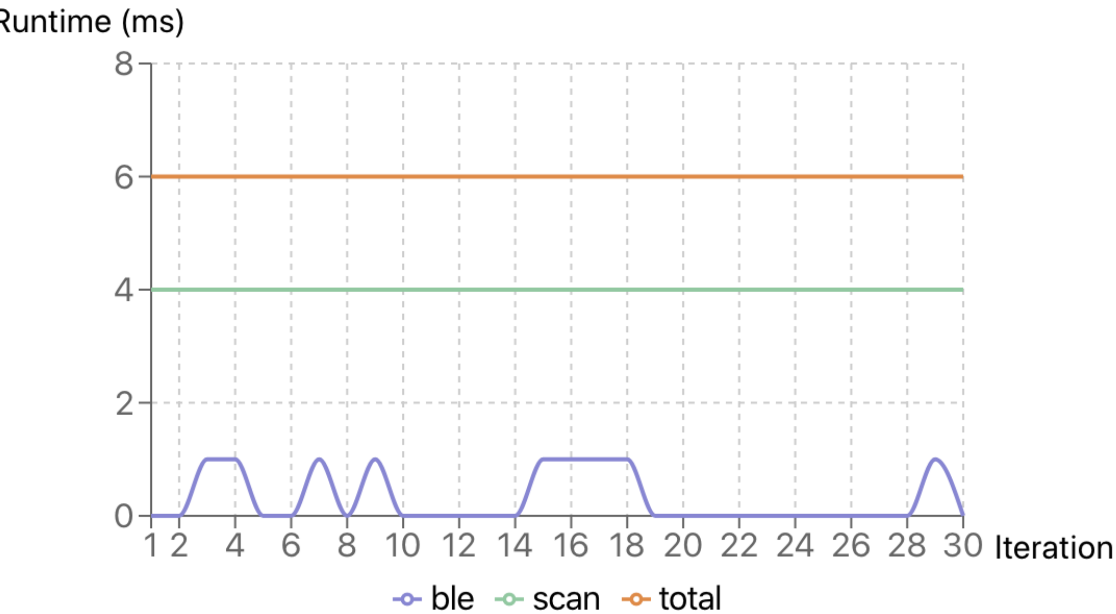

Figure 4.2: The runtime of a sense cycle program in isolation over 30 iterations.

device with the NRF Connect app on a smartphone [NRF, 2020]. These processes are denoted Scan and BLE in the table, respectively. The runtime of an entire iteration of the loop is denoted Total.

We see from the results in table 4.2 that the average runtime of our entire sense cycle in isolation is **6ms**. This number is slightly higher than the sum of its parts as presented – there is some time being spent on things that do not fall under either Environment Scan or BLE Communication. These tasks include moving variables to registers, printing values to a terminal for logging, etc. We conducted the experiment 25 more times to reach the typical number of statistical significance, 30. The results of these experiments can be seen in figure 4.2. We observe that only the BLE communication had variance; the other processes took the same amount of time in every single iteration.

We wish to end up with a set range of milliseconds we can expect the sense cycle to take. Such a data point is required if we are to calculate the energy consumption of our neural network – we need to make sure we aren't accidentally including the runtime of the static code in our invocation measurements. Intuitive values for these estimations would be the averages presented in table 4.2. However, before concluding them final, we integrate the sense cycle with the neural network and measure the time taken by the cycle in that context. Intuitively we don't expect the values to change much, but there are some changes imposed by the integration. For example, we moved all code related to the sense cycle into a separate file then imported it for the sake of clarity in the code. Although the code is the same, such changes can impose slight

runtime overhead due to extra memory registers needed to be allocated for the import, among other things. As such, we perform entirely new, independent measurements of the sense cycle when integrated into the larger project. The results of five iterations are shown in table 4.3. We again attempt to reach statistical significance by doing at least 30 scans, and these are shown in figure 4.3.

Task	Cycle 1	Cycle 2	Cycle 3	Cycle 4	Cycle 5	Average
Scan	7ms	6ms	6ms	6ms	7ms	**6.5ms**
BLE	1ms	1ms	1ms	1ms	1ms	**1ms**
Total	8ms	7ms	7ms	7ms	8ms	**7.5ms**

Table 4.3: Runtime of the different parts of our developed sense cycle, this time when integrated into the larger project.

Figure 4.3: The runtime of a sense cycle program in isolation over 30 iterations, this time when integrated into the larger project.

The averages in table 4.3 are calculated using the full 30 iterations, not just the five presented in the table. We note that indeed, some extra time was required to perform the sense cycle when integrated with the larger project. The interaction between sense cycle and neural network is studied further in section 4.3. With these values, though, we now have good estimates of how long we can expect the sense cycle to take on the CPU. For the sake of getting an idea of how certain we are of these numbers, we calculate the statistical mean μ and standard deviation σ of each data type. For these calculations we need to assume some statistical distribution. To test if we can assume

the normal distribution, we perform a normality test for the sensor scan readings. We do so by using a box and whisker-diagram [Sta, 2020], shown in figure 4.4.

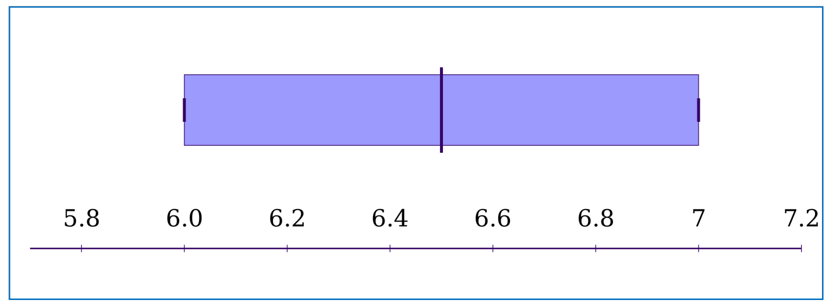

Figure 4.4: Box and Whisker chart displaying the mean and outliers of the runtime of sensor scans.

Using the diagram, we can see that the mean falls right in the middle of the distribution. Additionally, there are "few to no outliers" – none, in our case – which is another good sign of normality. Thus, we consider it reasonable to assume the dataset of sensor scan readings normally distributed.

As for the other two datasets, we first look at the BLE communication data. Perhaps unfortunately, we only observe uniform values of 1.0 ms here. Zero variance is unlikely given the physical nature of the system. We explained earlier that the culprit here might be an insufficient level of accuracy in our scans, and re-doing the experiment with an increased level of accuracy might yield more detailed results. For now, though, we consider the observed runtime of 1 ms to be a sufficient indicator. For the sake of statistical analysis, we consider it a constant. The total time elapsed then becomes a normal distribution plus a constant, which is also a normal distribution. We can thus calculate the mean and standard deviation for each relevant dataset. The results are presented in table 4.4.

Scan type	μ	σ
Sensor scan	6.5 ms	0.5 ms
Send data	1.0 ms	–
Total Sense cycle	7.5 ms	0.5 ms

Table 4.4: Final estimations of CPU runtime of the various parts of a sense cycle program.

4.2 Resource Consumption of Neural Networks

We now move on to the core of this project: implementing neural networks on an IoT device. We do not yet consider the application of a neural network to achieve power management, instead looking at neural networks in isolation to test the boundaries of possible configurations.

4.2.1 Procedure

In order to fit a neural network onto the limited space available on microcontrollers, we need to perform compression. The networks are trained using TensorFlow [Ten, 2019], and we compress them using the tool TensorFlow Lite Micro [Ten, 2020b]. These are described in more detail in section 2.3.3. Using these tools, we perform the following steps to go from a provided neural network to one runnable on a microcontroller:

1. **Receive a trained model.** This is typically in the form of a frozen agent's current neural network weights. It is provided as a zip-file, and TensorFlow's Python framework can read these to load the model into memory.

2. **Convert the loaded model into the .tflite (TensorFlow Lite) format.** This requires some knowledge about the model: which tensor is input and which is output; what algorithm is used, and some others. At this step, it is possible to add optimizations that reduce the size, runtime, or other parameters at the cost of accuracy. Testing indicated that the loss of accuracy was too severe to seriously consider using these, but especially the memory size optimization should be kept in mind as a backup in case a network becomes too large.

3. **Convert the .tflite file into a C array.** The format produced by TensorFlow Lite conversion is not directly compilable into native C, but we can manually compile it using terminal commands and represent it as binary values in an array. This step also includes modifying the resulting C file slightly, to make sure it is in a format compatible with microcontrollers. Specifically, because the network is stored as a single huge array, special care needs to be taken when allocating the memory address for this array. The exact steps can be found on the TensorFlow Lite Micro documentation page [Ten, 2020b].

4. **Import the C array into a main program.** Depending on underlying hardware and architecture chosen, this involves creating header files (.h) where the array is declared and made available. The *main program* designates the code intended to run on the microcontroller, containing the logic for performing the device's main task. In our scenario, this means importing the neural network as a C array into the main file of an Arduino program. When integrated into the larger project, it means integrated with the program that runs our sense cycle.

5. **Load the neural network.** TensorFlow Lite Micro provides library functionality for loading the neural network an imported C array. Some steps have to be taken before invocation can be performed, including allocating memory registers for loading the neuron layers and weights necessary for invocation. The exact steps vary slightly depending on the architecture, described in the project's documentation [Ten, 2020b].

6. **Invoke the neural network.** This is the final step; what we wished to achieve. Before the invocation, inputs need to be set. How these are chosen are highly dependant on the nature of the network. Once these have been set. a simple function call initiates invocation. If successful, the output is then made available as variables.

With these steps, we have a list of actions that take us from a trained neural network to a successful invocation on a microcontroller. We apply these steps to perform experiments with a single given neural network next.

4.2.2 Memory Consumption of a Single Network

We mainly aim to implement specific neural networks that lead to a certain desired effect. Meaning, the main goal of the project is **not** to explore the limits of neural network sizes on microcontrollers in general. Works such as [Berg, 2019] have already performed such an analysis. Thus, we first look at the memory consumption of a given power management neural network as the initial data point. This should give us a ballpark estimate for the flash memory and RAM requirements of a representative neural network.

In the interest of maximizing the performance of the power management provided by our neural network though, it is still useful to explore more general cases. Specifically, it would be useful feedback to the designers of the neural network to explore the size limits of neural networks with the specific purpose of power management, given our specific set of parameters. As such, we want to eventually look at how far the depth or width of a neural network in our specific setting can be pushed before exceeding the given hardware parameters. We look at a single given neural network first, then return to this boundary exploration in section 4.2.3.

For our initial test of a power management neural network, we use an externally provided and trained network. The network's parameters are shown in table 4.5.

Width	3
Height	128
Input Size	8

Table 4.5: The parameters used in the neural network.

This network had previously been used to achieve power management in IoT simulations. Its exact performance is unimportant; we are for the moment only concerned with its deployability onto our chosen IoT microcontroller.

We conduct our initial experiment by running this network through the steps described in section 4.2.1. With some effort, the network was successfully compiled and transferred to the Arduino Nano 33 BLE. With a neural network successfully

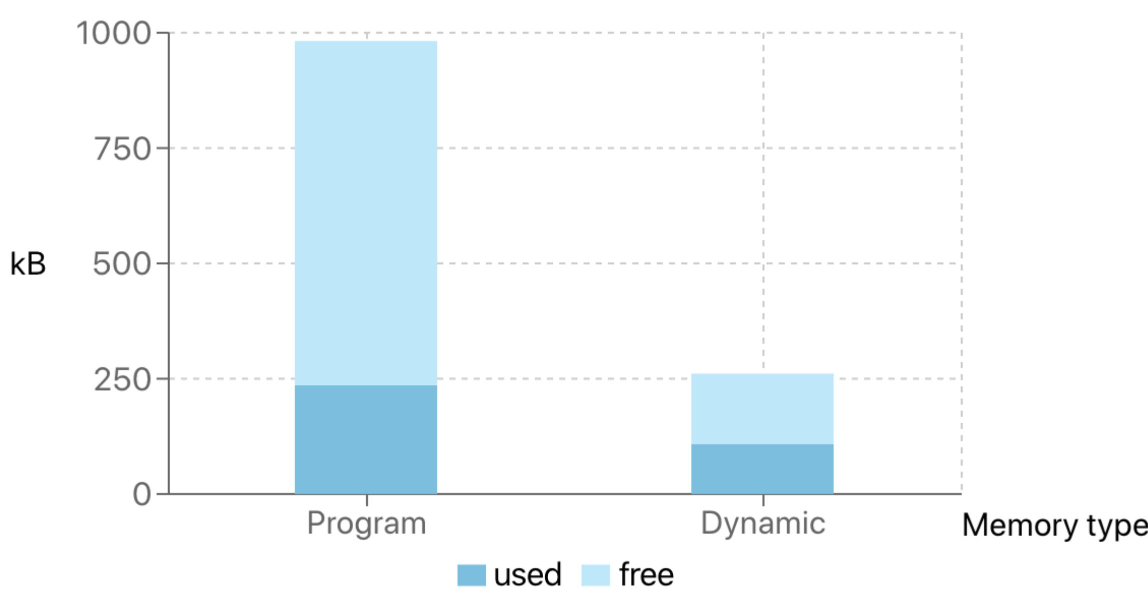

Figure 4.5: The memory consumption of the neural network.

compressed and transferred to a microcontroller, we have reached a major milestone in the project. With this, we can start making observations and measurements. To start, the amount of flash and RAM consumed by the neural network was found by the Arduino IDE at compile time. It is shown in table 4.6, and illustrated in figure 4.5.

Memory type	**Total**	**Taken**	**Free**
Flash	983 KB	236 KB (24 %)	747 KB (71 %)
RAM	262 KB	67 KB (25 %)	194 KB (75 %)

Table 4.6: Memory consumption of our initial neural network in isolation.

We wish to verify that the network produces correct results when deployed on a microcontroller in this manner. The received neural network takes **eight** floats between -1.0 and 1.0 as inputs. The output is a single floating point value, also in the range [-1.0, 1.0]. These values are meant as normalizations of real input and output. The first four inputs might combined represent detailed weather data, for example, and an output of 1.0 might mean a duty cycle of 100 % is selected. We are not overly concerned with the actual meaning of the values at this stage, just how they work so that we might verify our microcontroller implementation.

To verify that the behavior remains correct on a microcontroller, we plot the **expected** behavior and compare it with the observed values. Figure 4.6 shows the correct behavior of the neural network given arbitrarily selected input values. It was observed

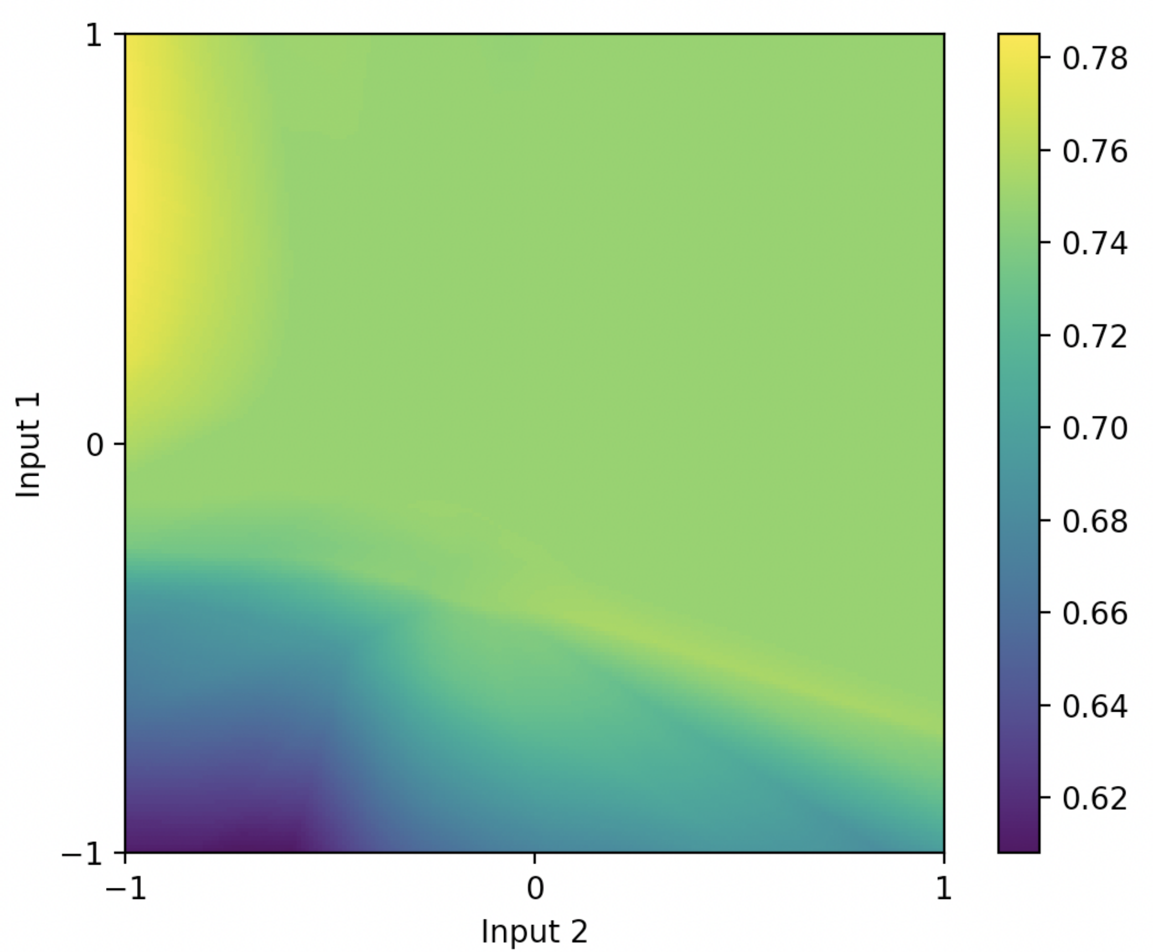

Figure 4.6: The behavior of our initially received neural network. The network takes eight values as input, but for the sake of visualization we sample two and then repeat those.

by loading the model into a Python program on a regular computer running OSX. It was then inferred from without any compression or conversion performed. Two input values, denoted Input 1 and Input 2 in the figure, were incrementally generated. Initially both were set to -1.0, then subsequently increased to produce every combination up to both being 1.0. The granularity was selected as steps of 0.01. For each combination of these two inputs, they were then repeated to the desired input length of eight to achieve enough inputs to make an invocation from the network. Thus, one full input might be:

Input 1: -1.0
Input 2: 0.4
Resulting Total Input: [-1.0, 0.4, -1.0, 0.4, -1.0, 0.4, -1.0, 0.4]

Following this example and varying inputs 1 and 2, figure 4.6 was generated. This sort of test does not provide any insight into how the network operates with realistic input, but it *does* provide behavior that is possible to visualize in a graph like this.

Given this "correct" behavior, we can make sample invocations of the neural network on the microcontroller for comparison. We again follow the arbitrary input pattern used to generate the input. We here choose a granularity of 0.5. Referring to figure 4.6, this should allow us to sample neural network outputs along each axis, as well as in the middle of the graph. We expect an input of [-1, -1] to lead to an output of around 0.60. On the other end, [1, 1] should produce approximately 0.75. Cross-referencing values like this should let us ascertain whether the neural network kept its intended behavior through the compression and transfer process. The result of invocation from our given neural network on the Arduino Nano 33 BLE is shown in table 4.7.

Input 1	**Input 2**	**Output**
-1.0	-1.0	0.61
	0.0	0.68
	1.0	0.70
0.0	-1.0	0.76
	0.0	0.75
	1.0	0.75
1.0	-1.0	0.78
	0.0	0.75
	1.0	0.75

Table 4.7: Memory consumption of our initial neural network in isolation, measured during runtime embedded on an Arduino Nano 33 BLE microcontroller.

We check the values most easily cross-referenced with figure 4.6 from the table. As hoped, an input of [-1, -1] leads to an output of 0.61, spot on with what was expected. [-1, 0] is the halfway point on the x-axis in figure 4.6, which has a dark cyan color. Matching the color bar perfectly, we observe an output of 0.68 when these values are fed into the neural network running on our microcontroller. Continuing this cross-referencing, we can verify that [1, -1], the top of the y-axis, has the maximum observed value of 0.78. Finally, the top right corner corresponding to [1, 1] and the middle of [0, 0] are both part of the wide green area of value 0.75. We thus conclude that indeed, the neural network is producing the same output on our microcontroller as it did on a full-fledged computer. Transfer and invocation of a neural network on an IoT device is a success.

4.2.3 Memory Consumption Boundaries

With the successful result of experiments with a single given neural network, the main remaining part of our experiments is to integrate it with a sense cycle to see whether it can be utilized for power management effectively. We do this in section 4.3. Before that, though, it is worth studying neural networks running on microcontrollers in isolation a little further.

The memory requirements of a neural network are strongly dependant on the structure of the network. The flash memory, used to store a program's code statically, is what's used to store the neurons and their connections. The RAM, used for runtime variables, is needed to load each layer and perform the actual invocation during runtime. As presented in section 2.3.1, the flash memory footprint of a neural network is mostly bound by the *depth* of the network. Correspondingly, the *width* of the network is mostly limited by RAM. We select some Reinforcement Learning algorithm that uses neural networks. For our project, we used the PPO2 algorithm [Schulman et al., 2017], mainly because it had been used with promising results in external IoT power management projects [Murad et al., 2019a]. Other options such as the TD3 algorithm introduces in section 2.2.3 could also have been used. Initial tests with TD3 showed no significant differences in resulting memory footprints, so we assume the PPO2 algorithm sufficiently general.

In order to explore the boundaries of neural network sizes, we apply the brute force method of checking every single combination within reasonable limits. These chosen limits are presented in table 4.8.

Max Depth	5
Max Width	1024
Max Input Size	128

Table 4.8: The selected limits of neural network size throughout our experiments.

The depth was incremented in natural steps of 1, but the width and input size in powers of 2. That is, we tested network widths of 1, 2, 4, 8, 16, and so on for every selection of the other two variables. This means that the smallest possible neural network configuration is depth 1, width 1, and input size 1. On the other hand, the biggest is depth 5, width 1024, and input size 128. Note that the provided network discussed in section 4.2.2 had values quite near the center of this range.

For each configuration, we compute various memory requirement indications. First is the size of the network when stored in a compressed TensorFlow Lite (.tflite) file. This is a quick way to get a rough sense of how the size of the networks compare to each other. Some of the most important peripheral values are shown in table 4.9. The entire generated data set is illustrated in figure 4.7. Note that we use a logarithmic scale in this figure, reflecting the exponential nature of our chosen network widths.

Looking at figure 4.7, there seems to be a relatively direct correlation between network parameters and .tflite file size. It would be useful to produce a way to predict the file size given network architecture, and we use *polynomial regression* [Agr, 2020] to look for a trend. The result is a multi-variable second-degree polynomial function with a root mean squared error of **less than a hundred bytes**. We denote the parameters as:

Depth	Width	Input size	.tflite size
1	1	1	**1 KB**
2	64	16	**20 KB**
3	128	64	**128 KB**
4	512	64	**3.2 MB**
5	1024	128	**16.8 MB**

Table 4.9: Memory consumption of some important network configurations.

$$x := \text{network width} > 0$$
$$y := \text{network depth - 1} \geq 0$$
$$\Pi_t := \text{Resulting file size}$$

The extrapolated formula is as follows:

$$\Pi_t = 4x^2y + 40x + 300y + 1000 \tag{4.1}$$

We thus have an idea oh how the TensorFlow Lite file size increased as neural network size increases. We study this result further in the upcoming section.

4.2.4 Compile-Time Memory

The TensorFlow Lite file sizes discussed in section 4.2.3 are a good indication of how we can expect network size parameters to translate into bytes required to store the network. What's important for us in the end, though, is how much Flash and RAM the networks consume when compiled onto a microcontroller. Thus, we first convert every test-network into C arrays as described in section 4.2.1. We then compile complete Arduino sketches using these ready-to-use neural networks, logging the resulting Flash and RAM requirements. These are the final memory requirements our IoT microcontroller actually deals with. The results for some distinct values are shown in table 4.10.

Depth	Width	Input size	Flash memory	RAM
1	1	1	**98 KB**	**107 KB**
2	64	16	**117 KB**	**107 KB**
3	128	64	**235 KB**	**107 KB**
4	256	128	**895 KB**	**107 KB**
5	1024	128	–	–

Table 4.10: Memory consumption of some important network configurations. Cells without entries denote network configurations for which compilation or transfer was impossible.

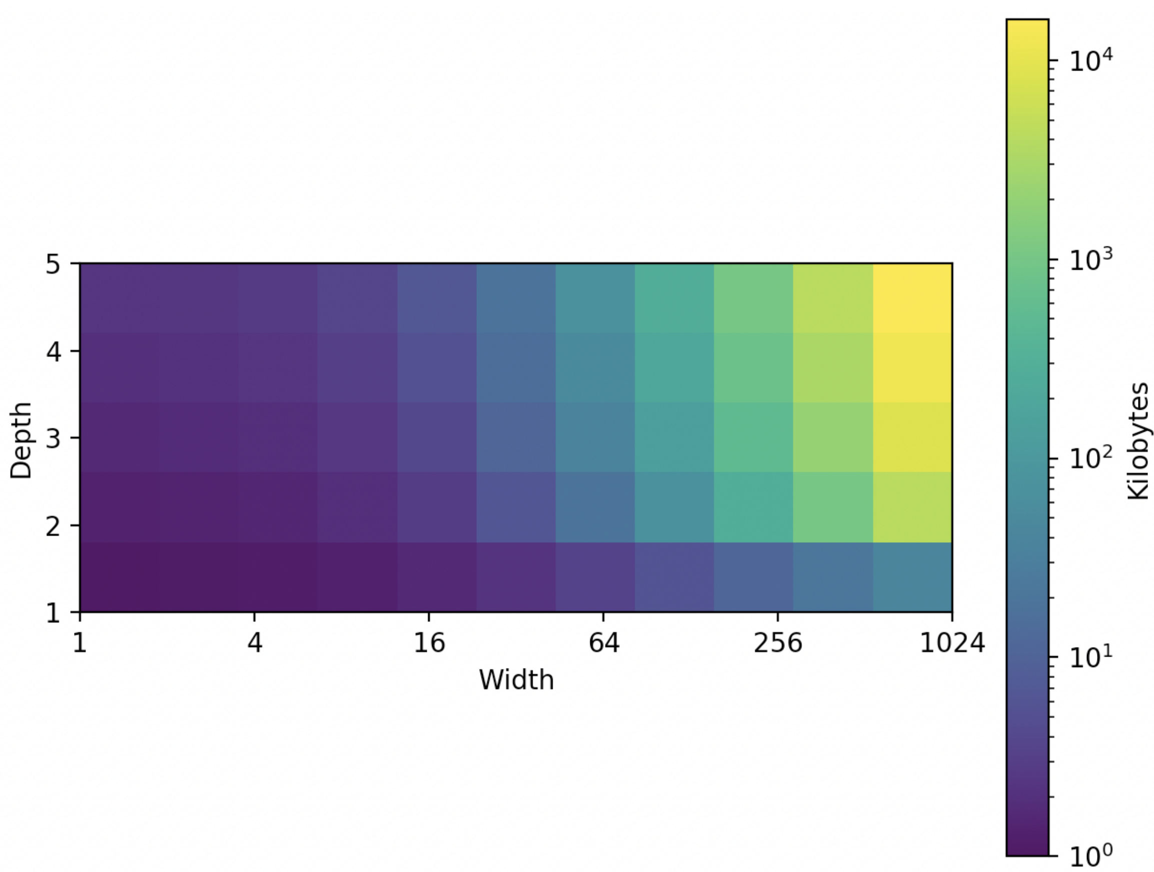

Figure 4.7: The file size of neural networks of various configurations when stored as compressed TensorFlow Lite files.

There are several interesting points to note about table 4.10. First, notice that the RAM requirements are constant for every entry in the table. Upon inspection of this phenomenon, we found that TensorFlow Lite Micro works by allocating a set amount of RAM to the neural network at compile time [**?**]. This amount is configurable. It just so happens that the default amount used corresponds to a RAM footprint of 107 KB, which is around 40 % of the Arduino Nano 33 BLE's RAM. Every neural network we compiled compile was able to function properly given the default amount of allocated RAM. The implication of this is that the neural networks likely could have functioned with a lower amount of allocated RAM. This is a useful observation to note for developers of neural networks on microcontrollers, in case RAM should ever become the bottleneck for a given application. For our purposes though, we mainly note that the Flash memory is at approximately 900 KB for the largest compiled network, not far from the limit of 1 MB. For this same network, 40 % of the RAM was enough for invocation. Thus, it is reasonable to conclude that for our selected boundaries given in table 4.8, RAM is not an obstacle for the transfer and invocation of a neural network. This is an important result.

The second interesting point to note about table 4.10 is that there are cells with no entries. The reason for this is that the computer used for generating these samples could

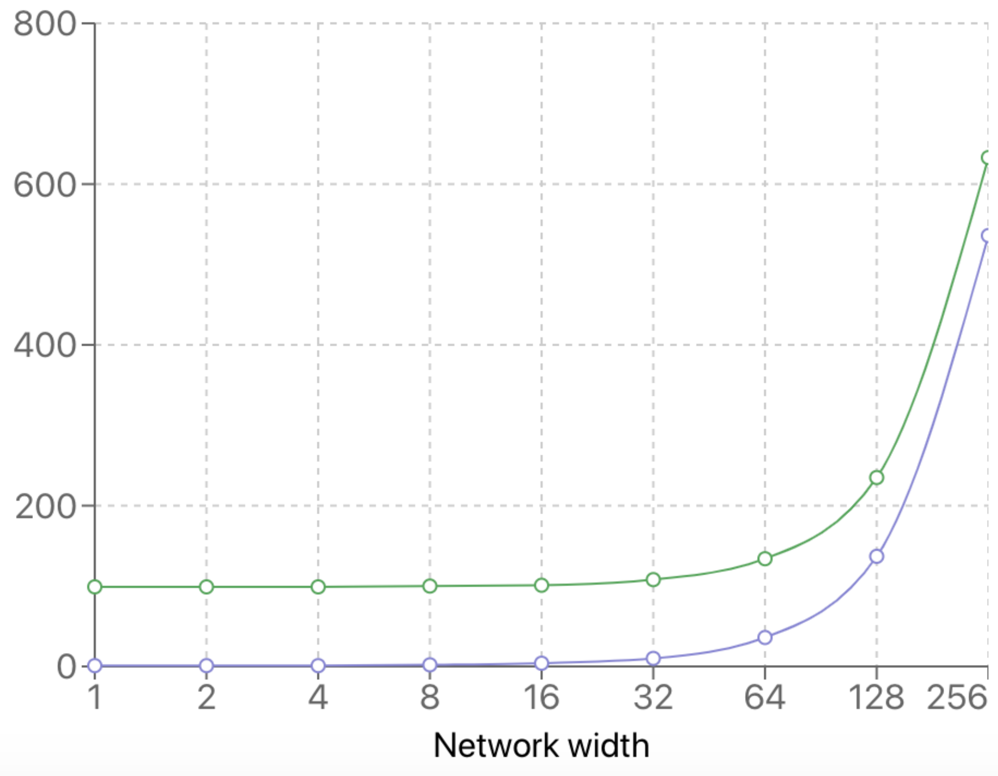

Figure 4.8: The tflite file size (blue) and final flash memory requirements (green) of neural networks with depth = 3 and various widths.

not handle computing networks of this size. Specifically, the process required more than the 16 GB of RAM available. Acquiring better hardware to perform tests of neural networks this large might be an interesting point of further study. However, there is another interesting observation we can make from the table that might render this unnecessary. When plotting the previously acquired .tflite file size against the resulting flash memory requirement, we see an interesting result. Figure 4.8 clearly illustrates that there is a direct relationship between TensorFlow Lite file size and compiled Flash memory requirements. Computing the difference between them at a wide range of neural network configurations, we find that there is a difference of exactly **97488 bytes** each time. That is, **the Flash memory consumption is always approximately 100 KB lower than the TensorFlow Lite file size**. With this result, we can adapt formula 4.1 into one that gives us the final Flash memory requirement of a neural network:

$$x := \text{network width} > 0$$
$$y := \text{network depth - } 1 \geq 0$$
$$\Pi_f := \text{Flash Memory Requirement}$$

$$\Pi_f = 4x^2y + 40x + 300y + 98488 \tag{4.2}$$

The constant 96488 is found by adding the constant 1000 in equation 4.1 to the observed difference 97488. It is worth noting that this value could likely change depending on a range of factors, including which hardware platform is used. We keep the specific value for our use in our project, but those who would adapt it should calculate their own constant $k = 1000 + \Delta_{tflite,flash}$.

Thanks to this direct relationship, we can use our observed TensorFlow Lite file sizes to calculate Flash memory requirements of the networks we were unable to compile. It can see from the TensorFlow Lite file sizes in table 4.9 that the networks at the top end of our test range vastly outgrow the boundaries for deployment onto our microcontroller. The largest network has a .tflite file size of 16.8 MB, meaning the corresponding Flash memory requirement is roughly 16.8 - 0.1 = **16.7 MB** – more than 15 times larger than what Arduino microcontroller can handle. It is thus of little consequence that we were unable to compile the larger networks for testing.

We would like to find the point at which the network dimensions grow too large. Formula 4.2 has two variables, so we fix the depth at different configurations and see how the width affects the resulting Flash memory size. Figure 4.9 displays the mathematical curves produces by applying this formula for depth > 1. For the special case of depth $= 1$, where we are not dealing with deep learning, figure 4.10 illustrates that the curve grows linearly. We can thus note the interesting fact that it is the inclusion of more than a single hidden layer in a neural network that causes the growth of the file size to become polynomial in nature. We can see this reflected in equations 4.1 and 4.2 – the term y becomes 0, canceling the only term with an exponent greater than one.

We observe from figure 4.9 that the larger the network depth, the lower amount of neurons we're able to fit per layer. This matches what we might expect intuitively – more layers means fewer neurons per layer. To find the exact limits for our given experimental setup, we fix y and solve equation 4.2 $= 1024$ for x. The results are shown in table 4.11.

Network Depth	**Width Limit**
1	23 752
2	482
3	342
4	280
5	242

Table 4.11: The width limits of a neural network given different depths, assuming they are required to fit onto a 1024 KB Flash memory.

We quickly mention the obvious oddity of table 4.11: the first entry. A depth of 1 gives

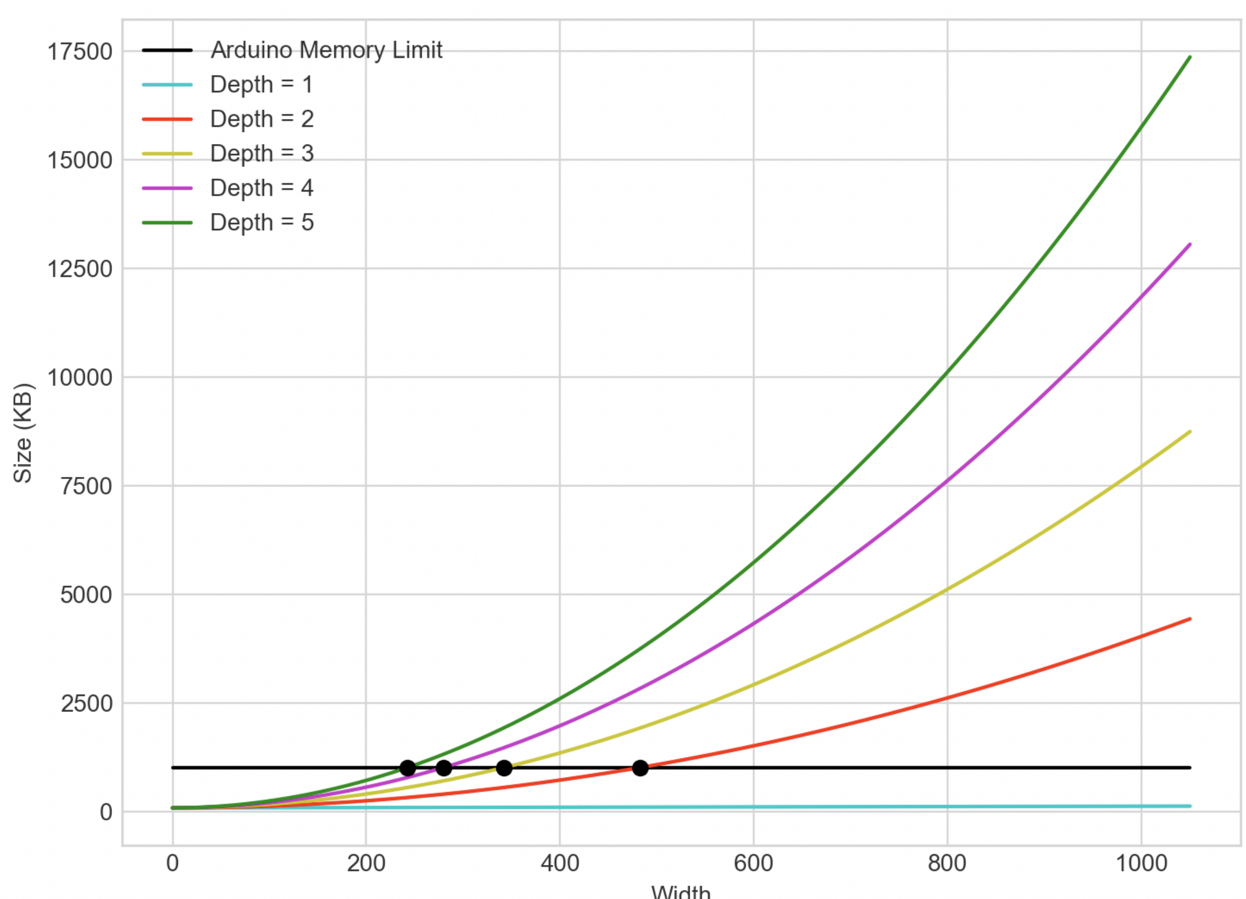

Figure 4.9: The Flash memory requirement of neural networks as a function of network width given five different network depths. The black line indicates the Flash memory limit imposed by our chosen microcontroller, 1024 Kilobytes. The point at which each network configuration exceeds this limits is indicated.

a width limit of more than 23 000, significantly larger than all other entries. This is due to the difference between linear and polynomial growth, as described. To confirm whether this was a mistake in our math, though, we perform tests using networks near these boundary values. Some overly hardware-specific details caused the numbers to be slightly off – for example, the Arduino Nano 33 BLE's actual available memory after allocation reserved for the OS and other overhead was around 950 KB. This is a bit lower than the 1024 KB described in the microcontroller's manual. This caused the limit to drop slightly. That being said, a neural network of depth 1 and width 22000 was successfully transferred to the microcontroller. Accurate invocation was then performed. A similar test with width 23000 yielded the following error: *region 'FLASH' overflowed by 126288 bytes*. This matches our expectations.

We then repeated the test with depth = 2, ascertaining whether the transition to deep learning indeed caused the file size growth to become polynomial. The networks we trained had widths in powers of 2, so we tried compiling networks just above and below the indicated threshold of 482 (table 4.11). As hoped, the network with

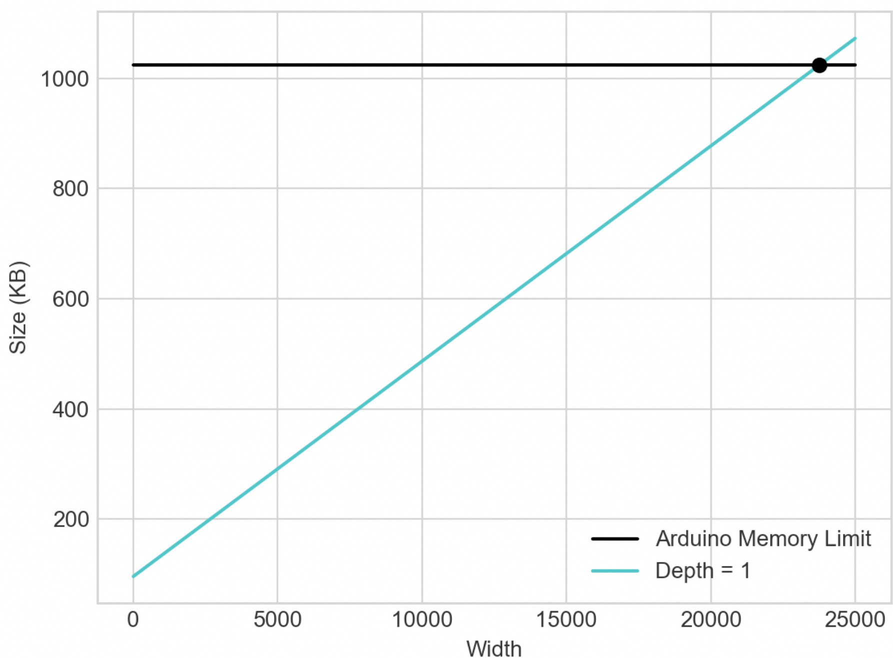

Figure 4.10: The Flash memory requirement of neural networks as a function of network width given depth = 1. The black line indicates the Flash memory limit imposed by our chosen microcontroller, 1024 Kilobytes. Note that the x-axis needs to extend significantly further than in figure 4.9 to reach the point where the lines meet.

depth 2 and width 256 compiled, while the one with width 512 exceeded the device's Flash capabilities. These data points indicate that both our deduced formula and the subsequently calculated limits reflect the microcontroller's actual limits.

It is worth noting at this point that we were able to invoke from these networks without extending the RAM allocated to the network beyond its default value. This supports our assertion that RAM is never the bottleneck for neural network size, as the flash memory requirement always becomes a barrier first. Of course, this only holds within our chosen boundaries.

The network size limits presented in this section are useful in the short term for development on the project that sparked this thesis. More importantly though, the **trends** shown in this section's figures and tables are applicable to a wider range of use cases. Different hardware and configurations will yield different exact limits, meaning the cutoff points for network size will wary. However, regardless of hardware, the provided formula and methodologies can act as general guidelines for how complex neural networks a developer can expect to fit on his on her device.

In this section we have provided data on how neural network size affects Flash and RAM when compiled into a form usable on microcontrollers. At this point it might be natural to move on to a discussion of the **energy consumption** of these same neural networks. This is highly relevant, and we will discuss it. However, the difference

between the energy consumption of neural networks in isolation versus that of an integrated power management system is nothing more than the static contribution of a sense cycle. Moreover, looking at neural network runtime in the context of power management context lets us compare the runtime of a single program both with and without invocation. This should let us identify the time taken by just the neural network, instead of mistakenly including the runtime of other unrelated processes. For these reasons, we save discussion of neural network energy consumption to the end of section 4.3.

4.3 Power Management Implementation

Now that we have looked at the hardware requirements of both a sensing cycle application and invocation of a neural network in isolation, it is time to bring them together. Our goal in this section is to present the result of integrating a sense cycle with a neural network to provide a realistic IoT application that achieves intelligent power management on real hardware.

To end up with such a power management program, we simply combine the code from the previous two sections. The only added functionality is using the actual invocation of the neural network to determine time to sleep between each loop of the sense cycle. This is our implementation of a duty cycle. The result of the code integration was a successful IoT application utilizing a neural network for power management. This is what we wanted to achieve, although we have yet to ascertain whether the performance is efficient or intelligent. We look at the memory and energy consumption of the program to provide a starting point for this evaluation next.

4.3.1 Total Memory

We want to evaluate how much memory the finished program consumes at runtime. In this section, we focus on the externally provided neural network that has been shown to produce intelligent power management behavior in simulations [Murad et al., 2019a]. We have already looked at how memory footprints can vary for both the sense cycle and the neural network, and the methodology for combining them is what we analyze here. Combining this methodology with the generalized approaches to memory size will allow us to discuss the interplay and trade-off between the two artifacts, which we do in chapter 5. For now, though, we only concern ourselves with a single sense cycle and a single neural network.

In order to deduce the memory consumption of a combined power management application, it is insufficient to naively add the memory requirements of its two parts to each other. The reason for this is that there is some common overhead when transferring any kind of runnable code. To analyze this overhead, we compile and transfer an entirely empty program with no dependencies:

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

void setup() { }

void loop() { }
```

This empty program still has a memory footprint, presented in table 4.12:

Memory Type	**Overhead**
Flash	76 KB (7 %)
Ram	42 KB (16 %)

Table 4.12: The unavoidable memory overhead of an Arduino sketch when compiled for the Arduino Nano 33 BLE.

We consider this the common, unavoidable memory overhead for our specific setup. 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:

$$M_{tot} = M_{sense} \cup M_{nn} \tag{4.3}$$

$$= M_{sense} + M_{nn} - M_{sense} \cap M_{nn} \tag{4.4}$$

As an initial data point, we use the example **sense cycle** program along with a neural network of width 3, depth 128, and input size 64. This network has shown power management behavior in simulations, and analyzing the memory footprint of this combination should give us an indication of how realistic our approach is. Inserting our collected data about flash memory into formula 4.4 gives us:

$$M_{flash} = 292KB + 236KB - 76KB$$
$$= \mathbf{452\ KB}$$

Likewise, the collected data for the RAM gives us:

$$M_{ram} = 67KB + 108KB - 42KB$$
$$= \textbf{133 KB}$$

These findings are summarized in table 4.13.

Memory	Overhead	Sense cycle	Neural network	Total
Flash	76 KB	292 - 76 = 216 KB	236 - 76 = 160 KB	452 KB (45 %)
RAM	42 KB	67 - 42 = 25 KB	108 - 42 = 66 KB	133 KB (50 %)

Table 4.13: Memory consumption of the various parts of our experimental program.

This percentage of total memory used is shown in figure 4.11. As we can see, around half of the flash memory is consumed, while a bit more than half of the RAM is allocated. To be precise, 54 % of the flash memory and 60 % of the RAM is spent on our IoT application. Further, the distribution of the memory allocation is shown in pie charts 4.12 and 4.13. Figure 4.12 shows the flash memory allocation, and figure 4.13 shows the dynamic RAM.

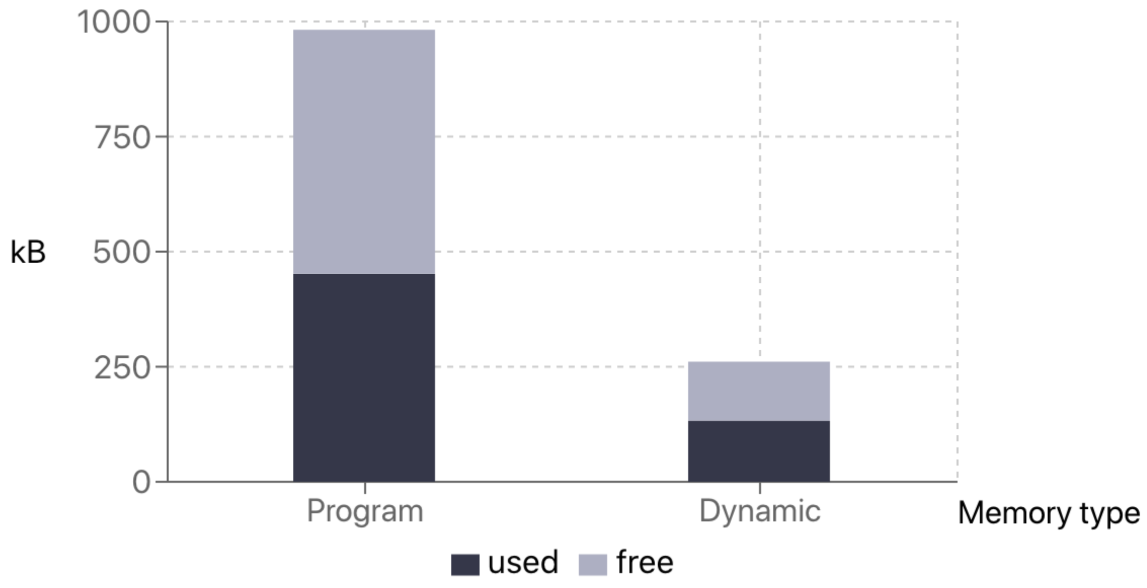

Figure 4.11: Total memory consumption at runtime.

To test the accuracy of this approach, we compile the actual combined power management application and log its memory footprints. The result is the console output shown in figure 4.14. The values match exactly what we computed earlier. We did so by first finding the common overhead through an empty program, assuming it

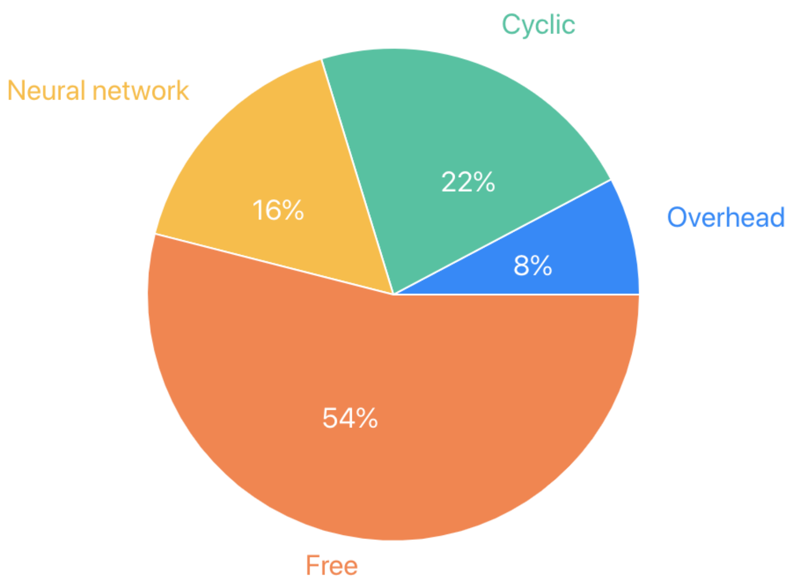

Figure 4.12: The distribution of flash memory at compile time.

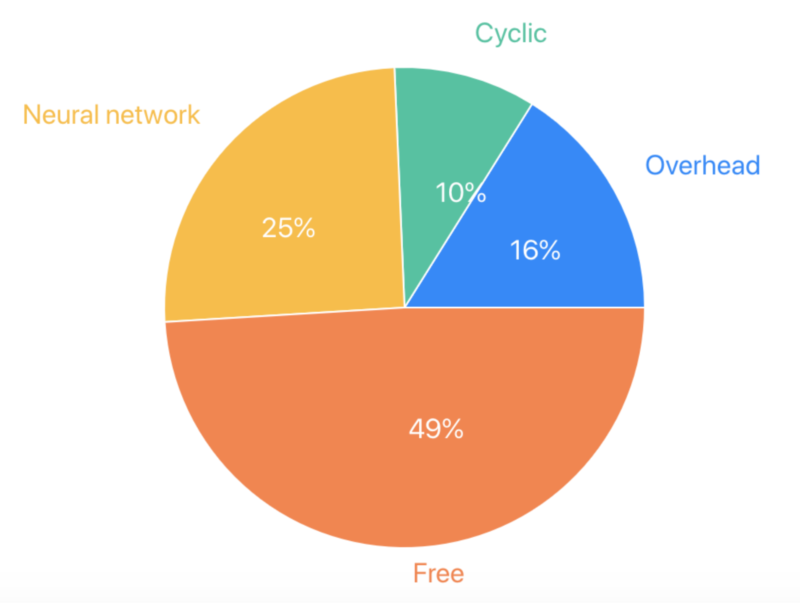

Figure 4.13: The distribution of RAM at compile time.

```
Sketch uses 451520 bytes (45%) of program storage space. Maximum is 983040 bytes.
Global variables use 133096 bytes (50%) of dynamic memory, leaving 129048 bytes for local variables.
```

Figure 4.14: The console output produced by compiling the final combined power management application.

to be the intersection of two sets corresponding to a sense cycle and a neural network, then calculating the union of these two sets. The fact that this calculation matched observed experimental value means that our assumption was correct. The common overheads presented in table 4.12 are correct.

4.3.2 Total Energy Consumption

The final section of this chapter deals with perhaps the most important result of our experiments: how much energy the neural network power management system consumes. Measuring this will tell us the energy cost of implementing the intelligent power management, which is crucial if we are to make any statement about the usefulness of the approach. No matter how much energy the neural network manages to save, doing so is pointless if it consumes as much or more energy in the process. We still work with the assumption that the energy consumption per time is roughly equal for different processes, meaning energy consumption is directly correlated with CPU runtime. This runtime is what we actually measure.

The IoT application is naturally divided into distinct phases, and we measured the time spent by each phase distinctly. The main phases are the sense cycle and the power management using our neural network. The total time spent on either of these phases was measured in each iteration. In addition, however, we look at the most important sub-processes constituting those two phases. In particular, we log the time taken by Bluetooth communication of the sense cycle, and by the invocation of the neural network. First, Bluetooth communication is interesting because it's based on using a separate piece of hardware, and it is the most likely candidate for violating our assumption of constant energy consumption. This value should not change based on neural network size, however, and we consider its discussion in section 4.1 sufficient.

Next, we say that we measure the invocation of the neural network as a sub-phase of the overall power management. The goal of this is to ensure that we get a measurement that includes as little else than the actual invocation as possible. In contrast to this, the overall power management phase also includes interpreting the invocation's output, selecting a new duty cycle, and more. While the time taken by this entire phase is interesting, as it is the actual time a developer will have to account for if they want power management, it is more prone to variation based on the code we wrote. It is to a larger degree specific to our experimental parameters. The invocation of the neural network itself, however, is a more pure data point that can let us make more general conclusions. It is for this reason we measure both.

The time spent by these phases on the CPU were logged for a wide variety of neural networks. Knowing whether network size affects runtime, and thus, energy consumption, is an important result. We followed the same patterns as before, using the limits in table 4.8. Depth was increased in increments of 1, and width in powers of 2. One difference from earlier configurations is that the input size was found to not affect the runtime, so it is omitted from our input parameters. Each configuration was allowed to run 30 times before termination. From each of these data sets, we calculated means μ and standard deviations σ. We denote the power management as pm, using the subscript μ_{pm} to denote its calculated mean and σ_{pm} as its standard deviation. The same subscript is used for the invocation, denoted i. Using these configurations, we performs a total of 5 width configurations * 11 width configurations * 30 iterations = **1650** straight-forward tests of runtime. The results of some select configurations are shown in table 4.14.

Depth	**Width**	μ_i	σ_i	μ_{pm}	σ_{pm}
	1	0 ms	0 ms	3.0 ms	0.0 ms
1	256	1 ms	0 ms	4.0 ms	0 ms
	1024	2 ms	0 ms	5.3 ms	0.5 ms
	1	0 ms	0 ms	3.0 ms	0.0 ms
2	256	12.5 ms	0.5 ms	15.5 ms	0.5 ms
	512	–	–	–	–
	1	0 ms	0 ms	3 ms	0.3 ms
3	256	24.2 ms	0.4 ms	**27.3 ms**	0.5 ms
	512	–	–	–	–
	1	0.2 ms	0 ms	3.0 ms	0 ms
4	256	9.3 ms	0.5 ms	12.5 ms	0.5 ms
	512	–	–	–	–
	1	0 ms	0 ms	3 ms	0.0 ms
5	128	12.8 ms	0.4 ms	15.6 ms	0.5 ms
	512	–	–	–	–

Table 4.14: Measured invocation runtime of neural networks of various sizes.

One obvious result that can be observed from table 4.14 is that, as before, there are blank entries in the table. As could be expected, we are unable to compile networks exceeding the limits presented in table 4.11. For the values we *could* compute, though, we observe an interesting trend: the runtime of invocation does indeed seem to increase with network complexity. This might have been intuitive, but it is an important result to show experimentally. Further, comparing μ_{pm} to μ_i, we see that μ_{pm} follows the same development as μ_i, plus 3. Thus, we arbitrarily choose to focus on one of them, landing on the overall power management pm. These are the values shown in the final two columns in table 4.14. The development of these run times given every tested neural network configuration is illustrated in figure 4.15.

We notice from figure 4.15 that the development of runtime as a function of depth and

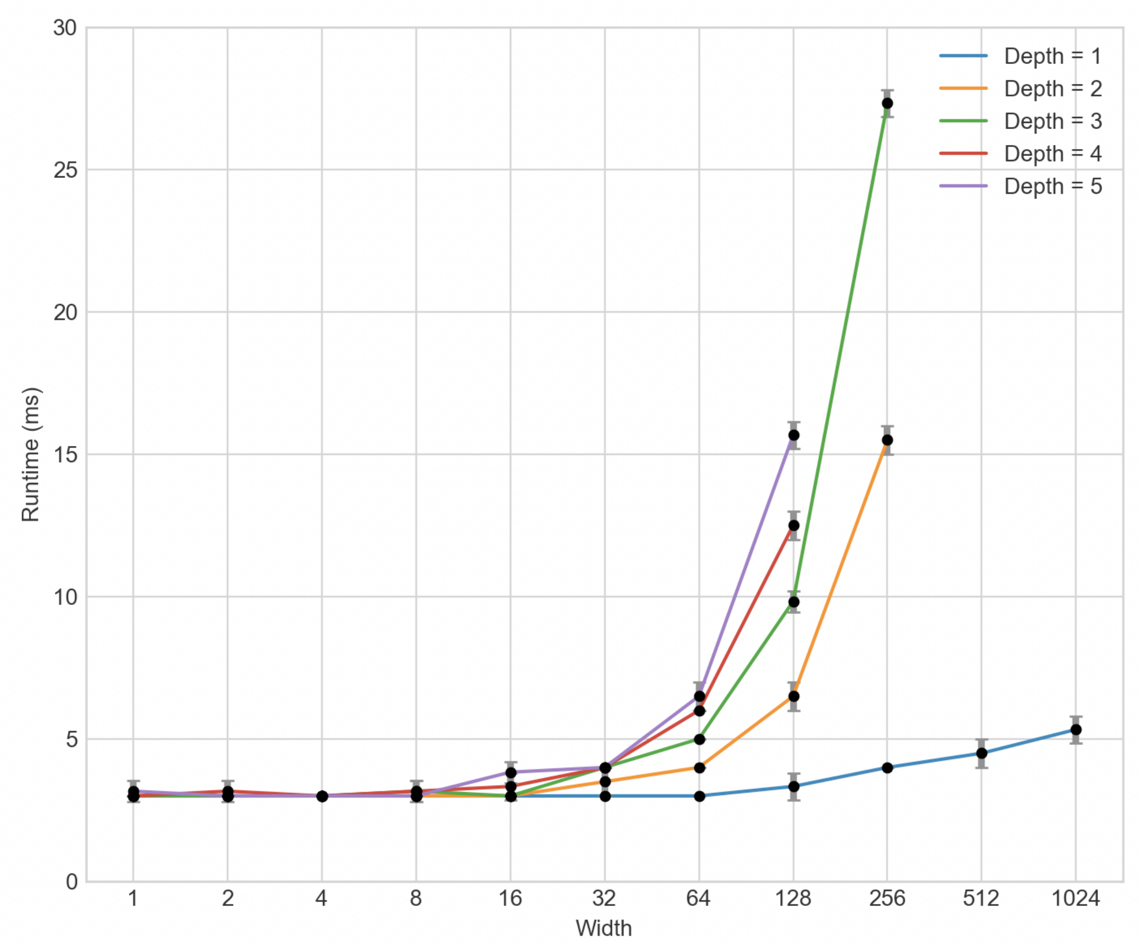

Figure 4.15: The runtime of the invocation of neural networks of various configurations. Each data point represents the mean of a sample size of 30 runs for that width / depth combination. The corresponding variance, expressed as standard deviation, is indicated through grey vertical lines.

width seems polynomial. The intuition is the same as when we made this observation for compiled flash memory consumption: we use polynomial regression to see if we can produce a mathematical formula predicting this behavior [Agr, 2020]. The result is equation 4.5.

$$\begin{aligned} x &:= \text{network width} > 0 \\ y &:= \text{network depth} > 1 \\ \Pi_e &:= \text{Runtime on Arduino CPU} \end{aligned}$$

$$\Pi_e = \frac{x^2y + 15xy^2 - 25xy + 50x + 30000}{10000} \tag{4.5}$$

Note that this formula only holds true for depth > 1. We can see from figure 4.15 that depth $= 1$ follows a fundamentally different curve than the others, reflecting the

stark difference in regular and *deep* reinforcement learning. We could have presented a formula that accounts for this and allows calculation of runtime for networks of depth = 1, but unlike equation 4.2, the result would not be nearly as neat as the more restricted equation 4.5. As we are mostly concerned with deep learning in any case, we stick with this representation. Note that the constant 30000 could be omitted to achieve a formula that predicts the runtime of neural network invocation in isolation, although somewhat less accurately.

In this chapter, we have shown the process used to conduct our experiments. Each step of each process has been outlined, and detail has been provided where it has been crucial for reproduction. The results have been presented as a series of tables showing select raw data, figures giving visual representations of the entire produced data sets, and mathematical formulas extrapolated from the data that represent the behavior in a generalized manner. These results form the foundation for a discussion of our initially proposed research question, along with the feasibility criterion and design science treatments described in chapter 3. This discussion is presented in chapter 5.

Chapter 5 Discussion

Chapter 5 provides a discussion of what conclusions we can make from the data provided in chapter 4. We do this in the order specified by the feasibility criterion presented in chapter 3. Section 5.1 first handles the question of whether it is possible to implement neural networks on IoT devices. Section 5.2 then looks at the performance of these neural networks, establishing methods for determining which settings and parameters are needed for the approach to be a good fit. Section 5.3 presents a case study utilizing these methods. Section 5.4 concludes by revisiting our proposed research question, using accumulated data and discussion to evaluate to which extent we are able to answer the question.

Throughout this chapter, we use the principles and terms of Design Science to guide our discussion. We consider the following to be our main artifacts: the neural network; its runtime parameters; and our chosen hardware. We consider the different ways these can be combined to produce intelligent power management systems as our *treatments* to be revised and iterated upon. Using the experimental data provided in chapter 4, we wish to **validate** our treatments by looking at how much energy they are able to save. We use the feasibility criterion defined in chapter 3 to discuss the merit of each of these artifacts and treatments. Finally, we wish to look at our work in the context of the larger project we are a part of. That project also uses an engineering cycle, and our work constitutes validation of provided treatments in this larger scope.

5.1 Fitting Neural Networks into Device Memory

We first answer whether neural networks fit in the memory of IoT devices at all. To determine how much memory these neural networks had to work with, we first implemented a static application on our IoT device. Experimental data for this *sense cycle* was generated in section 4.1. We now look at what conclusions this data can lead to.

5.1.1 Memory Required by a Sense Cycle

Going back to the phrasing of sub-criterion 1.1, we recall that it read:

Verify that the static code constituting a sense cycle can be transferred and run on an IoT microcontroller without significant resource consumption.

Whether this criterion is fulfilled depends on what exactly is meant by *the static code constituting a sense cycle*. If we assume the specific sense cycle we actually developed in section 4.1, we already have the answer. The application was shown in that section to be runnable on the given hardware. Its memory and runtime footprints were presented, neither of which constituted a large percentage of the hardware's capabilities. We thus directly conclude the criterion fulfilled in the case of our specific experimental sense cycle, which is intended to be representative of IoT applications.

In the more general case, we explored the memory limits of a sense cycle in section 4.3. There, we showed the remaining available memory given a specific power management neural network – around 30 %. Thus, assuming this neural network is representative of the size we can expect power management to consume, the stakeholder would have 70 % available for their regular application. This upper limit for sense cycle size is of course directly dependant on the size of the network. As described in section 4.2, we can increase the size of the neural network until it fills 100 % of both the flash memory and the RAM if we want to. The trade-off then becomes the added benefit of increasing network size versus the space required for a working sense cycle application.

Our experiments indicate that the sense cycle application size is unlikely to reach a size where it severely impacts the possible configurations of a neural network. If this were to change, we would combat it using the strategy posed in *proposition* criterion 1.2: *reduce the scope of the program so as to less accurately reflect a real IoT node's function*. This is a clear guide to follow for the case that the sense application becomes too large. We thus consider the discussion of the memory required by a sense cycle complete.

5.1.2 Fitting a Neural Network in the Remaining Memory

We defined another sub-criterion relating to transferal to a microcontroller. It focuses on the *neural network*, rather than the sense cycle, and it reads as follows:

Verify that neural networks of appropriate size fits on an IoT's device flash memory. Further, verify that invocation does not require more memory than available in the device's RAM.

This is a two-part question. We need to look at both the Flash and RAM boundaries in order to ascertain whether the criterion is fulfilled. We consider RAM first, as it is more straight-forward. We asserted multiple times throughout chapter 4 that given the neural network boundaries we chose (table 4.8), RAM never became the bottleneck. The Flash memory limit of 1 MB was always exceeded before the 256 KB RAM limit showed signs of becoming an issue. We thus conclude that for our selected hardware and network size boundaries, the RAM aspect of sub-criterion 1.2 is fulfilled.

In the general case, several parameters could change to make RAM problematic. For one, different hardware might have a different ratio of Flash and RAM. A microcontroller with 1 MB Flash and 16 KB RAM, for instance, would likely experience RAM as the bottleneck. Moreover, changing what boundaries we consider realistic and relevant could affect the result. We found in section 4.2.4 that the network width was bound by the Flash memory size, growing too large at around width = 23000. The work on microcontrollers in Berg's work indicates that this limit is approximately 40 000 given similar hardware boundaries as used in our project [Berg, 2019]. Thus, given only somewhat different boundaries, our conclusion that sub-criterion 1.2 is verified might change. In such a scenario, we would again turn to the appropriate proposition criterion for guidance on how to remedy the situation. Proposition criterion 1.2 gives us a straight-forward solution: *If either memory constraint is violated, specify either how much the hardware would need to improve or the network size to be reduced for a fit.*

The second part of sub-criterion 1.2 is whether the neural networks fit onto the microcontroller's *flash* memory. Unlike with RAM, we found that there are some configurations within our boundaries for which the neural network does **not** fit. These limits are an important result, and they are presented in table 4.11. For neural networks exceeding these limits, we can not say sub-criterion 1.2 is verified. However, we knew from the outset that there would have to point at which a neural network gets too large to use in an IoT context. The important question is whether neural networks that will actually be used for power management fit – called *neural networks of* ***appropriate size*** in the sub-criterion. We take our externally provided neural network a reference point. We recall that it had a width of 3 and a depth of 128 – meaning it is well within the computed limits. We studied its exact memory consumption further in section 4.2, and we found that it only consumed approximately 16 % of available flash memory. We use this result to conclude sub-criterion 1.2 verified, keeping in mind that it only holds for neural networks within the limits of table 4.11.

With this, we have looked at every part of every sub-criterion constituting feasibility criterion 1. For our selected experimental parameters, we found that they were all verified. We can thus conclude that the overarching main criterion is verified as well: we are able to both transfer and infer from neural networks on a microcontroller. This was a major milestone along the path to verifying the neural network as a *treatment* in the context of the larger project. The network had been shown to produce promising results in simulations, but the developers had no idea whether the system could actually

be used on IoT devices. For each aspect of the transferral process that might go wrong, we have also established which steps would need to be taken in order to improve the situation. We thus move on to the discussion of the **performance** of our neural network power management system with the knowledge that within our boundaries, the transferral aspect of our project is thoroughly examined and unproblematic.

5.2 Power Management Performance

We now move to discuss the *performance* of our neural network-based power management. This topic is introduced by sub-criterion 2.1:

Verify that there are possible configurations in which a neural network is able to help save more energy than it consumes.

When determining whether the neural network *helps save more energy than it consumes*, we need to look at how much energy the various parts of an IoT device's life cycle consumes.

5.2.1 Energy Budget of an IoT Device

We want to examine the effect of a neural network as part of the energy budget of an IoT device. On one hand, the intention of the neural network is to help the device save energy. It does so by adjusting the duty cycle or by not sending data of low value, for instance. On the other hand, the invocation of the neural network *consumes* energy, as it requires the IoT device to conduct computations. Obviously, for power management via neural networks to be beneficial, the invocation of the neural network must not consume more energy than it helps save. In other words, the power management policy executed by the network must have a certain minimum performance compared to a non-neural-network solution to break even. To discuss this for a wide variety of platforms and independent of the performance of a specific policy, we will discuss first the minimum required performance of a policy compared to the energy consumption of a sense cycle. In later sections, we will then study how these methods apply to specific use cases.

To facilitate the discussion of an IoT device's energy budget and the neural network's role in it, we first need to clearly define what processes are part of this budget. This will let us distinguish between the neural network and everything else running on the device, which is necessary if we are to discuss both the neural network's benefit and its drain on the system at large. These values will let us determine whether the neural network is beneficial enough to be worth including as a power management system.

Figure 5.1 illustrates a starting point for defining what constitutes a full IoT life cycle. It shows the different phases of the loop of a typical IoT device, including a sensor

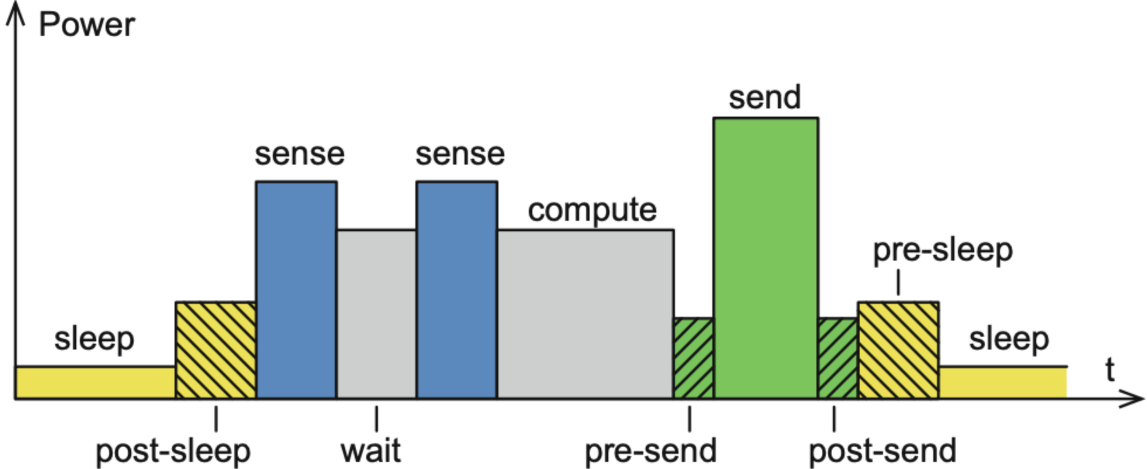

Figure 5.1: An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019].

scan, network communication, and sleeping. In our case, we are mainly interested in the invocation of a neural network. This will correspond to a single such phase, and it is the only distinction that is important for our discussion. We thus abstract the rest of the life cycle into a single phase, which is what we call the *sense cycle*. We assume the sleeping phase consumes negligible energy, and thus do not include it in this definition. The resulting abstraction is reflected in figure 5.2.

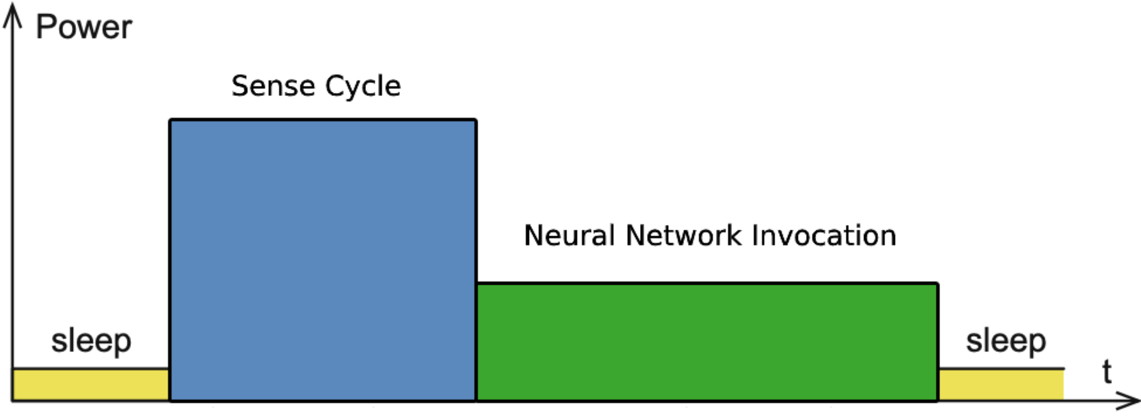

Figure 5.2: A further abstracted model of the energy consumption of different phases in an IoT sensing node's life cycle. Power and time consumption are merely indicative. Based on [Tamkittikhun, 2019].

With this definition of a sense cycle, we have removed any distinction between processes such as sensor scans and network communication. While these might have vastly different energy requirements in reality, it does not make a difference for our discussion. We simply find the average power consumption this sense cycle, and we calculate the amount of energy saved by the neural network based on this amount.

We have thus established a clear idea of how to approach the discussion of a neural network's performance as it relates to the surrounding IoT application while focusing only on the relevant details.

5.2.2 Energy Saved by the Neural Network

We now wish to find the parameters for which using neural networks for IoT power management makes sense – that is, which parameters lead to a reduction in overall energy consumption. These parameters include the sense cycle and neural network *sizes* explored so far, and we have already looked at how these affect both memory footprint and runtime in detail. In this section, we look at two important parameters that have so far gone ignored: first, *how often* the application should ask the neural network for updated runtime configuration. Second, given this frequency, *how much* energy the power management needs to save in order to net gain energy. These parameters will help us define when neural network power management is appropriate for the IoT field, as well as when it is not. The rest of this section focuses on these parameters.

The power management systems considered in this project are implemented through duty cycles – how often the loop of a cyclic application should run. If this frequency is set to a static value for say, a year, we cannot say we are using an *intelligent* power management. This approach is what the static algorithms described in section 2.1.1 did, and they lead to an unadaptive system. Instead, we wish to invoke a neural network to get an updated, optimized duty cycle given the current weather, battery level, and other inputs available to the neural network.

This leads to a crucial question: **how often should the neural network be invoked?** It could for example be asked to update the duty cycle every minute, every hour, every day, or even less frequently. We call this invocation frequency ρ, defined as the ratio of loops *with* and *without* neural network invocation. Let us first consider the extreme options for ρ. On one end, the maximum value ρ can take is 1, meaning the neural network is asked every single loop of the sensing application. In this scenario, the neural network would decide the time to sleep between *every* iteration of the loop. This would allow the power management system to make as finely granulated decisions as possible. In theory, this approach should allow the most efficient power management possible. However, there is a trade-off. The invocation of neural networks costs some amount of energy. If it consumes more than it saves, we are not accomplishing anything of value and sub-criterion 2.1 is *not* fulfilled. Invoking the neural network less often can thus be interesting, leading to less power consumed for potentially similar amounts of energy saved. As an extreme example, $\rho = 0.001$ would mean we only invoke the neural network every thousandth iteration. In this case, the power management system would need to save very little power to be worth including.

We wish to determine how much energy a neural network needs to help the IoT device save in order to break even. To do so, we need to determine the energy consumption

of both the sense cycle and of the neural network itself. We ask how much energy each phase consumes in an *average iteration* of an IoT application's loop. The answer is the phase's energy consumption rate times its runtime, and the result is measured in milliwatt-seconds (mWs) or microwatt-seconds (μWs). We denote this total energy consumption per loop E_{sense} for our sense cycle, and E_{nn} for our neural network phase. Building on this, E_{tot} is the total energy consumption of an iteration. We only incur the energy cost of the neural network whenever we actually invoke it, and we must reflect this in the formula for E_{tot}. We defined this ratio as ρ, meaning its average contribution to energy consumption is ρE_{nn}. We thus get $E_{tot} = E_{sense} + \rho E_{nn}$.

Rather than finding absolute values for E_{sense} and E_{nn}, we are more interested in the *ratio* between the two. Whether the sense cycle consumes a couple of microwatt or several gigawatt, it is the **relationship** with the power management consumption that will tell us whether the neural network saves energy. We introduce a new variable ϕ to represent this relationship. We choose to define ϕ as E_{nn}/E_{sense}. That is, ϕ represents how much energy the invocation of a neural network consumes compared to the static sense cycle. If this ratio is 1, the sense cycle and the neural network thus consume equal amounts of energy whenever they are run. If ρ is also 1, this would mean the neural network *doubles* the total energy consumption. The power management system would thus need to enable the IoT device to gather the same amount of data in **half** as many scans in order to break even. A lower ϕ would mean a smaller increase in overall power consumption, meaning the system would have to save less energy to break even.

We look at some extreme values to get a sense of what ϕ means. If ϕ is close to 0, that would mean the power management system consumes a trivial amount of energy compared to what the device is already doing. This would be a good sign, and would likely let us invoke the neural network every single iteration of the loop for optimal power management. On the other hand, if ϕ is large, the power management system consumes a lot of power compared to the sense cycle. This would mean that the configuration would have to be updated quite infrequently, meaning a lot of the benefit of using an *intelligent* power management would be lost. The value of ϕ can thus be used as a quick indicator of whether using a neural network-based power management system is appropriate – the lower, the better.

The definition of the variables introduced so far are summarized in table 5.1.

E_{sense}	Power Consumption of a Sense Cycle in a single loop
E_{nn}	Power Consumption of the Invocation of a NN
E_{tot}	Total Energy Consumption per Loop
ρ	How often the Neural Network is Asked
ϕ	E_{nn} / E_{sense}

Table 5.1: The definitions of our various parameters.

We now wish to use these definitions to discuss the net amount of power the neural network can save. Moving the terms of the definition of ϕ, we get

$$\phi = \frac{E_{nn}}{E_{sense}}$$
$$E_{sense} = \frac{1}{\phi}E_{nn}$$

We can use this result to find another expression for E_{tot}:

$$\begin{aligned} E_{tot} &= E_{sense} + \rho E_{nn} \\ &= \frac{1}{\phi}E_{nn} + \rho E_{nn} \\ &= (\frac{1}{\phi} + \rho)E_{nn} \end{aligned} \tag{5.1}$$

To find how much energy the system saves, we must first express the portion of total energy that is **consumed** by the invocation of the neural network. We introduce the term ψ to represent this idea – that is, ψ is the percentage of all energy input that goes to the neural network. The formulaic expansion of ψ becomes the ratio of a **neural network's** energy consumption per loop, ρE_{nn}, and the **total** energy consumption per loop. Using equation 5.1 for this latter term, this yields:

$$\psi = \frac{\rho E_{nn}}{E_{tot}} = \frac{\rho E_{nn}}{(\frac{1}{\phi} + \rho)E_{nn}} = \frac{\rho}{\frac{1}{\phi} + \rho} \tag{5.2}$$

ψ tells us **how much energy the neural network needs to save in order to break even**. To see why, remember that we defined ψ as the percentage of total energy consumption that the neural network is responsible for. If we save *more* than this percentage, the system will have saved energy overall. This result is illustrated in figure 5.3, with ρ from 0 to 1 and ϕ from 0 to 2. Notice that when both invocation frequency ρ and energy consumption ratio ϕ is low, the amount of energy the neural network needs to save in order to break even approaches zero. The neural network has to save trivial amounts of energy to be worth including. On the other hand, at the maximum values of $\rho = 1$ and $\phi = 2$, we are invoking the neural network every time and the invocation costs *twice* as much energy as the sense cycle. The figure shows that the power management then needs to reduce overall energy consumption by as much as **70 %** to break even. To analyze some more concrete examples, we plot their exact values in table 5.2.

There are no entries in table 5.2 where either ϕ or ρ is 0. This comes from the realization

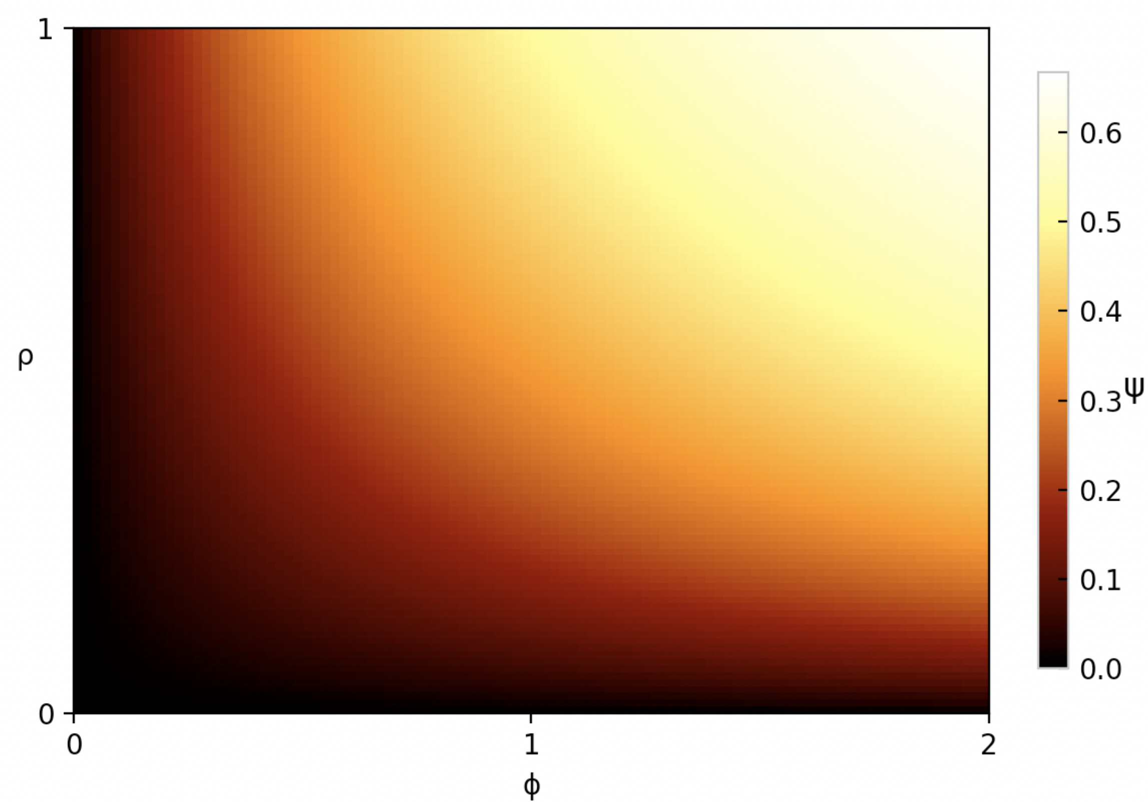

Figure 5.3: ψ, the percentage of energy that a neural network has to save in order to break even with its consumption. ϕ is the proportion of energy input going to the neural network, and ρ is the frequency of invocation. Calculated using formula 5.2.

that $\rho = 0$ means a situation where we *never* invoke our neural network, meaning the rest of the discussion makes no sense. In the case of ϕ, equation 5.2 has a term where ϕ is the denominator, meaning the equation is not defined for $\phi = 0$. Thinking about the real world scenario this would represent, we realize that this value would imply that the sense cycle consumes infinitely more energy than the power management system.

ρ	ϕ	ψ
	0.01	0.0
0.01	0.5	0.005
	1	0.01
	0.01	0.005
0.5	0.5	0.2
	1	0.33
	0.01	0.01
1	0.5	0.33
	1	0.5

Table 5.2: The value of ψ given different invocation ratios ρ and energy consumption ratios ϕ.

This is another scenario where discussion is nonsensical. We thus conclude that both ρ and ϕ must be greater than 0.

The value combinations in table 5.2 can give us an understanding of what configurations our system makes sense for. First, consider the final entry in the table: $\phi = 1$, $\rho = 1$, and $\psi = 0.5$. These values correspond to the example we described earlier: if you ask the neural network every time, it ends up doubling energy consumption. You then need to **halve** the total energy consumption in order to break even – $\psi = 0.5$. As another example, consider $\rho = 0.5$, keeping $\phi = 1$. This would mean that the neural network still doubles energy consumption *on the loops it is used on*, but you only ask every other loop. After two loops, then, the device has consumed E_{sense} in the first loop and $E_{sense} + E_{nn}$ in the second, meaning the neural network consumes a third of all power. We would expect it to need a total power reduction of 33 % in order to break even. Consulting the table, we see that given $\rho = 0.5$ and $\phi = 1$, we indeed get $\psi = 0.33$. This is a good indication that our formula reflects reality. As long as it does, we have successfully developed a method of determining which parameters lead to the neural network breaking even energy-wise.

5.3 Case Study

The final sub-criterion we defined, and thus the final question we need to answer to have addressed our research question in full, read as follows:

Verify whether the externally provided neural network saves more energy than it consumes.

This criterion calls for a closer examination of one particular neural network. We wish to perform a case study of this network for two reasons: First, because doing so will provide insight into the methods developed so far by putting them into practice. And second, to provide insight into the externally provided neural network to the larger project our work is a part of.

5.3.1 Evaluation of Externally Provided Neural Network

We wish to evaluate our externally provided neural network (ENN), and we do so by applying formula 5.2. The formula produces a concrete value for how much energy the neural network needs to save in order to break even, providing insight into which parameters are required for the neural network approach to make sense. We recall that the ENN, shown to produce power management behavior in [Murad et al., 2019a], had depth = 3 and width = 128. We thus begin establishing the various input required by formula 5.2.

With the goal of using formula 5.2 to evaluate the ENN, we first want to determine a value for ϕ. To do this, we need to determine E_{nn} and E_{sense}. We begin with E_{nn}, the energy consumption of the invocation of the neural network per loop. To find this value, we first need the power consumption of the neural network invocation. Luckily, the neural network uses no sensors or other external hardware, but simply runs on the CPU. This means that finding the energy consumption rate of the CPU also gives us the rate of neural network invocation. We find this value by looking at the datasheet of the Arduino Nano 33 BLE's microcontroller, and we see that it has a reported power throughput of 52 μA when using its CPU [Ard, 2020b]. Given the Arduino's operating voltage of 3.3V, this means a power consumption rate of **0.17 mW** for any process running on its CPU. We find E_{nn} by multiplying this energy consumption rate with the runtime of invocation. The runtime can be found using table 4.14 or figure 4.15, and is approximately 9.3 ms for our ENN. That means E_{nn} for this network is 0.17 mW * 9.3 ms = **1.581 μWs.**

The next step in the evaluation of our ENN is determining E_{sense}. This requires first determining the energy consumption *rate* of a sense cycle. We do this in two ways: first, assuming the sense cycle is simply regular code. This means it runs directly on the CPU, and the energy consumption rate is the same as the CPU's: 0.17 mW. The second way is to also include some sensor scan or network transmission, producing a consumption rate that likely reflects real use cases more closely. Finding the energy consumption rate of our Arduino microcontroller's BLE sensor can serve as a good starting point for this second approach. We observe directly from the Arduino's datasheet that when the sensor is active, it has a power consumption of **15.5 mW**. This is more than 90 times larger than the standard CPU rate.

We now combine the established sense cycle consumption rates with their observed runtimes to produce the sense cycle's energy consumption per loop. We observed in section 4.3 that a representative sense cycle took approximately 5 ms. Assuming the use of our experimentally developed sense cycle without any sensors, we know that we have a power consumption rate of 0.17 mW. This yields an E_{sense_1} of 5 ms * 0.17 mW = **0.85 μWs**. If we also wished to include a sensor scan, we observed in section 4.1 that the Arduino Nano 33 BLE's Bluetooth sensor scan consistently took 1 ms. Given the sensor's established power consumption of 15.5 mW, we get an additional 77.5 μWs. This would mean E_{sense_2} becoming **78.35 μWs**.

To make sure our conclusions for the sense cycle are not too hardware-specific, we also look at the power consumption rate of other commonly used sensors. [Ferry et al., 2011] is a work in which the power consumption of various hardware, including sensors, was experimentally observed. Figure 5.4 shows their result. We see that all but the lowest couple of nodes have power consumption rates significantly higher than our nRF CPU. Although not an extensive study, we take this as an indication that IoT sensors typically consume energy at a rate significantly higher than that of regular CPU operations. A deeper look at commonly used IoT sensors' power consumption could be an interesting point of further study.

Components	Type	Min (mW)	Max (mW)	Avg. (mW)	%
Concertina	Sensor 1	355.66	393.38	368.84	51.54
Miwi 1	Radio 1	115.5	181.5	115.92	16.2
OLTC 50	Gas sensor 1	0	1074.6	72.458	10.13
OLTC 80	Gas sensor 2	0	884.51	58.216	8.135
MT48T35AV	RAM	0	150	26.326	3.679
LM3100	DCDC	25.84	55.03	25.928	3.623
Mygale	Sensor 2	0	583.49	19.268	2.693
MAX618	DCDC	0	93.446	10.822	1.512
PIC24F	CPU	0	52.8	9.2664	1.295
LM3100	DCDC	0	84.023	6.0276	0.8423
LM3100	DCDC	0	12.627	2.2161	0.3097
µCAM	Video	0	305	0.21	0.029
Miwi 2	Radio 2	0	181.5	0.0834	0.0116
Total Sensors				519	72.58
Total Radios				116	16.23
Total DCDC				45	6.3
Total RAM				26	3.63
Total CPU				9	1.26

Figure 5.4: Power Consumption of various nodes. Taken from [Ferry et al., 2011].

Having established both E_{sense} and E_{nn}, we can now calculate ϕ. If we assume the sense cycle to be simple code running on the CPU, we get $\phi_1 = \frac{1.581}{0.85} = \mathbf{1.86}$. If we include the BLE scan, we get $\phi_2 = \frac{78.35}{0.85} = \mathbf{0.02}$. Recall that we stated that large values for ϕ indicated a poor fit, while small ones indicated the scenario was appropriate for neural network power management. We make the observation that when the sense cycle includes a BLE scan, the neural network appears to be a significantly better fit.

The only variable left to determine at this point is the rate at which the neural network should be invoked, ρ. This depends on the application. Some sensing applications may need to adjust their duty cycle, for instance, once every hour, while others may only need to do that once a day. Depending on the frequency of the sensing cycle, ρ can hence take a large range of values. For now, we look at what ψ evaluates to for a couple of different arbitrary values:

ϕ	ρ	ψ
	0.01	0.0
0.02	0.1	0.002
	1	0.02
	0.01	0.018
1.86	0.1	0.157
	1	0.65

Table 5.3: The amount of energy a neural network power management system needs to save in order to break even ψ, given observed ϕ and selected ρ.

In table 5.3, we have calculated the value for ψ given the two different ϕ values observed earlier and a couple of selected ρ. We interpret the result as follows. Taking the final entry as an example, with $\phi = 1.86$ and $\rho = 1$, we are here assuming that the neural network is asked for updated configuration every iteration of the loop and that the sense cycle does *not* use any sensors. Given this, the ENN would have to reduce power consumption by **65 %** to break even. If we observe a reduction in scan frequency (duty cycle) of 65 % without the quality of the data dropping, the system will have gone energy neutral. This is a quite large percentage. If we instead assume that the sense cycle *does* use a BLE scan, meaning ϕ is 0.02, ψ drops to a mere **0.02**. The system would then only have to save 2 % energy to break even. This reflects the nature of ϕ as an indicator of fitness.

The values for ρ in table 5.3 were arbitrarily selected, but its value has a significant impact on the result. Often, realistic values for ρ can be deduced using two factors. First, the neural network's inputs. For our ENN, for instance, weather forecast is the major component the neural network uses to update its policy. As these are only updated three or four times a day, invoking the neural network more often than this yields no benefit. Second, consider the nature of the IoT device's purpose. This can often indicate what actual scan frequencies are realistic. If an IoT node is monitoring the temperature of its environment, for example, we know that the time between each scan should probably be on the scale of minutes or hours. Thus, on one end of the spectrum, we might want temperature updates roughly every five minutes but a policy update only once a day. This means we invoke once per 300 regular loops, or a ρ of $\frac{1}{300} = 0.0033$. On the other end of the spectrum, we might want policy updates four times a day, and only perform temperature scans every other hour. This yields a ρ of 0.33. Testing a series of values for ρ in the range $[0.0033, 0.33]$ might then be appropriate. Using this sort of deduction lets us likely values to test for ρ.

We are interested in how much energy the ENN needs to save in order to break even, ψ, given these various ρ. We do so by plotting ψ given the two observed ϕ and *all* possible ρ. Figure 5.5 shows the result. By analyzing the figure, we can quickly get an idea of whether the ENN is appropriate for IoT power management for a given set of parameters. If ϕ is as low as 0.02, for instance, we immediately observe that the ENN would almost certainly be able to break even. If ϕ is instead closer to the observed 1.86, we would need to determine one remaining parameter in order to make any conclusion. If the neural network was determined to need a ρ of at least 0.5 to be effective, for example, the figure would tell us that the ENN would need to reduce power consumption by around 45 % in order to break even. Figure 5.5 thus serves as a conclusion of our case study.

5.4 Research Question Revisited

Revisiting our research question, we recall that it was phrased as such:

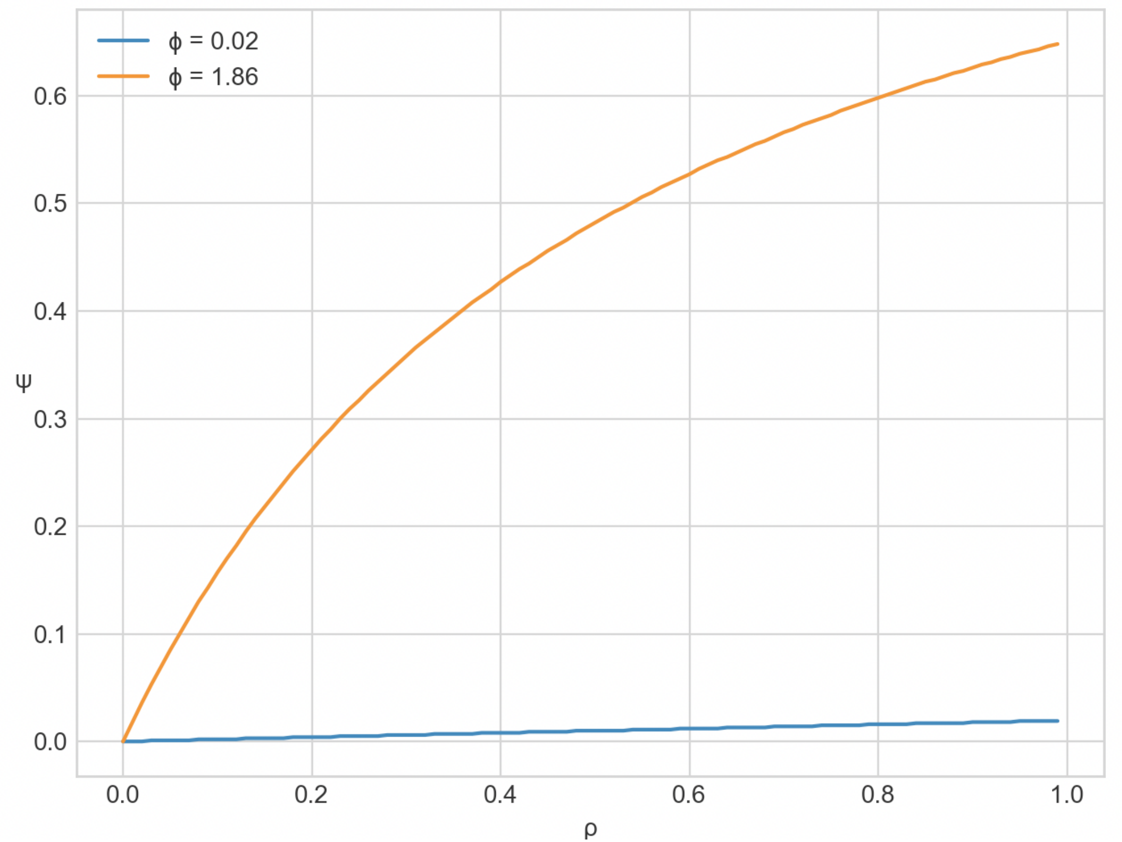

Figure 5.5: The percentage of energy the neural network needs to help save in order to break even with its consumption, ψ. Plotted for all ρ and the two observed values of ϕ, 0.02 and 1.86.

Are we able to utilize neural networks on today's IoT devices in such a way that they help save more energy than they consume?

With the data provided in chapter 4 and the discussion in this chapter, we now have a better grasp on whether each part of this question is fulfilled. The first part is whether we are able to utilize a neural network on IoT devices at all. Going all the way back to the introduction of the project, we posed this problem as a Secondary Research Question: **Do neural networks representing power management policies fit on the restricted hardware of IoT microcontrollers?** With the data provided in sections 4.1 and 4.2, along with their thorough discussion in section 5.1, we can now make a conclusion: yes, they do. There are limits to the size of the neural networks for them to fit, and we found these limits for our experimental set-up in section 4.2, but neural networks of realistic power management sizes did indeed fit.

The second part of the Research Question regards how the neural network *performs*. Again remembering back to chapter 1, we posed this problem as Secondary Research Question 2: **In what circumstances is the neural network-based power management able to help save more energy than it consumes?**. We spent a significant part of our discussion developing a method for answering this question. The

result is summarized as the following procedure.

1. Train or choose a neural network for IoT power management.

2. Compile and transfer the neural network to a microcontroller using the steps described in section 4.2.1.

3. Measure the runtime of the application both with and without invocation of the neural network.

4. Find the energy consumption rates of the CPU and of any sensors being used.
5. Combine the runtime and consumption rates to calculate (E_{sense}) and (E_{nn}).

6. Calculate $\phi = E_{nn}/E_{sense}$.

7. Choose some invocation frequency ρ.

8. Calculate $\psi = \frac{\rho}{\frac{1}{\phi}+\rho}$.

The result of the presented procedure is ψ, the percentage of energy the neural network needs to save to break even. If the neural network is able to help reduce energy consumption by this much, it will have broken even energy-wise. Combined with the data on when neural networks fit in a microcontroller's memory, this results in an answer to our research question.

Chapter 6
Concluding Remarks

The motivation for this work was to help verify the applicability of the neural network approach to IoT power management. We have taken extensive steps to test whether this approach is appropriate. First, we verified that the neural networks fit and are runnable on representative IoT hardware. As a part of this process, we found the boundaries a neural network has to respect if it is to fit on an IoT microcontroller. We then looked at the *energy consumption* of these neural networks. We established that for networks within our chosen size limits, the energy consumption is on an order of magnitude that reassures us that any power it saves it not outweighed by its consumption. Together, these observations led to a concrete range of parameters for which a neural network is able to help save power. We then made the observation that for a particular neural network designed for IoT power management, all parameters were well within the established bounds. Within these parameters, we can thus make the conclusion that indeed, neural networks are appropriate for use in IoT power management.

Besides the theoretical results that should generalize over a wide range of cases, our results are also practically achievable. We present the developed step-by-step procedures as guides for users to adapt our methods and results into their own work. Using the procedure outlined in section 4.2.1, they have a detailed description of how to go from a trained Tensorflow model to code runnable on a resource-restricted microcontroller. Equation 4.2 can be used to calculate whether the network will fit in the device's memory, given the neural network's size. Equation 4.5 can be used to calculate the CPU runtime of invoking the neural network. Finally, the procedure presented at the end of chapter 5 can be used to obtain a concrete value for how much energy the neural network saves. Combined, these methods provide insight into the details of power management using neural networks of any size, at any invocation frequency or efficiency, within the given limits. In this way, this work not only helped the advancement and validation of the larger project we are a part of, but it also provides general principles and guidelines for analysis of neural networks in the IoT power management field.

Throughout the project, we observed several tasks that might be interesting to explore further as future work. One is to test our externally provided neural network with real input over an extended period of time. This would let us determine some

of the unknown variables of the project, facilitating a deeper analysis of the neural network's performance. Another parameter we might wish to expand upon is the size limits we set for our neural networks. A depth of 5 is already considered rather deep, but widths exceeding 1024 are not too uncommon [Ope, 2018c]. Other works in the field have suggested that the RAM of a microcontroller starts becoming an issue at a width of around 40 000. To study the ramifications of RAM restrictions, either expanding our boundaries in this direction or reducing the available hardware RAM might be interesting. A final topic that might deserve further exploration is what we called the *sense cycle* – the static application running on an IoT device. A deeper look at what such applications typically look like among deployed IoT devices would provide greater insight into the relationship between the neural network and its environment.

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*.

[Sta, 2020] (2020). Assumption of normality. *https://www.statisticshowto.com/assumption-of-normality-test/*.

[Res, 2020] (2020). Feed-forward neural network overview. *https://www.researchgate.net/figure/Feedforward-neural-network*$_f$*ig*1$_3$29586439.

[Ard, 2020c] (2020c). Millis. *https://www.arduino.cc/reference/en/language/functions/time/millis/*.

[NG:, 2020] (2020). Neural networks and deep learning. *https://www.coursera.org/learn/neural-networks-deep-learning.*

[NRF, 2020] (2020). Nrf connect. *https://www.nordicsemi.com/Software-and-tools/Development-Tools/nRF-Connect-for-mobile.*

[Agr, 2020] (2020). Polynomial regression. *https://agrimetsoft.com/regressions/PolynomialtoolSection.*

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

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

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

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

[Ferry et al., 2011] Ferry, N., Ducloyer, S., Julien, N., and Jutel, D. (2011). Energy estimator for weather forecasts dynamic power management of wireless sensor networks. *J.L. Ayala et al. (Eds.): PATMOS 2011, LNCS 6951, pp. 122–132, 2011.*

[Haukland, 2019] Haukland, J. (2019). Modelling the energy consumption ofnb-iot transmissions. *Master's thesis, NTNU.*

[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.

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

[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.

Finn Julius Stephansen-Smith

Using Neural Networks for IoT Power Management