NTNU
Innovation and Creativity

Using neural networks for IoT power management

Finn Juliuus Stephansen-Smith

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

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

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

Problem description:

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

Responsible professor:	Frank Alexander Kraemer, IIK
Supervisor:	Frank Alexander Kraemer, IIK

Abstract

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

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

Sammendrag

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

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

Contents

List of Tables **VI**

List of Figures **VIII**

List of Equations **IX**

1 Introduction **1**
1.1 Background and motivation . . . 1
1.2 Problem scope . . . 4
1.3 Methodology . . . 5
1.4 Outline . . . 7

2 Background **9**
2.1 Reinforcement learning . . . 9
2.1.1 Key concepts and terminology . . . 10
2.1.2 Q-learning . . . 11
2.1.3 Reinforcement learning algorithms . . . 12
2.2 Feed-forward neural networks . . . 14
2.2.1 Neurons and layers . . . 14
2.2.2 FFNN in reinforcement learning . . . 16
2.3 Power management in IoT . . . 17
2.3.1 Static algorithms . . . 18
2.3.2 Reinforcement learning in IoT . . . 19
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 . 26

3 Methodology **28**
3.1 Research question and context . . . 28
3.2 Framework for experimentation . . . 28
3.2.1 Sense cycle . . . 28
3.2.2 Neural network on a microcontroller . . . 28

4 Experiments **31**
4.1 Memory consumption 31
4.1.1 Static sensing cycle 31
4.1.2 Neural network invocation 31
4.1.3 Total memeory 32
4.2 Runtime memory 35

5 Results **37**
5.1 Some section 37
5.1.1 Some subsection 37

6 Concluding remarks **38**

List of Tables

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 the devices used in [Berg, 2019] and those considered in our project. Taken from [Berg, 2019] pp.38 and [Semiconductor, 2019]. 29

4.1 Memory consumption of the various parts of our experimental program. 34

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

2.1 The basic structure of Reinforcement Learning. Taken from [?]. 10
2.2 Illustration of a feed-forward neural network, in which connections never go backwards. Taken from [Res, 2020]. . 15
2.3 A taxonomy of some of the most popular algorithms used in modern RL. Taken from [Ope, 2018d] 17
2.4 One year of solar power availability at a particular geographical location. Taken from [Buchli, 2014]. 18
2.5 Comparison of root mean square deviation of energy neutrality of each month of spring for three comparing methods. Taken from [Hsu et al., 2015]. 20
2.6 An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019]. 24

2.7 The intended agent/environment setup of [Murad et al., 2019]. 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., 2019]. 26
2.8 Graphs showing simulated solar power and corresponding duty cycle chosen by an agent trained using neural networks. Taken from [Murad et al., 2019], ... 27

3.1 The iterative nature of design science. Taken from [Des, 2019]. 29
3.2 The iterative process we will follow for the design and validation of the neural network. 30
3.3 Arduino Nano 33 BLE, the physical IoT device we plan to use. 30

4.1 The memory consumption of our static program. 32
4.2 The memory consumption of the neural network. 33
4.3 Total memory consumption at runtime. 35
4.4 The distribution of program memory at compile time. . . 36
4.5 The distribution of dynamic memory at compile time. . . 36

List of Equations

2.1 Q function 11
2.2 Getting optimal action from a Q-function 13
2.3 Energy neutrality 20
2.4 Berg's runtime memory estimation 23
2.5 Total energy consumption 24
2.6 Simple energy consumption 25
4.2 Union of sets 33

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 **??** then briefly explains how neural networks (NNs) can be applied to help achieve efficient power management in the IoT domain. Section 1.3 presents the concrete steps we propose to take in order to advance scientific knowledge about the subject. Finally, Section 1.4 gives a brief summary of the chapters constituting the rest of the report.

1.1 Background and motivation

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

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

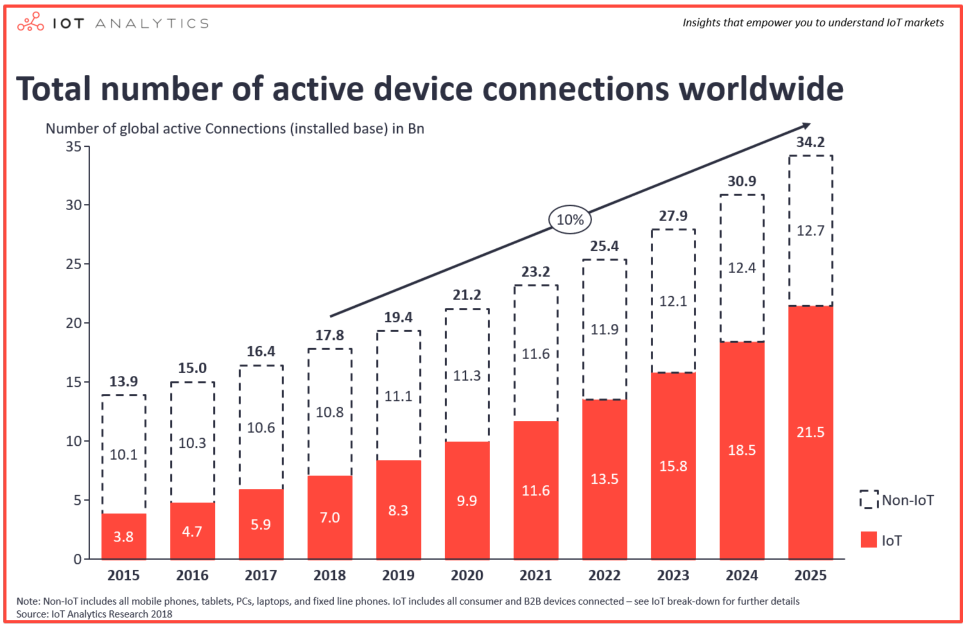

Figure 1.1: The number of devices connected to the internet. Taken from [Lasse Lueth, 2018].

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

In the traditional era of computers, power management was a non-factor. Being connected to a power grid meant a practically infinite supply of energy. With the transition into laptops and smart phones, this changed: optimizing both hardware and software so as to maximize battery life became essential. IoT takes this one step further. While there are many IoT devices connected directly to power, there are also many that are not. Some examples include temperature sensors, wildlife monitoring, and even urban applications such as traffic sensors or parking weights. Unlike laptops or smart phones, it is not practical to plug these devices in and charge them in regular intervals. Instead, one of two main alternatives must be chosen. One is for manufacturers to simply supply the device with a large enough battery that it does not run out of power for its expected life time. This life time can typically be on the scale of a couple of years at most, at which point many devices need maintenance or replacement anyway.

Another more sustainable approach exists, however. 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 1.2.

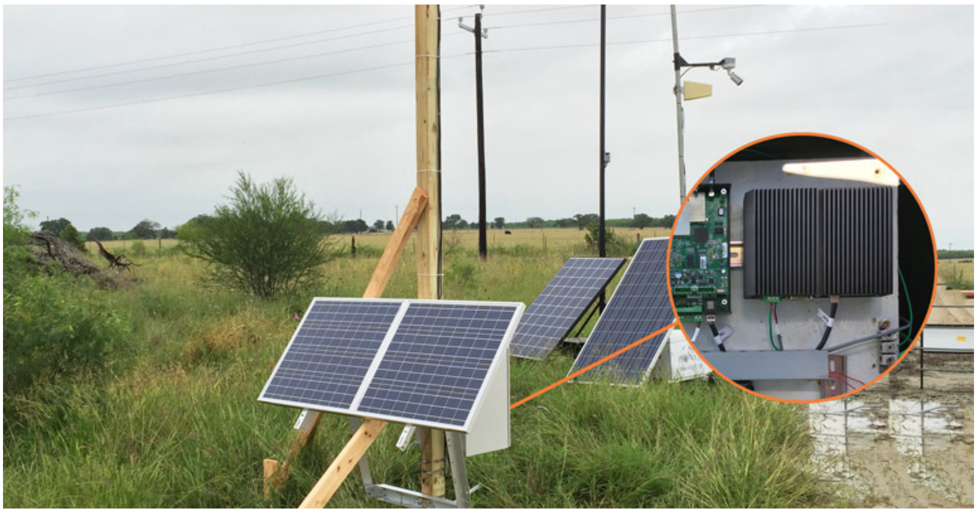
Figure 1.2: An example of solar panels being used to provide sustainable energy for a deployed IoT device. Taken from [OnL, 2017].

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

The interesting case is the one where the balance between energy coming in through the solar panels and that being spent will be relatively close. Without this being the case, the IoT device consumes too much or too little power for any software or hardware decisions we make to really matter. When there is such a balance present, however, things change. If we let the device perform its function at 100% capacity at all times, it would consume more battery than provided with and die. Such power failures are undesirable. We also shouldn't turn the throughput down too much either, though; we want as much output from the device as possible given the available power.

This is where neural networks come in. Using machine learning, we aim to enable an intelligent utilization of available power. Our example uses solar power and weather data, but the approach is generally applicable. By combining data about weather patterns, the state of the device's battery, as well as inputs such as weather forecasts, we train a neural network to output an optimized level of operation for the device. Thus, we are able to achieve efficient power management in the Internet of Things.

1.2 Problem scope

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

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

We pose the following Research Question (RQ): **Are we able to *implement* neural networks on *today's* IoT devices in such a way that they utilize energy better than if they had used previous**

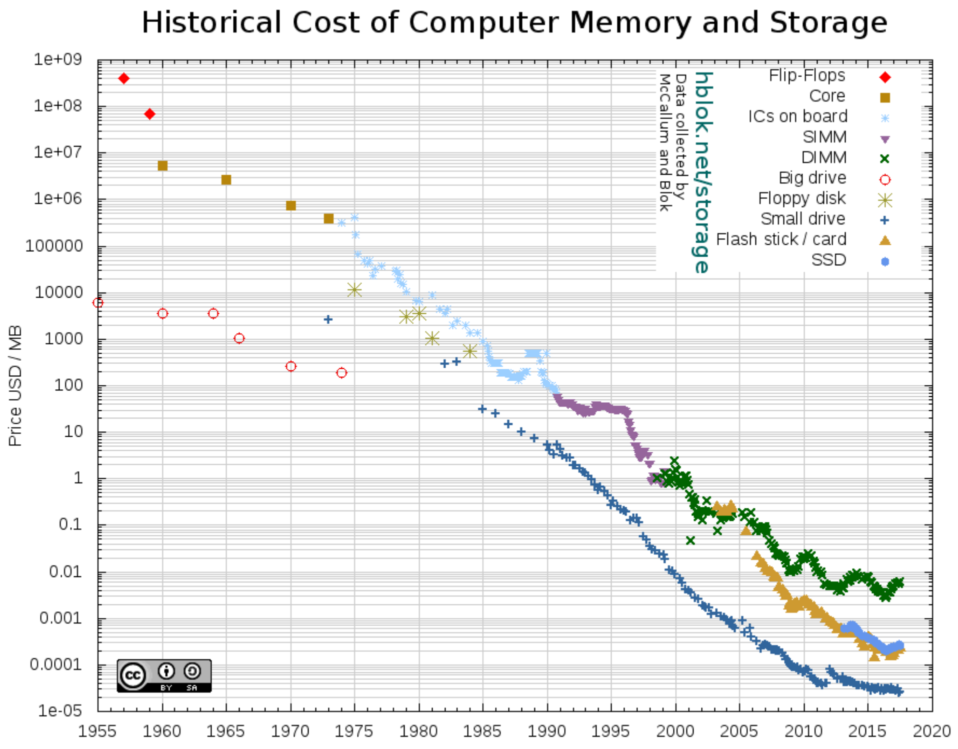

Figure 1.3: The cost of computer memory over time. Taken from [hbl, 2017].

approaches?

By *implement*, we mean transferring a neural network model to a device, then successfully utilizing 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.

1.3 Methodology

We investigate the research question in several steps. First, a suitable IoT device must be chosen for the experiment. If the hardware we test on is strongly divergent from the industry standard, the likelihood of our work being left behind by future development increases. In addition, it's important that our work does not become too hardware-dependant. The chosen device ended up being the **Arduino Nano 33 BLE** [Ard, 2020b], as it matches all of these criteria as well as being well documented, widely used, and contains the highly IoT-relevant BLE hardware.

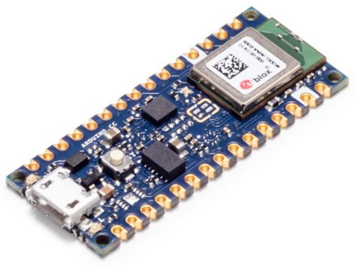
Figure 1.4: Arduino Nano 33 BLE, the physical IoT device we plan to use.

Our chosen Arduino device only has 1MB of storage and 256KB of RAM [Ard, 2020b]. This means that fitting any amount of code on it is a challenge. As such, we will spend time on analyzing the memory footprint of the different parts of a program required to realize power management. This information can be used to improve the neural network, reducing its size if necessary, and thus helping fit these models for real world applications. In our case, this memory is affected by the nature of Arduino devices and the Arduino IDE. We look at which aspects are specific to our chosen hardware, and we extrapolate this into a general statement about the feasibility of using neural networks on resource-constrained devices.

The research is design-based, meaning it involves something designed by humans. Phrasing the process of making something so generally is useful because it clearly differentiates the scope of our work from that of the nature sciences. In those, the goal is typically to examine something out of our control with the goal of determining some truth about how it works, *without changing it.* In contrast, Design Science provides the means for us to academically look at scientific processes that aim to change the world [Des, 2019]. This often means solving some problem or providing some new value for stakeholders. As we aim to provide IoT device owners with the new possibilities granted by intelligent power management, we very much fit into this category. As such, we rely on the principles outlined by Design Science to guide our work.

With a representative hardware platform chosen and a scientific framework for the experimentation in place, we have the foundation we need for our work. Our experiments will be based a typical application for a sensing IoT device. After making this, we will attempt to integrate it with a neural network. By invoking from a pre-trained model, IoT devices will be able to update their power management parameters in an intelligent manner. Specifically in our work, this could mean the device asking the neural network how often often it should perform its sensing cycle. The network will be trained to answer this in such a manner that the device scans as often as possible while not running out of battery. Our experiment will consist of iterating our sensing cycle code, analyzing the memory available for a neural network on the device, using this to adjust the neural network, and repeat. This will enable us to move closer to an answer to our research question.

1.4 Outline

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

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

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

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

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

Chapter 2
Background

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

2.1 Reinforcement learning

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

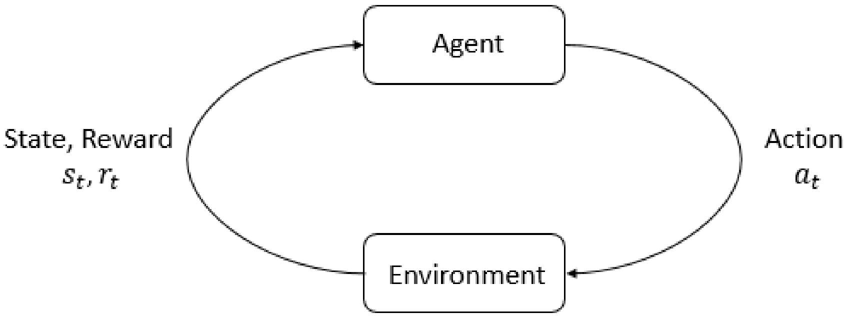

Figure 2.1: The basic structure of Reinforcement Learning. Taken from [?].

2.1.1 Key concepts and terminology

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

In order to *affect* the world around it, the agent can perform *actions*. We denote this by saying that it performs an action a on state S. In response to an action, the agent typically receives an indication of how good the new world ends up being. By providing appropriate strategies for picking actions, as well as how the *goodness* of the world is calculated, we enable the agent to train. An action leading to a better world state means the agent becomes more likely to perform that action in the future. The result is an agent able to perform well in its given world — exactly the result we're aiming for. Hopefully, this behaviour that works well in our simulated world **also** works well when the agent is put to test in the real world. Only if it is have we successfully used reinforcement learning to achieve a desirable real-world effect. It is thus we see the importance of what defines *goodness* in the simulated training world: it needs to match what's good in the real world. The element calculating this goodness is called the *reward function*, and choosing or designing a suitable reward functions is both immensely challenging and fundamentally essential in reinforcement learning [Ope, 2018c].

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

2.1.2 Q-learning

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

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

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

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

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

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

2.1.3 Reinforcement learning algorithms

A strategy for how to apply the various Reinforcement Learning aspects and how to specify variables to achieve actual learning is called an *algorithm*. An algorithm is no more than a series of steps to take in order to achieve a goal. In the context of reinforcement learning, the term also sometimes encapsulates specifications of the parameters mentioned earlier, such as the reward function.

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

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

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

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

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

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

2.2 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 2.2. Section 2.2.1 introduces the necessary details of FFNNs, while section 2.2.2 describes how this can be used in conjugation with Reinforcement Learning.

2.2.1 Neurons and layers

As can be seen in figure 2.2, 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 2.2. Correspondingly, the largest

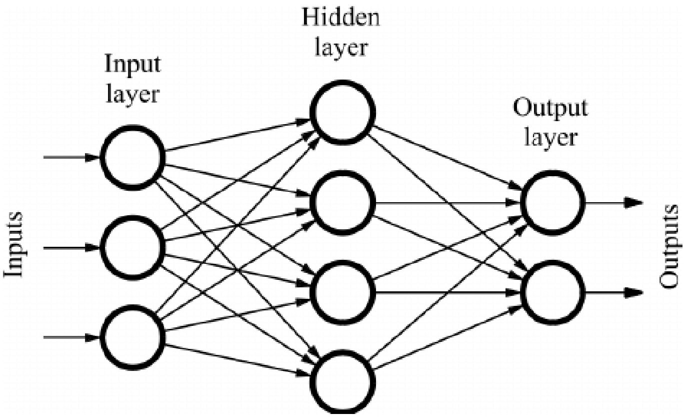

Figure 2.2: Illustration of a feed-forward neural network, in which connections never go backwards. Taken from [Res, 2020].

amount of neurons in a single layer denotes the *width* of the network, in our case 4.

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

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

2.2.2 FFNN in reinforcement learning

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

There are advantages and disadvantages to this approach. Training neural networks can be a heavier process computation-wise than other approaches, to name one. There is also a larger dependency on knowledge on the side of the developer; the math behind neural networks and the skill required to design an appropriate reward function is far from trivial. However, there are advantages when compared to traditional approaches as well. By defining the output of training as a set of weights for a neural network, we are effectively able to handle a *continuous* spectre of input. This can be a crucial advantage over discrete outputs, providing increased accuracy and enabling a whole new field of real-world scenarios. This continuous nature also allows a whole range of mathematical tricks and optimizations to improve the training process. These are encapsulated in the practical specifications of reinforcement learning: algorithms.

A wide variety of algorithms in reinforcement learning have been proposed, and more are being developed every year. Figure 2.3 provides an overview of some of the most common ones used today. We'll look closer the TD3 algorithm, introduced in section 2.1.3. TD3 has several policy strategies available. One, based on convolutional networks ("CnnPolicy"), is mainly used for image processing and recognition. The other major option is caled Multilayer Perceptron policy, or MlpPolicy. Multilayer Perceptron is a class of feed-forward neural networks. It simply means that there at least a single hidden layer. If there are more than a single hidden layer, we say we're dealing with *deep* learning.

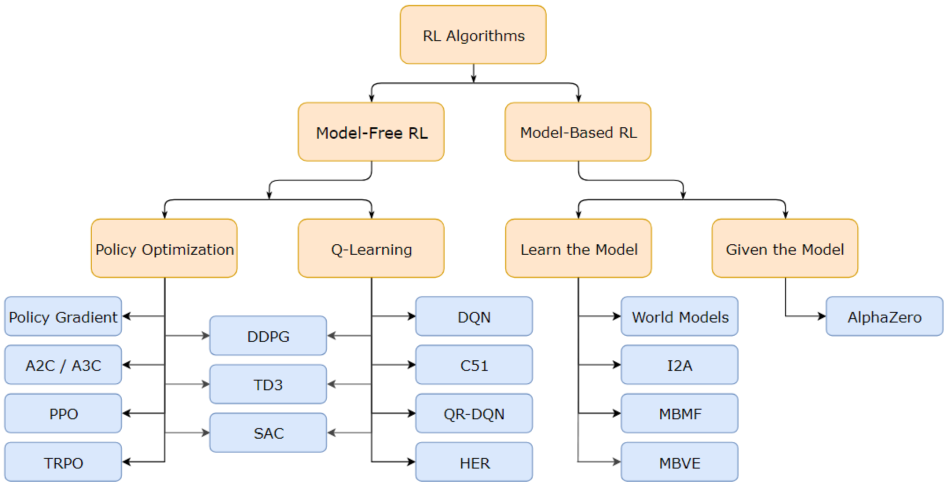

Figure 2.3: A taxonomy of some of the most popular algorithms used in modern RL. Taken from [Ope, 2018d]

It is through the choice of this particular approach that our project becomes focused on neural networks. It is here we diverge from existing works such as [Hsu et al., 2009b], who have attempted achieving our particular goal with reinforcement learning, but without neural networks.

2.3 Power management in IoT

Although we focus on neural networks, there has been a large variety approaches suggested to achieve efficient power management in the IoT. [TODO rewrite to avoid plagiarism from semester thesis] The naive solution, used in a multitude of devices deployed today, is to always "go". If the purpose of a device is to read temperature, for example, this *always-go* policy would involve performing a temperature scan every x seconds. There would be no consideration of current battery level, estimated cost of scan, the likelihood the reading being interesting, or other factors that might influence whether performing a scan is actually a good idea. To improve upon this, several strategies have been suggested.

2.3.1 Static algorithms

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

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

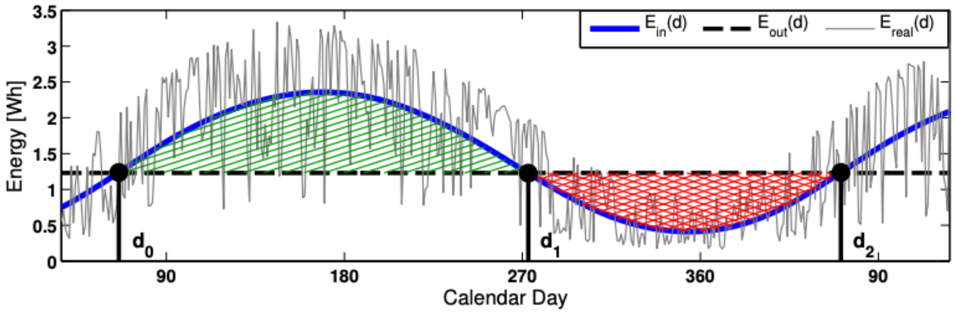

Figure 2.4: One year of solar power availability at a particular geographical location. Taken from [Buchli, 2014].

[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 2.4. The blue line $E_{in}(d)$ is their expected solar input, extrapolated from historical data. Their algorithm used this to produce the dotted line $E_{out}(d)$, meaning their IoT device ran at a rate corresponding to a constant consumption of around 1.3 W/h. This is designed by the algorithm to build a buffer (green) during summer months that brings the device safely through the power deficit (red) of winter.

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

2.3.2 Reinforcement learning in IoT

Going a step further than static algorithms, there are existing works exploring the approach of utilizing reinforcement learning in IoT power management. These have largely focused on Q-learning algorithms. To the best of our knowledge, none have used a neural network-based policy. One consequence of this is that the policy they produce is discrete, while neural networks can achieve a continuous representation.

Focusing on the Q-learning approach, Hsu et.al have published a series of work on the topic since 2009 [Hsu et al., 2009b]. Their work is based on the introduction of the term *energy neutrality*, defined as follows:

$$E_{distance\_from\_neutral} = E_{harvest} - E_{consumed} \tag{2.3}$$

That is, the difference from energy neutrality is 0 when the device consumes exactly as much energy as it receives. We say that it is *energy neutral*. Achieving this means ideal power management. In reality you might want some buffer to ensure the battery doesn't die, but Hsu et.al. among others work with the slightly idealized situation that a perfectly energy neutral device is the perfect, unobtainable goal of power management. With this assumption, they are able to use the definition of energy neutrality to derive mathematical formulas. By attempting to minimize equation 2.3, they can pose the power management challenge as an optimization problem. Specifically, they formulate the reward function of their reinforcement learning algorithm so that a lower energy neutrality leads to a higher reward. With this pretext, they train an agent using basic reinforcement algorithms, iteratively improving their approach in various ways to try to further reduce distance from energy neutrality [Hsu et al., 2009a] [Hsu et al., 2014] [Hsu et al., 2015].

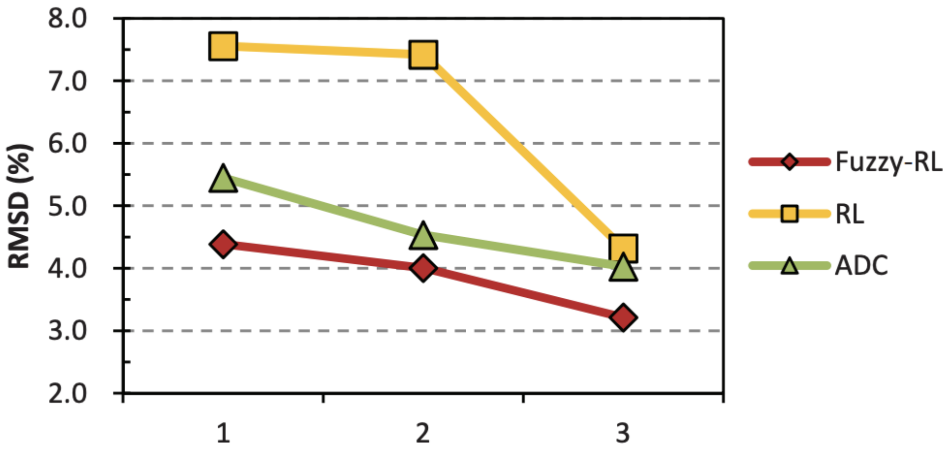

Figure 2.5: Comparison of root mean square deviation of energy neutrality of each month of spring for three comparing methods. Taken from [Hsu et al., 2015].

With each iteration, they show that their results improve compared to previous approaches. This is shown in figure 2.5, where *RL* and *Fuzzy RL* refer to two particular reinforcement earning algorithms they used and *ADC* (Adaptive Duty Cycle) is a static algorithm. We hope to continue and improve upon their work, outmatching them using neural networks as the tool for training the agent.

2.4 Hardware constraints

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

Unfortunately, neural networks are infamous for demanding a large amount of computational resources. This infamy comes largely from the *training* of neural networks, which can take days on even the most powerful of supercomputers. Performing any sort of neural network training **on** IoT devices is completely infeasible with the sort of hardware specifications on State-of-the-art microcontrollers today. Luckily however, invoking responses from these networks **after** training consumes resources on a scale many orders of magnitude below. This is why it's potentially feasible to utilize neural networks on IoT devices with memory capabilities as small as most microcontrollers.

2.4.1 Memory consumption estimation

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

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

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

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

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

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

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

2.4.2 Energy consumption estimation

In addition to estimating memory footprints, we need to look at the battery consumption imposed by invocation of a neural network. A 2017 paper with the title "Energy Consumption Estimation for Energy-Aware, Adaptive Sensing Applications" [Tamkittikhun, 2019] is of particular interest given our goal of IoT power management. One conclusion we can draw from their work is the following. Given an action for the CPU to perform, the energy consumed by the action depends almost solely on the amount of *time* spent on it by the CPU. In other words, given the time an action takes, we can usually calculate the amount

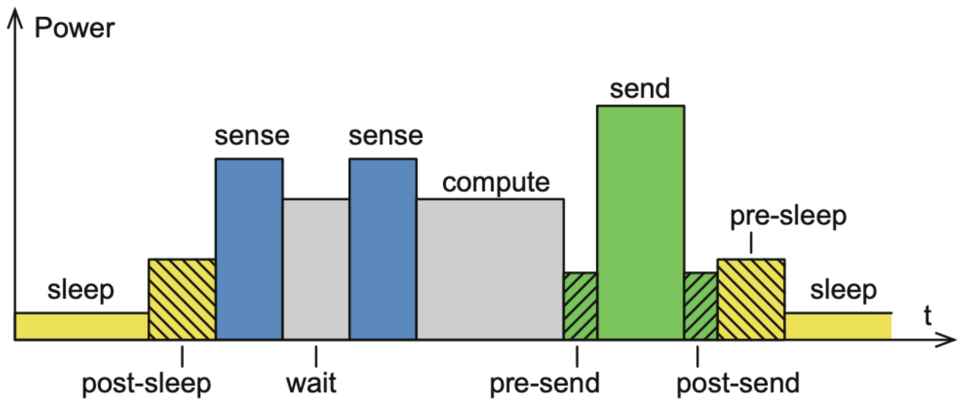

Figure 2.6: An abstract model of the energy consumption of different phases in an IoT sensing node's life cycle. Taken from [Tamkittikhun, 2019].

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 2.6. 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 2.6. E, then, gives us the total energy consumption of the node's entire life cycle by summing the consumption of each phase.

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

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

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

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

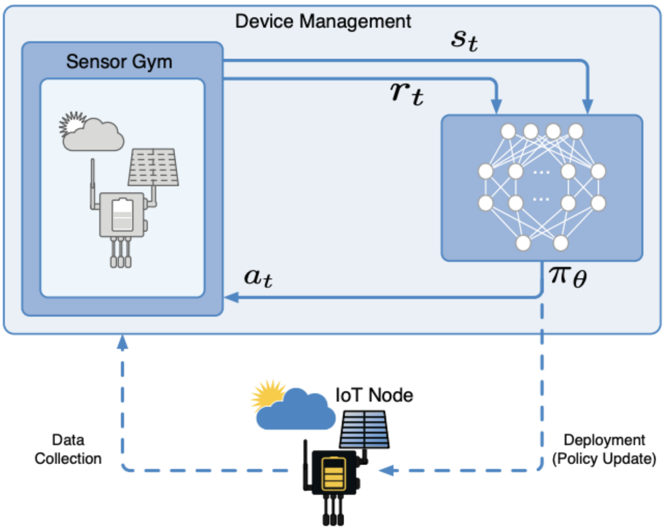

Figure 2.7: The intended agent/environment setup of [Murad et al., 2019]. The upper parts represent training and invocation from a neural network, while the lower is the updating of the policy of an actual IoT device. This lower part was only simulated in their work. Taken from [Murad et al., 2019].

2.4.3 Applicability of neural networks in the IoT domain

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

However, we are not the first to consider this approach. [Murad et al., 2019] is a work in which a reinforcement learning agent is trained using neural networks, then deployed in a simulated IoT scenario. Their setup is

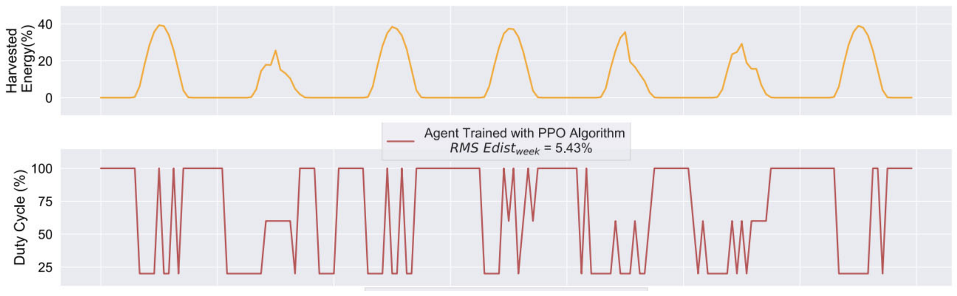

Figure 2.8: Graphs showing simulated solar power and corresponding duty cycle chosen by an agent trained using neural networks. Taken from [Murad et al., 2019],

shown in figure 2.7. 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 2.8. 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., 2019].

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

Chapter 3 Methodology

Chapter 3 does X. Section 3.1 does A. **??** does B. **??** does c.

3.1 Research question and context

Our stated Research Question (RQ) is as follows: **Are we able to *implement* neural networks on *today's* IoT devices in such a way that they utilize energy better than if they had used previous approaches?**

3.2 Framework for experimentation

Some content.

3.2.1 Sense cycle

Sense cycle.

3.2.2 Neural network on a microcontroller

NN micro.

Figures:

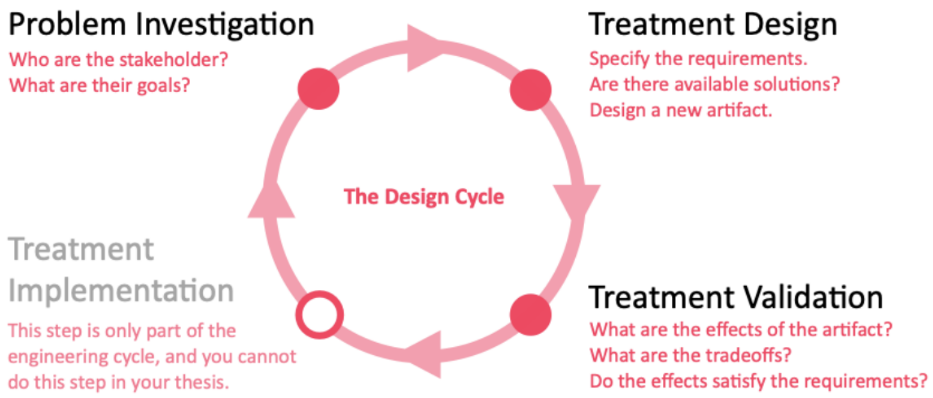
Problem Investigation
Who are the stakeholder?
What are their goals?
Treatment Design
Specify the requirements.
Are there available solutions?
Design a new artifact.
The Design Cycle
Treatment Implementation
This step is only part of the engineering cycle, and you cannot do this step in your thesis.
Treatment Validation
What are the effects of the artifact?
What are the tradeoffs?
Do the effects satisfy the requirements?

Figure 3.1: The iterative nature of design science. Taken from [Des, 2019].

Device name	**CPU**	**Flash**	**RAM**	**mbed-os**
nRF52840 (Berg)	64 MHz	1MB	256kB	Yes
nRF9160 (NB-IoT)	64 MHz	1MB	256kB	**No**
nRF52-DK (BLE)	64 MHZ	192KB + 512KB	24KB + 64KB	Yes

Table 3.1: Comparison of the most important specifications of the devices used in [Berg, 2019] and those considered in our project. Taken from [Berg, 2019] pp.38 and [Semiconductor, 2019].

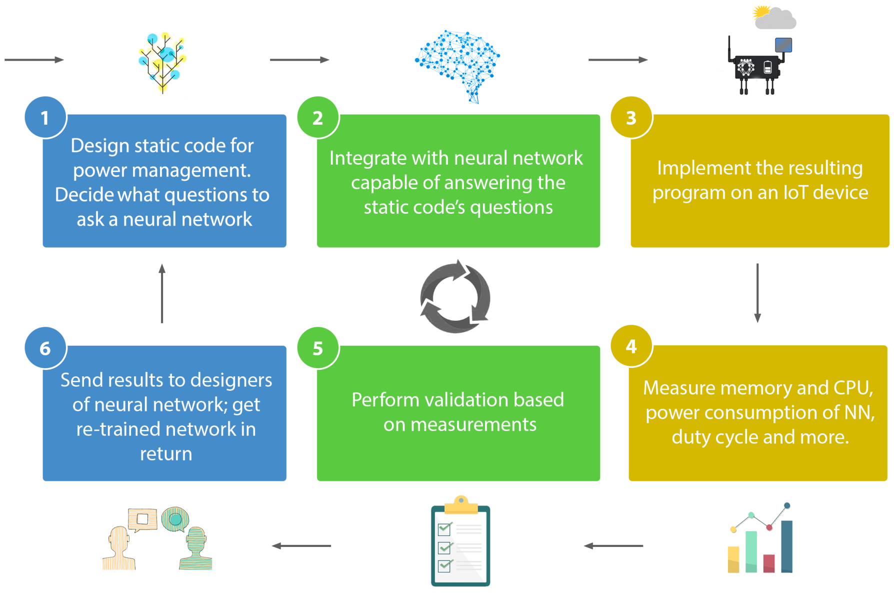
1
Design static code for power management. Decide what questions to ask a neural network
2
Integrate with neural network capable of answering the static code's questions
3
Implement the resulting program on an IoT device
4
Measure memory and CPU, power consumption of NN, duty cycle and more.
5
Perform validation based on measurements
6
Send results to designers of neural network; get re-trained network in return

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

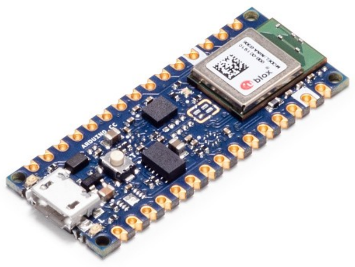

Figure 3.3: Arduino Nano 33 BLE, the physical IoT device we plan to use.

Chapter 4
Experiments

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

4.1 Memory consumption

4.1.1 Static sensing cycle

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

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

4.1.2 Neural network invocation

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

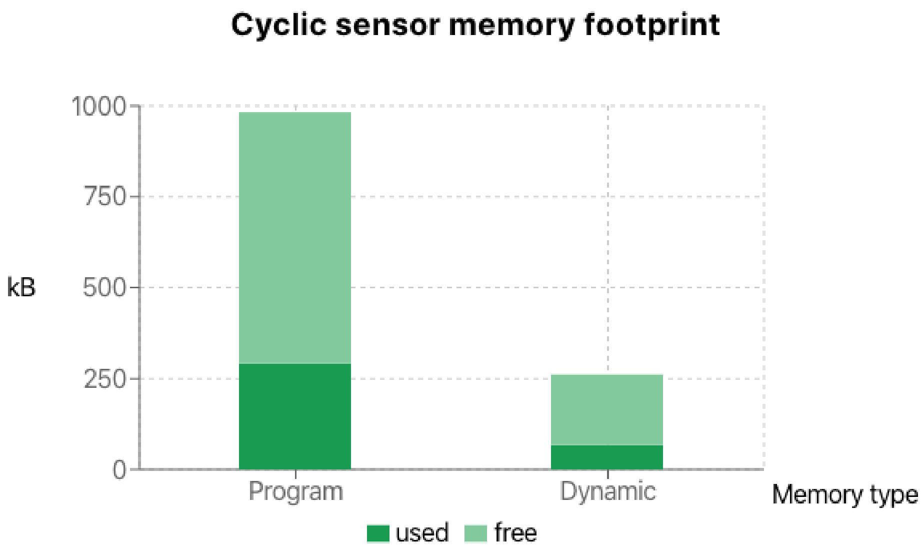

Figure 4.1: The memory consumption of our static program.

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

4.1.3 Total memeory

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

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

void setup() { }

void loop() { }
```

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

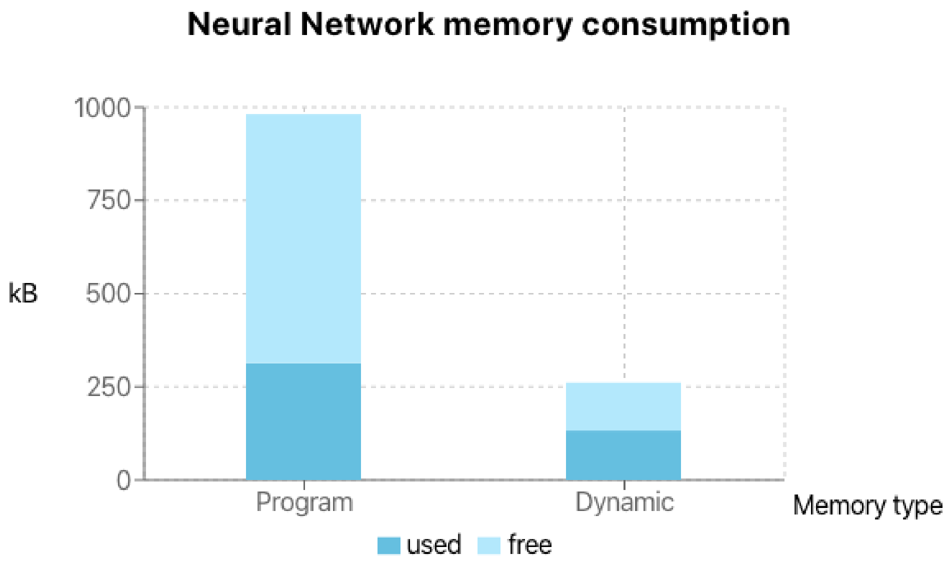

Figure 4.2: The memory consumption of the neural network.

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

$$\begin{aligned} M_{tot} &= M_{sense} \cup M_{nn} \qquad (4.1) \\ &= M_{sense} + M_{nn} - M_{sense} \cap M_{nn} \qquad (4.2) \end{aligned}$$

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

$$\begin{aligned} M_{program} &= 292kB + 314kB - 76kB \\ &= \mathbf{530kB} \end{aligned}$$

Likewise, the collected data for the dynamic gives us:

$$\begin{aligned} M_{dynamic} &= 67kB + 132kB - 42kB \\ &= \mathbf{157kB} \end{aligned}$$

The findings are summarized in table 4.1.

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

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

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

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

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

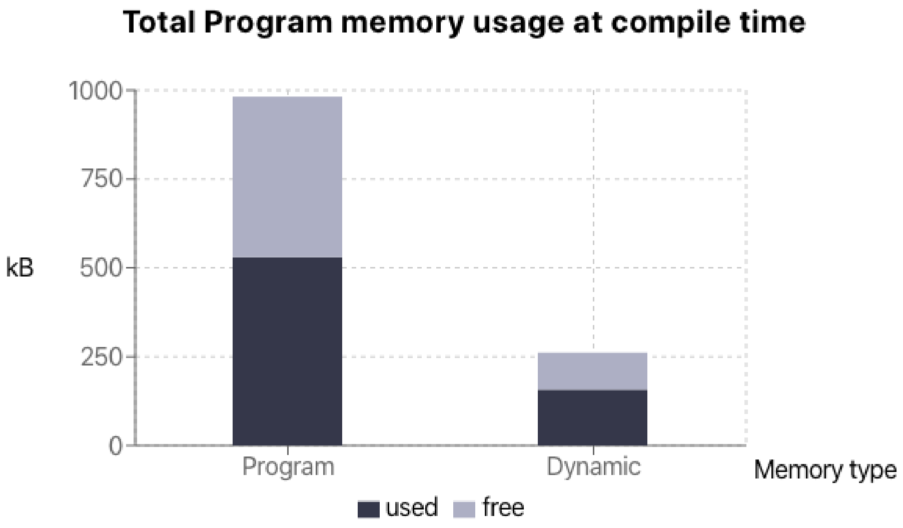

Figure 4.3: Total memory consumption at runtime.

4.2 Runtime memory

The contents of this subsection

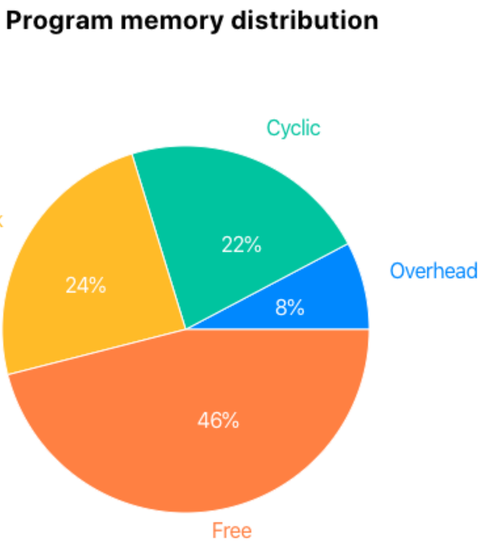

Figure 4.4: The distribution of program memory at compile time.

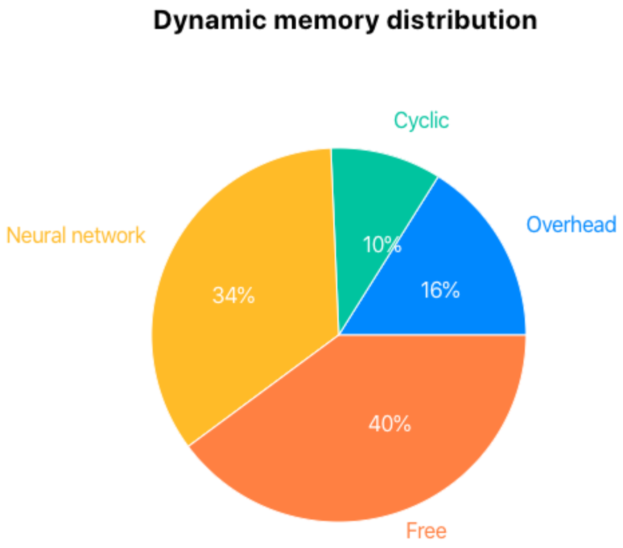

Figure 4.5: The distribution of dynamic memory at compile time.

Chapter 5 Results

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

5.1 Some section

Introduction to the topic

5.1.1 Some subsection

The contents of this subsection

Chapter 6
Concluding remarks

Concluding remarks!

Bibliography

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

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

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

[Ope, 2018b] (2018b). Openai spinning up: Intro to policy optimization. *https://spinningup.openai.com/en/latest/spinningup/rl*$_i$*ntro3.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*$_i$*ntro2.html.*

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

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

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

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

[Res, 2020] (2020). Feed-forward neural network overview. *https://www.researchgate.net/figure/Feedforward-neural-network*$_{f}ig1_{3}$29586439.

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

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

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

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

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

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

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

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

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

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

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