Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Hans Olav Lofstad

A Study of Artificial Neural Networks on 8-bit Microcontrollers

Master's thesis in Electronic Systems Design
Supervisor: Guillaume Dutilleux
Co-supervisor: Amund Aune, Asgeir Schanke, Erling Holten Wiken
June 2021

NTNU
Norwegian University of
Science and Technology

MICROCHIP

Hans Olav Lofstad

A Study of Artificial Neural Networks on 8-bit Microcontrollers

Master's thesis in Electronic Systems Design
Supervisor: Guillaume Dutilleux
Co-supervisor: Amund Aune, Asgeir Schanke, Erling Holten Wiken
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Abstract

The topics of machine learning and 8-bit microcontroller units (MCUs) are generally regarded as incompatible, due to the limited computational resources available. Artificial neural networks (ANNs) are an exception to this general statement, seeing as the resource demanding process of this technique happens during its training phase, while performing model inference requires significantly less processing power. ANNs can therefore be trained on a computationally resourceful machine, and later deployed on the 8-bit MCU.

This thesis explores the feasibility of running such networks on 8-bit MCUs. An implementation process is presented, as well as different optimization techniques to limit the data and program memory requirements for the system. Several tests are presented, evaluating the performance of the deployed models, and the impacts of different optimization techniques.

The thesis concludes with stating ANNs on 8-bit MCUs as well within the realm of feasibility. Accuracy drops for deployed models are shown to be virtually non-existent, and inference times are shown with typical values below 50ms.

Sammendrag

Temaene om maskinlæring og 8-bit mikrokontrollere blir generelt ansett som inkompatible, som følge av den begrensede prosesseringskraften man har tilgjengelig. Kunstige nevrale nettverk er et unntak av dette generelle utsagnet, som et resultat av at den ressurskrevende prosessen skjer under treningsfasen av nettverket. Behovet for prosesseingskraft er betydelig redusert når nettverket er ferdig trent, og implementert til å utføre dens designerte oppgave. Slike nevrale nettverksmodeller kan derfor trenes på en maskin med tilstrekkelig prosesseringskraft, og deretter overføres til en 8-bit mikrokontroller.

Denne masteroppgaven utforsker mulighetene for å kjøre slike nevrale nettverk på 8-bit mikrokontrollere. En implementasjonsprosess blir presentert, i tillegg til forskjellige optimaliseringsteknikker, med formål å begrense behovet for program- og dataminne. Flere tester blir presentert, som undersøker ytelsen til de nevrale nettverkene på 8-bit systemer, og som undersøker påvirkningen av de forskjellige optimaliseringsteknikkene.

Oppgaven konkluderer med at 8-bit mikrokontrollere er godt rustet til å kjøre kunstige nevrale nettverk. Implementasjonene viser til at prediksjonsnøyaktigheten i nettverkene ikke endres mellom maskinen det trente på, og mikrokontrolleren det kjører på. Eksekveringstiden til nettverksprediksjonene har vist typiske verdier lavere enn 50ms.

Preface

The following work presents a masters thesis, written in the occasion of finalizing a master degree in Electronic Systems Design at the Norwegian University of Science and Technology (NTNU). The thesis is written in collaboration with Microchip Technology Inc., who supplied guidance and expertise on 8-bit microcontrollers, as well as general supervision in the direction of the thesis. The supervisors from Microchip Technology include Amund Aune, Asgeir Schanke and Erling Holten Wiken. Guillaume Dutilleux served as the thesis supervisor from NTNU, and provided guidance in report writing, and the general direction of the thesis.

Contents

Figures

Tables

Code Listings

Acronyms

AI Artificial Intelligence. 2, 5, 6, 43, 44

AIfES Artificial Intelligence for Embedded Systems. 19, 20, 38, 45

ANN Artificial Neural Network. 1–3, 5–9, 11, 16–18, 20, 21, 23, 25–27, 30, 37, 39, 40, 44, 45, 47, 48

API Application Programming Interface. 11, 17, 18, 21, 23, 40

BCE Binary Crossentropy. 12, 22

CCE Categorical Crossentropy. 12, 13

CNN Convolutional Neural Network. 14–16, 45

CPU Central Processing Unit. 17

MAE Mean Absolute Error. 12

MCU Microcontroller Unit. xix, 1–3, 17–20, 23, 25–27, 30, 32, 37–41, 43–45, 47, 48

MSE Mean Squared Error. 11, 12

MSLE Mean Squared Logarithmic Error. 12

NNoM Neural Network on Microcontroller. 19–23, 25, 26, 37, 38, 40, 41, 44, 45, 47, 48

RNN Recurrent Neural Network. 7, 8, 45

SCCE Sparse Categorical Crossentropy. 13, 25

SH Squared Hinge. 12

Glossary

cloud computed AI Cloud computed AI refers to resource constrained devices executing machine learning algorithms, by remotely communicating with a more powerful system, and outsourcing the heavy processing tasks. 1

edge computed AI Edge computed AI refers to microcontroller units performing machine learning algorithms, without outsourcing the processing to more powerful systems. 2

forward-pass Forward-pass refers to the operation of porting a pretrained neural network to a resource constrained device, such as a MCU. 19

inference In machine learning, inference refers to the process of running a model, by inputting data, and prompting it to return a prediction. 16, 18, 19

Chapter 1

Introduction

1.1 Problem Statement

The aim of this thesis is to evaluate the feasibility of implementing Artificial Neural Networks (ANNs) on 8-bit architecture Microcontroller Units (MCUs). Implementing neural networks on such resource constrained devices pose a number of challenges. These challenges will be identified, and possible solutions will be presented. The work will focus on implementation, and finding techniques and best practices to optimize the ANN performance on 8-bit MCUs.

1.2 Background

The topic of machine learning and ANNs seem to get more relevant by the day. These topics have long been associated with a need for great processing power, an association that holds true to a certain degree. Most machine learning algorithms require powerful processors to execute, but there is a clear exception to this general rule; namely ANNs. Before such an algorithm can be deployed, it needs to undergo a training phase, where the network learns the desired behavior for an application. The training phase does require a significant amount of processing power. However, when the training is complete, the network only requires a fraction of said processing to execute. This allows for networks to be trained on powerful systems, and later transferred and run on resource constrained devices. This trait makes ANNs particularly well suited for embedded applications, where processing power is a scarce resource. This thesis will tackle the challenge of implementing ANNs on 8-bit MCUs.

1.2.1 Motivation

Machine learning on 8-bit MCUs have traditionally been performed through *cloud computed AI*, where the MCU remotely outsources the heavy processing tasks to a more powerful system. This method comes with a set of weaknesses, which

renders the technique unsuited for certain applications. *Edge computed AI* is a contrasting technique to cloud computing, where all processing occurs on the MCU. Edge computing could prove to be a suitable alternative to cloud computing for applications where the latter is unsuited. According to [1], key reasons why cloud computing might be unsuited is privacy, latency, and network congestion. Some applications might be collecting sensitive data, which poses the issue of privacy. In cloud computed systems, the MCU transmits data wirelessly to a cloud server. This data could potentially be collected by unauthorized parties, which depending on the application at hand, could pose security threats. This issue would be eliminated in an edge computed system, seeing as no data leaves the MCU wirelessly. Another potential issue is that of response latency. Wireless communication introduces a number of networking variables that will affect the response time of the cloud server. These variables do not affect an edge computed system, which will be able to deliver results with a more predictable latency. If a system consists of multiple nodes, all of which rely on cloud computing, one might eventually experience issues with network congestion. In such a case, transferring some, or all nodes to an edge computed solution, will help improve the performance and stability of the system. In addition, sending information wirelessly is a power demanding task, which might prove overwhelming for some battery powered solutions. Replacing cloud computed ANNs with an edge computed alternative could significantly improve the battery lifetime of such devices. Lastly, it is worth noting that cloud systems are expensive to implement compared to edge computed systems. Edge computed nodes only cost as much as the MCU and potential supporting hardware, while a cloud server will require a larger investment to implement.
Cloud computing is a solution that provides a great deal more computing power. Computationally heavy tasks could certainly be executed with lower latency times in cloud servers, compared to resource constrained MCUs. There is no definitive answer to which approach is better in general, but rather a consideration that needs to be made on an application specific level.

1.2.2 8-bit Motivation and Current State of Research

Another key aspect of the thesis topic is that of MCU architecture. MCUs are commonly designed with 8-bit, 16-bit or 32-bit architectures, all with varying strengths and weaknesses. Compared to 32-bit MCUs, 8-bit is more cost effective, consumes less power, is easier to work with (implementation-wise), and usually carries a smaller form factor. 32-bit MCUs have greater clock speeds, computing power, and generally more internal memory. Due to the superior computing power and internal memory, 32-bit MCUs have traditionally been the architecture of choice for embedded Artificial Intelligence (AI) applications. In fact, the vast majority of research done on embedded AI have been focused on 32-bit MCUs. The author was not able to locate a single published resource exploring implementation details, different trade-off aspects, or the general feasibility of ANNs on 8-bit MCUs. A small number of publications and demonstration videos have been found, showcasing a proof of concept, without emphasis on implementation and performance details. [2] presents an 8-bit ANN implementation that successfully tracks the maximum point of solar irradiance, for use with solar energy harvesting. It does not present details surrounding the 8-bit aspect, but serves as yet another proof of concept. This thesis thus serves to fill the void, and to provide a thorough investigation of implementation methods, optimization, performance, and the general feasibility of ANNs on 8-bit MCUs.

1.3 Author Qualifications and Intended Audience

At the start of this thesis project, the author had no previous knowledge or experience on the subject of ANNs. He did however start out with proficiency in both Python programming, and C programming for MCUs. Additionally, seeing as this is a masters thesis, the author started with the qualifications provided by a five year degree in electronics, with a major in sensor systems. The thesis assumes that the reader is familiar with general programming concepts, with a main focus on the programming languages Python and standard C. Previous experience with MCU programming is also assumed, as well as experience with general engineering concepts. The thesis does not assume knowledge of ANNs. All relevant machine learning concepts will be presented in chapter 2.

Chapter 2

Theory of Artificial Neural Networks

2.1 Core Concept

The fundamental theory presented here is largely based on material from [3] and [4]. These sources, with emphasis on the latter, are recommended for further reading, should the reader be interested in learning more.
ANNs are a popular type of AI that are conceptually modeled after the way human brains work. The brain consists of intricate networks of a special cell type called *neurons*, which communicate with one-another using electrochemical signals. A neuron will emit an electrochemical signal based on the excitement it receives from other neurons. ANNs borrow from this concept. In short, a ANN is a network consisting of a number of digital neuron models, interconnected in a certain topology (see figure 2.1). Each neuron model receives a number of input signals, and outputs a single resulting signal. This output is thus fed to the next layer of neurons, whose outputs are influenced by the signal received from the initial neuron. Follow the arrows in figure 2.1 to see the directions of data flow.
A network will receive an input, which excites the network neurons in a certain way, and in the end grants a resulting output. This input can be sensor data, images, or any other desired data type. A commonly used example is the process of recognizing handwritten digits. The network is fed images of handwritten digits, and is tasked with returning an integer value from 0 to 9. In order to do this the network needs to be trained, so that each neuron knows at which input data it should "fire" (binary high signal), and when not to. The training is done by feeding the network a large dataset of such images, and then simply allowing the network to guess the correct output. Each image has a label attached to it, containing the correct answer for the particular image. Each time the network attempts a guess, it adjusts the firing threshold of the neurons individually, in an attempt to improve at the task. Initially the network will have a success rate of about 10%, which in an output range of 10 elements is pure guesswork. It will however improve gradually for each image it analyzes. After repeating this process several thousand times, it

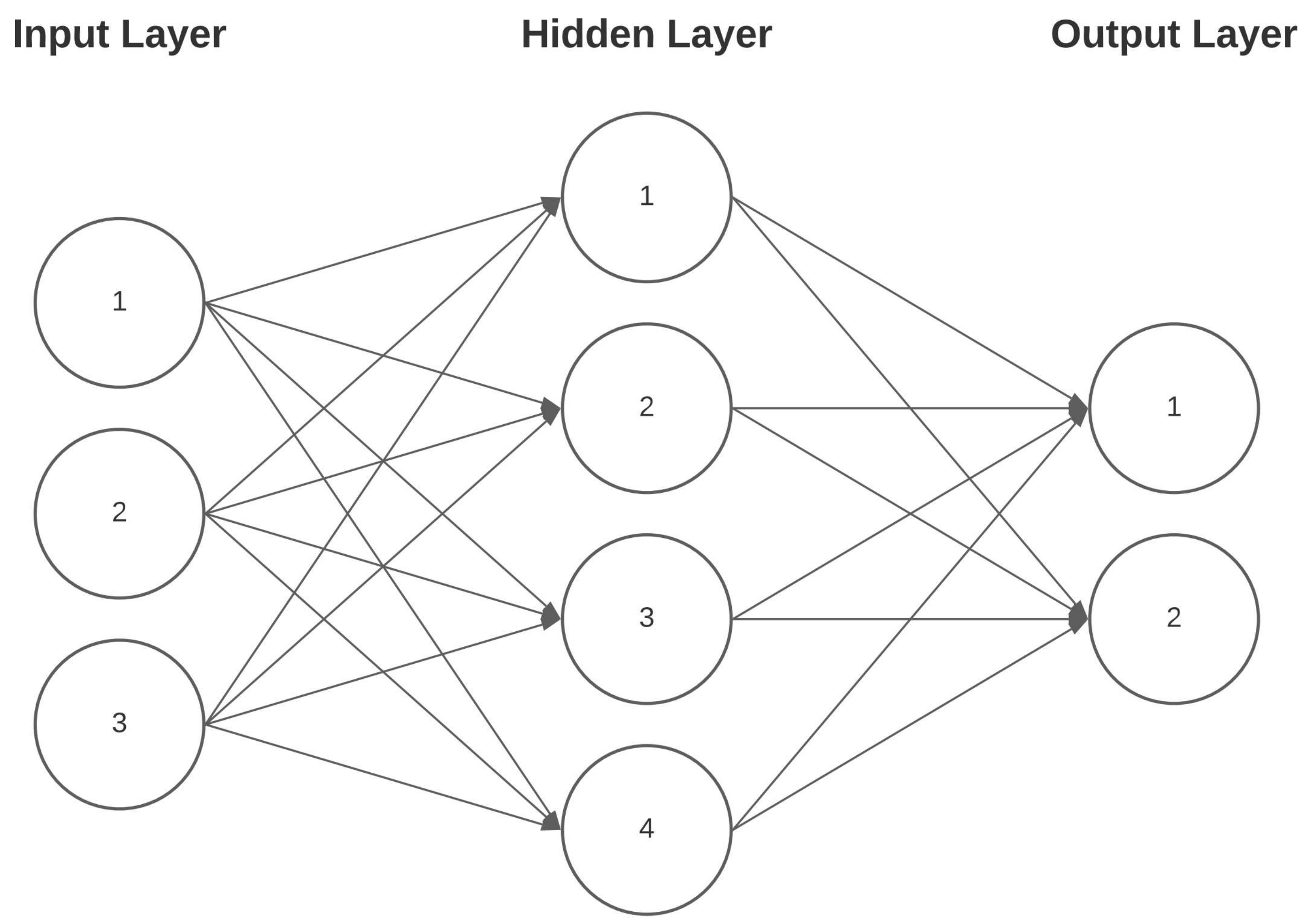


Figure 2.1: Topology of a basic feed-forward Artificial Neural Network.

will eventually become quite good at the task. The accuracy will certainly depend on several different factors, such as network topology, size of dataset, and a wide variety of other factors. The current state of the art handwritten digit classifying networks reach an accuracy level upwards of 99.84% [5].
One of the major advantages of ANNs is the ability to treat it somewhat as a black box. They can solve complex problems with just the need of example data, while the programmer can abstract away a great deal of the details. Imagine having to solve the handwritten digit problem without the use of such AI; This would no doubt be a challenge. As opposed to other AI techniques, ANNs require very little computing power to run. The resource intensive part of ANNs occur during training, which potentially is a one-time process.

2.2 Network Topology

The topology of the network is usually organized in layers, and are categorized as an input layer, an output layer, and a number of hidden layers in between. The network architect can vary the number of hidden layers, and the number of neurons in each layer of the model, based on the particular use-case. The abilities of the network rely heavily on this topology. The network in figure 2.1 has three nodes in its input layer, which means it accepts a total of three data points as input. This network would not be able to handle image data, considering each pixel of the image would require its own input node. The output layer in turn has

a layer width of two. This of course only allows for two output values, e.g. true or false. For the handwritten digit classifier one would need ten output nodes, one for each digit.
The hidden layers in between is where the network properties are decided. Different use-cases require different topologies to perform well. It is difficult to properly generalize how to design a network given a use-case, as this is a problem largely solved through experience, and trial-and-error. However, researching which topologies have been successful for similar tasks is a much used approach.
The figure 2.1 network is a feed-forward ANN. This means that the information only flows in one direction, from a high layer to a lower layer. This differs from what is known as Recurrent Neural Networks (RNNs). RNNs are networks where one or more nodes loop in the backwards direction. Sending information back to previous layers gives the network a form of short-term memory. This is very useful for applications such as speech recognition, where previous words matter a great deal when deciphering the meaning of a sentence.

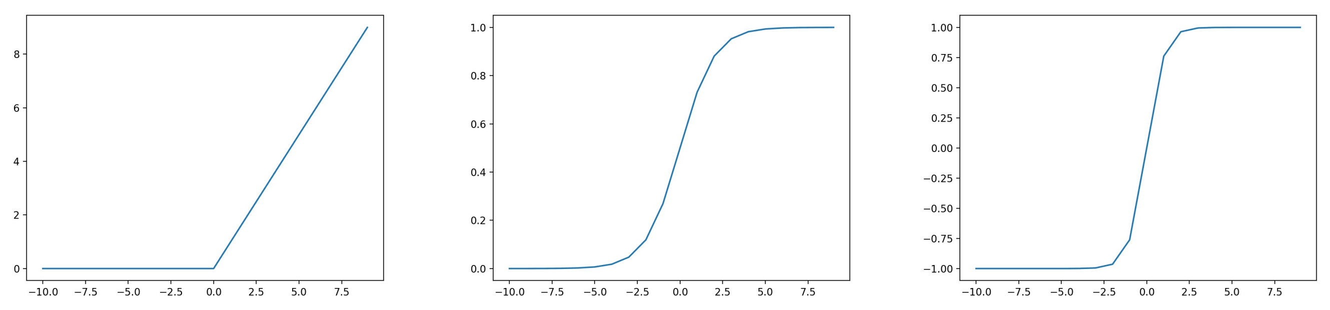
2.3 Activation, Weights and Biases

This subsection is written with a basis in the most basic form of ANNs, namely densely connected networks. This is a network where every node in a layer sends its output to every node in the next layer. Figure 2.1 displays such a network.
Each connection between two nodes holds its own unique *weight*. The weight is a number that specifies how much influence that particular connection has on the *activation* of the receiving node. The activation is, in short terms, the output value of the node. In addition, each node has a *bias*, which can be thought of as a threshold for how great the activations at the input of a node need to be, in order to activate said node. The activation (output value) of a node is determined by applying an *input function* on the node inputs, and feeding this result to an *activation function* (see section 2.4). The input function can be calculated by taking the weighted sum of all the inputs of a node, and then adding the bias. Every connection n of the total input count n_{tot} at the node input, has an activation value a_n, and a weight w_n. The node itself has a bias b. The input function f_{in} then becomes:

$$f_{in} = (\sum_{n=0}^{n_{tot}} w_n * a_n) + b \tag{2.1}$$

2.4 Activation Functions

As mentioned in section 2.3 the input function is usually coupled with an activation function when determining the activation level of a node. To clarify, activation functions are utilized in every network type, and is not specific to densely connected networks. The following is based on material written by Dr. Jason Brownlee



(a) ReLU **(b)** Sigmoid **(c)** Tanh

Figure 2.2: The three most common activation functions for hidden layers. X-axis is input value, y-axis is output. [6]

[6]. The information is summed up in table 2.1.
There are several different types of activation functions, which work well for different applications. Choosing the right activation function will help the network train more efficiently, which in turn makes the network perform better. Activation functions are usually defined for each layer, and the output from every node in that layer will then pass through the defined activation function, before propagating to the next layer. The first distinction to make is between the hidden layers and the output layer, and which activation functions serve their respective purpose. It is most common for all hidden layers in a neural network model to share the same activation function, and for the output layer to have a different one. The three main activation functions used for the hidden layers are called *ReLU*, *Sigmoid* and *Tanh* (see figure 2.2). ReLU is the most commonly used of the three, and is viewed as the default activation function. Sigmoid and Tanh were the defaults of their time, in roughly the early 1990s and the 2000s respectively[6]. However, they all have their their use-cases in modern neural networks. In short, the recommendation is to use ReLU for most cases, except for in RNNs, where both Sigmoid and Tanh are commonly used[6].
The activation functions for the output layer is selected based on the type of output the network generates. *Regression* and *classification* are two common problem types in machine learning. Regression problems attempt to predict a numerical variable, and as a result it needs only one node in its output layer. This output node will hold the predicted numerical value. For such problems it is advised to use the *linear* activation function[6], whose output value is the same as its input value.
Classification problems revolve around placing some kind of input data into a fitting category (class). For example, if the input data is an image of an apple, then the ANN should place this in the *apple* class, and not the *strawberry* class. There are a few more factors to consider when choosing an activation function for classification problems. This problem type can be subdivided into three more categories:

- Binary classification

  - One output node; Two classes; Input belongs exclusively in one (mutually exclusive). Use the **Sigmoid** activation function[6].
- Multi-class classification
  - Three or more output nodes; Three or more classes; Input belongs exclusively in one (mutually exclusive). Use the **Softmax** activation function[6].
- Multilabel classification
  - Two or more output nodes; Two or more classes; Input can belong in several (mutually inclusive). Use the **Sigmoid** activation function[6].

Both binary and multilabel classification problems are suitably solved by the Sigmoid activation function. The reason for this is that the Sigmoid will return a number between zero and one. This value can be interpreted as a probability of class membership. Each output node has such a value, representing how likely the network believes the input value to belong in that particular output class. For the case of binary classification, where there is only one output node, a value close to zero points to one class, and a value close to one points to the other. Multi-class classification suits use of the Softmax function. The reason for this is that all the outputs are mutually exclusive, meaning the input data can only belong in *one* of the output classes. The Softmax function will assign a probability value between zero and one to each of the output classes. What separates this from the Sigmoid function is that, all the output probabilities using Softmax sums up to one. The class with the highest predicted probability is thus the mutually exclusive prediction.

Activation Functions			
Hidden Layers		Output Layer	
Feed-Forward	Recurrent	Regression	Classification
ReLU	**Sigmoid** or **Tanh**	**Linear**	Binary: **Sigmoid** Multi-class: **Softmax** Multilabel: **Sigmoid**

Table 2.1: Table of the most commonly used activation functions for a selection of network types.

2.5 Learning in ANNs

One of the great advantages of ANNs is their ability to learn on their own, by simply feeding it example data. This process does of course not happen on its own, and involves a fair bit of calculus to perform. The most common approach when creating an ANN is to use premade libraries, such as Tensorflow and PyTorch, which has state of the art learning algorithms implemented. It is therefore outside the scope of this thesis to explore the details of such algorithms. Should however the reader be interested in exploring this topic, it can be recommended to watch the video series by Grant Sanderson (3Blue1Brown on YouTube) on the topics of *gradient decent* and the *backpropagation* algorithm [7]. The book *Artificial Intelligence: A Modern Approach*[3], chapter 18.7, also provides a detailed explanation of these concepts. The following sub-chapter will present and refer to these concepts on a high abstraction level, and will be used to explain central learning parameters one would encounter while using machine learning libraries such as Tensorflow.

2.5.1 Backpropagation, Optimizers and Loss Functions

The loss function is used to measure how good a prediction from the network is, by comparing the prediction to the label of the input data. Put in the context of the handwritten digit problem; if the input image depicts the digit four, it is expected that the output node representing this digit will return a probability close to one. Similarly, for the image of a four, one would expect all other output nodes to return a probability close to zero. The loss function calculates a performance score, called the *loss*, based on the difference between this expected output value, and the actual output value. A very simple loss function would be the absolute value of this difference. E.g. if the input is a four, and the output node representing this digit returns 0.85 probability, then the loss for that *particular node*, for that *particular training example*, would be calculated:

$$loss = abs(1 - 0.85) = 0.15$$

This is just one example of a loss function, among several that are commonly used. In order to learn, the network needs to tweak its different weights and biases until the generated output matches the expected output. The loss function is the mechanism that informs the network of which weights and biases to adjust, in order to perform better. For an arbitrary training example, the loss is calculated for each output node. The value of the loss indicates how much the individual nodes wishes to increase or decrease their respective output value. In order to influence this, the node will indicate that it wishes to adjust certain weights in its input connections, which in turn would nudge the output value for this particular training example in the right direction. The output node will also indicate to the relevant nodes in the previous layer, which activations should be higher or lower. The nodes in the second to last layer will then perform this same operation, by requesting changes to weights in their input connections, and activations according to its own calculated loss. This loss calculation is based on the difference between the current activation value of the node, and the sum of the activation change requests from all the nodes in the output layer. This operation will work its way through every layer of the network. This is called the *backpropagation* algorithm. Once this has been done for every training example, the individual weight change requests are averaged, and these average values are ultimately used when updating the weights of the network.

This averaged list of weight changes can be seen as the negative gradient of the network, indicating which changes to make in order to minimize the overall network loss. This approach at minimizing the network loss is know as gradient descent. The concept of optimizers are central in ANN theory, and gradient descent is the foundation for the most commonly used optimizers. At its core, training neural networks is an optimization problem, where the goal is to find the weight and bias settings at which the overall network loss is at a minimum. The most commonly used evolution of the gradient descent algorithm is the *ADAM* optimizer, and can be regarded as a default choice of optimizer. There are however times where one would prefer to use a different optimizer, but this falls outside the scope of this thesis. Dr. Jason Brownlee covers the topic of choosing a suitable optimizer for different use-cases, and is a recommended read for those interested [8].

2.5.2 Choosing a Loss/Cost Function

The recommendations made in this subsection are based on material written by Dr. Jason Brownlee[9] and Peltarion[10].
A noteworthy distinction often encountered is that of loss function and cost function. A loss function is, as described previously in this chapter, an evaluation that happens on a node level. A cost function on the other hand is used to measure the overall network performance. These terms do however get mixed up frequently, and are often used interchangeably. A reason for this is that a cost function contains a loss function term, but adds a new layer of functionality. For example, the cost function might take the squared difference of each output node in the network, and then average these values. The result is then a single value which represents the overall loss of the network. In this case the loss function is the operation happening at the node level, namely the term which squares the differences. This, combined with the averaging term, makes up the cost function. The ANN training presented in this thesis utilizes the Tensorflow Keras Application Programming Interface (API), which refers to both loss and cost, as losses. The following will present recommendations for which cost function to use for different situations. The choice of cost function, quite similarly to the choice of activation function, depends on the type of prediction problem, which is either regression or classification (see section 2.4 for definitions).

Regression - The default loss function for regression problems is the Mean Squared Error (MSE). This cost function squares the difference between the predicted value and the actual value at the output node, and averages the results across a set of training examples. This cost function punishes the network for large numerical mistakes, due to the differences being squared. This is an efficient approach for most use-cases, and it is therefore recommended to use MSE unless the network has clear reasons not to. See [11] for further details on MSE.
One reason to not use MSE is if the output variable has a large numerical range of possible correct values. E.g. if the network for one input has the correct answer 25,

and for another input the correct answer 1500. In this example the possible span of output values is significant, numerically speaking. Since MSE punishes large numerical differences, it might be better to use a different loss function. Mean Squared Logarithmic Error (MSLE) introduces a logarithmic term into MSE, which makes the cost function respond to percentual differences between expected and actual output, as opposed to numerical differences. See [12] for further details on MSLE.
If the dataset in question contains several outlier results, it might affect the training performance. A cost function used to remedy this is the Mean Absolute Error (MAE), which uses the absolute value of the differences, and averages this across several training examples. This cost function will reduce the effect outlier results may have on the network performance. See [13] for further details on MAE.

Binary Classification - The default choice of cost function for binary classification is called Binary Crossentropy (BCE). In classification problems the output value is interpreted as a probability of class membership. In the case where such networks are structured with only one output node, a value close to one indicates one class, while a value close to zero indicates the other. In such cases, BCE determines a cost value based on the difference between the target value (either zero or one) and the actual output value, averaged over several training examples. It is recommended to use BCE for binary classification, unless there are good reasons not to. See [14] for further details on BCE.
When using BCE the network returns probabilities of class membership. If the network architect is uninterested in these probability values, and is only interested in the binary true/false output, then the Squared Hinge (SH) cost function might be a good alternative to BCE. SH finds the maximum margin between the different classes, so that the network is fully confident that the prediction presented at the output is correct. This does of course not mean that the output will always be correct, but the network will still be just as certain in its prediction. This cost function is best used in combination with the Tanh activation function. See [15] for further details on SH.

Multi-class Classification - The default choice of cost function for multi-class classification problems is called Categorical Crossentropy (CCE). This cost function will output a class membership probability to each of the network output nodes. Using CCE the sum of said output values will always equal to 1, indicating that only one class can be the correct one. CCE is in principle similar to BCE, but is in contrast tailored towards three or more output classes. When using CCE it is important that the training examples are one-hot encoded. Take the handwritten digit problem as an example, which is a classic multi-class classification problem. The training examples each contain an integer label representing the digit displayed in the image. When using CCE this integer needs to be converted to a one-hot encoding. This means converting the label to a vector of ten elements, where every element except one is zero. If the training example contains the digit four, then element

number four in this vector needs to be a one, and every other element a zero. This is what is referred to as one-hot encoding. See [16] for further details on CCE.
One-hot encoding can in some cases be impractical. E.g. in some cases a network might have thousands of output classes, which for one-hot encoding would mean each training example holding a vector containing thousands of elements. This would pose significant memory requirements on the system in use. An alternative would be to use Sparse Categorical Crossentropy (SCCE). This cost function performs the exact same operation as CCE, but does not require the one-hot encoding. This spares the training procedure from the excessive memory requirements that can come from CCE.

The recommendations presented here are not an exhaustive list of possibly suitable cost functions, but provide an overview of the ones most commonly used for different problem categories.

2.5.3 Batch Size, Epochs and Learning Rate

Section 2.5.1 states that the weights of a model are adjusted only after the backpropagation algorithm has been executed on every training example in the dataset. In practice this is not the case. For datasets containing several thousand training examples, evaluating each of these simultaneously would require a significant amount of memory. To overcome this obstacle the concept of training batches is introduced. The dataset is divided into batches containing a user defined number of training examples. The backpropagation algorithm is then applied to each batch separately, and the weights are updated after each batch evaluation. Typical values for batch size are 32, 64, 128 and 256 training examples, but is not limited to any of these settings. The batch size does not need to be a power of two, and can be both greater and lower than the provided example values.
Another parameter one encounters when training a model is epochs. Epochs define how many times the network should evaluate each training example. If the epoch parameter is set to 50, the network will evaluate every training example 50 times. When training a model one needs to find a balance between three parameters, namely batch size, epochs and *learning rate*. The latter defines the gradient descent step size used by the optimizer. A typical starting value for the learning rate might be 0.001. Increasing the batch size generally means that the learning rate should be increased, and similarly reduced when decreasing batch size. A larger batch size will lead to reduced training time, but could negatively impact the training results. To increase the batch size without sacrificing results, one can increase the number of epochs [17]. This in turn will lead to longer training times, and so this is a matter of balance. Selecting appropriate values for these parameters is largely done through experimentation.

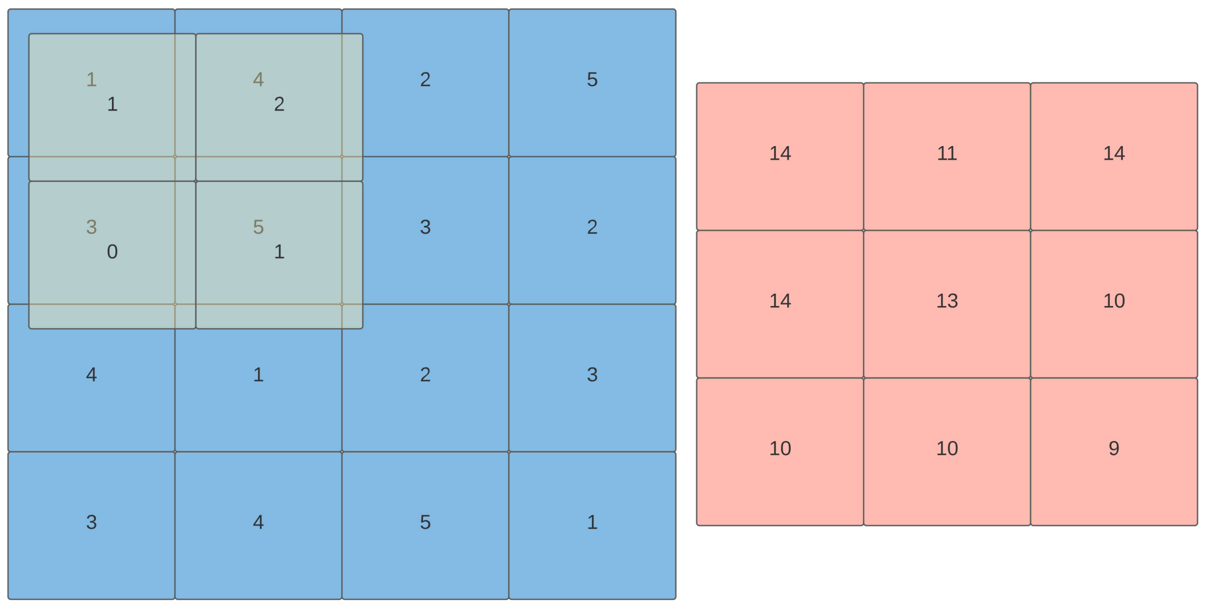


Figure 2.3: The convolution process visualized. The convolutional filter (yellow) convolves over the input data (blue), which grants the resulting data array (red).

2.6 Layer Types

There are several different layer types commonly used in neural networks. Up until now the only layer discussed has been the densely connected layer, where the output of a node is connected to the input of every node in the next layer. A large subset of layer types will not be presented, and the focus will target the ones relevant throughout this thesis. The theory presented is based on sources [18] and [19].

2.6.1 Convolutional Layers

Convolutional layers are the building blocks that make up Convolutional Neural Networks (CNNs), which are famed for their abilities in image recognition tasks (although useful in other applications). As discussed previously, the densely connected layers receive a one dimensional vector of inputs, and outputs a single value. Convolutional layers receive a two dimensional array of inputs, and outputs a two dimensional array of slightly smaller size[1]. Take as an example the handwritten digit problem; a CNN will receive a picture of a digit as an input, and forward the image data to the first convolutional layer. The purpose of this layer is to enhance certain features from the image (e.g. edges and circles). This is done through a process called convolution (hence the layer name).
The network architect specifies a number of *filters* for each convolutional layer. These filters are two dimensional arrays containing a set of weights. The convolution operation happens by sliding the filter over the input data, multiplying each filter element with the input data point it currently covers, and then adding the resulting value from each of the individual multiplications. This resulting value is then placed in a two dimensional array of its own. Figure 2.3 displays an example

[1]This refers to 2D convolutional layers. Both 1D and 3D convolutional layers exist, but are not as common.

of such an operation. Here, the yellow filter covers the input data so that the top leftmost "tile"[2] of both arrays align. Each filter value is thus multiplied with the input value tile it is currently covering. When each individual multiplication is summed up, it grants an output value, which is seen in the top leftmost tile of the red output array. Once this operation is done, the filter window slides one tile to the right, and performs the same operation. Once the filter has convolved over the top row of the input data for three iterations, it slides back to its leftmost position and down one tile. This is due to the filter dimensions exceeding the dimensions of the input data if it were to slide further to the right, which is not allowed in a standard convolutional operation. This is why the output array is of size 3x3, instead of 4x4 like the input array.

As mentioned, the network architect specifies how many such filters should be included in the convolutional layer, all of whom are initialized with an individual set of weights. Each filter would accept the full input array, and, independently from other filters, perform a convolutional operation on the data. Each of the filter outputs combined make up what is called a *feature map*. As mentioned, the role of the convolutional layers is to extract features in the input data; and the idea is that each filter, after the network is trained, will have a sensitivity to its own distinct feature. The feature map is a three dimensional array containing each of the filter outputs. This feature map is commonly put through one or more additional convolutional layers, which has the effect of extracting more advanced features for each layer it is passed through (to a certain extent). Take as an example a network that is trained on pictures of human faces. The first convolutional layer might detect edges, and the next layer might use the information about edges to extract the oval contour of the face. A few layers down the line, the network might be able to detect the full human face, even highlighting features such as the nose and eyes.

After each convolutional layer the feature map is passed through an activation function, whose output is also referred to as a feature map. Commonly used activation functions in such cases are ReLU and Tanh. Two parameters one encounters when implementing CNNs are kernel size and stride. Kernel size refers to the dimensions of the convolutional filter (the kernel size in figure 2.3 is 3x3). Stride refers to the step length of the filter; more precisely how many tiles it shifts after each convolution. The example above has a stride of one.

2.6.2 Pooling Layers

A challenge with convolutional layers is that they might become sensitive to the absolute pixel positioning of the different features, instead of the relative positioning. In high resolution images, small visually perceived positional changes could mean a significant offset in pixel indices. A remedying measure is to down sample the image, effectively reducing the amount of pixels in the image, while retaining the most vital information. Fewer pixels would mean smaller pixel index offsets.

[2]These tiles can more realistically be thought of as pixels in an image.

Pooling layers are the most commonly used down sampling technique in CNNs. There are two main types of pooling layers, namely max pooling and average pooling. Both can be visualized in the same way as the convolutional operation in figure 2.3. Imagine a window passing over the input array, but in contrast to the convolutional filter, this window does not have any weights. Instead, it takes the maximum value (max pooling), or calculates the average value (average pooling), of the input tiles covered by the window, and places the resulting value in an output array. The window continues to slide over the input data in the same fashion as during convolution, whilst performing the selected pooling operation. For pooling layers the network architect needs to specify the pooling type, stride, and pool size. Stride and pool size are similar to that described in section 2.6.1, although in convolution, pool size is referred to as kernel size. Pooling layers are normally applied after a convolutional layer and its activation function. It is very common to set a pool size of 2x2, and a stride of 2 [19].

2.7 ANN Optimization Techniques

A core component of implementing neural networks on resource constrained devices, is the process of optimization. There are two main techniques that are commonly used, namely *quantization* and *pruning*. Both aim to decrease the memory requirements of the neural networks, in order to fit more nodes and layers, while using as little resources as possible. Libraries such as Tensorflow and PyTorch use 32-bit float values to store weights, biases and other data. This means allocating four bytes of storage for every variable, regardless of the magnitude of its stored value. Quantization addresses this issue, and converts every variable in the model to 8-bit integers, thus reducing the size of the model by a factor of 4 [1]. Quantization also brings the benefit of a significant computational speedup, potentially reducing the model inference[3] time by a factor greater than 3 [20]. Note that this speedup factor applies when using the quantization functionality within Tensorflow, and does not necessarily apply to other libraries. Quantization is either done while the network is training, called *quantization aware training*; or after the training is done, which is called *post-training quantization*.
ANN predictions are determined by the fine-tuned weights and biases within the network. Each connection carries a unique weight that contributes to the final prediction; although, not every weight carries the same significance. Pruning is a technique that identifies connections within the network that carry less significance towards the final output, and then removes said connections. Removing connections means reducing the number of variables stored in the network, which in turn reduces the model size.

[3]In machine learning, inference refers to the process of running a model, by inputting data, and prompting it to return a prediction.

Chapter 3

Implementing ANNs on 8-bit MCUs

The following chapter will present the basics of implementing ANNs on MCUs. This includes how to choose a suitable MCU for the task, and which neural network frameworks to use. A simple implementation example will be presented, to highlight important design considerations; and to familiarize the reader with the relevant APIs, which is needed before diving deeper in chapters 4 and 5.

3.1 Choice of Microcontroller

There are a few key factors to consider when selecting an MCU on which to implement a neural network. The arguably most important factors are program memory and data memory, which can become a scarce resource in large networks. Program memory refers to the memory where the program code is stored, while data memory refers to the storage location of runtime variables. When choosing a MCU to perform experiments for this thesis, the author decided not to use devices that are top of the line for such applications. This decision being under the philosophy that, if neural networks are successfully implemented on less powerful hardware, then the relevant shortcomings and issues would be more pronounced, and thus easier to identify. If ANNs are deemed feasible on less powerful hardware, then surely top of the line devices are feasible for use as well.
The author decided to use an *ATmega4809* MCU; a device that holds 48kB of program memory, and 6.1kB of data memory. This is an 8-bit AVR device, running with an internal CPU clock of 20 MHz. These specifications are neither bottom nor top of the line, and should therefore provide a satisfying reference for the capabilities of neural networks in 8-bit MCUs. An example of a more powerful device well suited for ANN applications is the AVR128DB64, with 128kB of program memory, and 16.4kB of data memory. This device will not be evaluated in this thesis, but serves as a device recommendation for the reader.

3.2 Choosing Neural Network Frameworks

As mentioned in section 1.2, the reason why ANNs is a promising machine learning approach for MCUs, is that the networks require relatively little processing power to run once the network training is complete. The solution is therefore to train the networks on a powerful PC, before transferring it to a resource constrained MCU. To accomplish this one needs both a training framework for the PC, and an inference framework for the MCU. The following section presents the selection process of these frameworks.

3.2.1 Framework for Model Training

There are a few key criteria when choosing a neural network framework to use for MCU model training. An important selling point to those who are new to machine learning and ANNs, is an easy to use API. In the same spirit of usability, the framework should be well documented. Different optimization features, such as quantization and pruning, should be well implemented, to facilitate for maximum resource utilization on the MCU. The two most popular frameworks used to date are Tensorflow and PyTorch, both of which are mainly implemented as Python libraries. Both libraries are well documented, and support pruning and quantization. Tensorflow is actively developing a sub-package called Tensorflow Lite, which is aimed towards mobile applications. Smart phones and other mobile systems are in context regarded as resource constrained devices, and are frequently implemented with 32-bit processors. Tensorflow Lite was attempted implemented as an inference framework on the *ATmega4809*, but proved to be incompatible (see section 3.2.2). Tensorflow Lite does however still provide a state of the art toolkit for reducing the size of ANN models. The compression factors of Tensorflow Lite and PyTorch have not been directly compared.
Tensorflow is widely regarded as a less intuitive library than PyTorch, however, as of 2019, Tensorflow natively supports the Keras API. Keras is a high level API, which has long been praised for its simplicity, and its usability for quickly prototyping neural network models. Considering Tensorflow both supports Keras, and provides a toolkit as extensive as Tensorflow Lite, one could conclude that Tensorflow is slightly more suited for the application in question. Another factor to consider, however, is that Tensorflow and PyTorch save models in different file formats. When choosing an inference framework for the MCU (see section 3.2.2), one might discover that certain libraries only support one or the other format. The Tensorflow Lite sub-package also defines its own format, which is not necessarily compatible with inference frameworks listed as Tensorflow compatible. This could thus be a decisive factor in both training and inference framework selections.

3.2.2 Framework for MCU Inference

Once a model has been trained, it can be ported to a MCU using an appropriate inference framework. Such a framework refers to a code base written to optimize for size, using uint8 types wherever possible, instead of float32. A few companies, like Cartesiam, offer this forward-pass[1] as a paid service, but do not necessarily grant access to the framework source code. Such paid services will therefore not be considered. Implementing an inference framework from scratch was considered, but this was deemed too time consuming for the limited time frame of this thesis project.

The open source market for such frameworks is thinly populated, where most solutions are underdeveloped and unmaintained. One exception to this is the Neural Network on Microcontroller (NNoM) library [21] written by Jianjia Ma. This is an open source project that performs forward-pass for MCUs. It is designed to work seamlessly with Tensorflow, allowing the user to convert a model from the Tensorflow format to an MCU friendly format by the call of a single function. It supports a wide variety of functionality, including different layers types, activation functions and loss functions. NNoM is by no means a perfect library. The documentation is of poor quality, and the provided application examples are outdated, and are no longer functional. There are a number of aspects to consider when designing a model in Tensorflow, to make it compatible with NNoM, which are either poorly documented, or not documented at all. These aspects are detailed in section 3.3. The library does not seem to be in active development, based on the commit frequency on GitHub. There is therefore a risk that the library might break with future Tensorflow releases , without NNoM being maintained for continued support. As mentioned, NNoM supports the Tensorflow format, but it does not support Tensorflow Lite models. This is turn leads to NNoM not being able to utilize the optimization tools provided by Tensorflow Lite. NNoM does however have built-in quantization functionality, so this is no large issue.

Another promising framework is one developed by Fraunhofer IMS, called Artificial Intelligence for Embedded Systems (AIfES). As of writing this thesis the library is unreleased, but is scheduled for open source release sometime during the summer of 2021. Some details of the library has been gathered through conversation with the library author. At release, the library will support the dense layer type, with convolutional layers being scheduled for a later date. Similarly to NNoM, it will support the Tensorflow model format, but it is unknown if it will support Tensorflow Lite. It is said to support quantization, but no more details are know about this. [22]

Tensorflow Lite was attempted implemented as an inference framework on the MCU. The official documentation does state that the framework requires a 32-bit processor [23], so these attempts were, unsurprisingly, unsuccessful. The reason for this incompatibility is not stated.

[1]Forward-pass refers to the operation of porting a pre-trained neural network to a resource constrained device, such as a MCU.

Inference Library Discussion

As of writing, the only suitable MCU inference framework the author could find was NNoM. It is important to note that this library was likely developed with 32-bit MCUs in mind, considering the lack of published work for 8-bit MCU ANNs. The library documentation does not mention which MCU architecture it was developed for, which is likely a deliberate decision, as to not restrict the library to one or the other. The reason why very few alternatives to NNoM exists, might be the large appeal and popularity of Tensorflow Lite. Up until now there has been no need for competing ANN inference frameworks, due to the existence and consistent development of Tensorflow Lite. As described above, Tensorflow Lite proved less than trivial to implement on the *ATmega4809*, and is therefore, as of writing, not a contender for the 8-bit implementations. AIfES will hopefully prove to be a suitable choice on release, but this remains to be seen.
Although NNoM has flaws, it is still more than capable of providing a proof of concept, and to serve as a reference point when determining the feasibility of ANNs on 8-bit MCUs. NNoM will therefore be used for ANN implementations throughout this thesis.

3.3 Implementing a Simple ANN Using the NNoM Library

This chapter provides an introduction to using the NNoM library, to implement ANNs on MCUs. This includes special design considerations one needs to make during the network structuring and training phase, as well as how to generate the MCU model, and ultimately perform inference on the MCU. This will be presented through a simple example, as to not complicate the procedure with unnecessary network complexity. The example simulates a simple security system, designed to detect movement. The imagined movement sensor has a tendency to occasionally report false positives, and false negatives. To remedy this, the system is fitted with two such sensors, placed in close proximity. The system will only report a "movement detected" if both sensors are triggered. Such a task could naturally be solved using a logical AND operation, but for the sake of this example it will be solved using a densely connected neural network.

Generating Dataset

In order to train a neural network, one needs a labeled dataset for training, containing representative input data, and the desired output data. Listing 3.1 shows how the training dataset for this example was generated. The Python script generates 10 000 training examples, of which 5% indicate "movement detected", and the remaining 95% indicate no movement. The test data was generated in the same fashion, with the same label distribution, although only containing 1000 test examples. After the data is generated, the Python lists are converted to Numpy arrays, which is a requirement from Tensorflow. Finally the data is shuffled, to ensure the different label occurrences are evenly distributed in the dataset. The value range of the dataset is set to $[-64, 64]$, due to design considerations that will be presented in chapter 5.

```python
import numpy as np
from sklearn.utils import shuffle

#------------- Training data -------------#
train_samples = []
train_labels = []

# Movement detected
for i in range(500):
    train_samples.append([64,64])
    train_labels.append(1)

# No movement detected
for i in range(500):
    train_samples.append([64, -64])
    train_samples.append([-64, 64])
    train_labels.append(0)
    train_labels.append(0)
for i in range(9000):
    train_samples.append([-64,-64])
    train_labels.append(0)

train_samples = np.array(train_samples)
train_labels = np.array(train_labels)

train_samples, train_labels = shuffle(train_samples, train_labels)
```

Code listing 3.1: Generating dataset for the Sensor System example. Test data is generated using the same code, but with the sample amount reduced to a total of 1000 samples.

3.3.1 Defining and Training ANN Model Using Tensorflow Keras

Listing 3.3 shows the definition of a densely connected network, using Python and the Keras API. The example uses the Sequential model type, due to its simplicity and readability. Both the Sequential and Functional model types have been verified to work with NNoM. The topology of the defined network has been visualized in figure 3.1. When defining a network for NNoM, it is important that every layer and activation function is declared explicitly. It is very common when training ANNs using Keras, to declare activation functions through an argument to the preceding layer, as shown in listing 3.2. NNoM will not recognize layers declared is such a manner, and the format in listing 3.3 must therefore be followed.
The output of the network can take one of two values, either "movement detected", or "no movement detected"; which implies that this is a binary classification problem. As presented in section 2.4, the activation function to use on the output layer for binary classification problems, is *Sigmoid*. Similarly, as presented

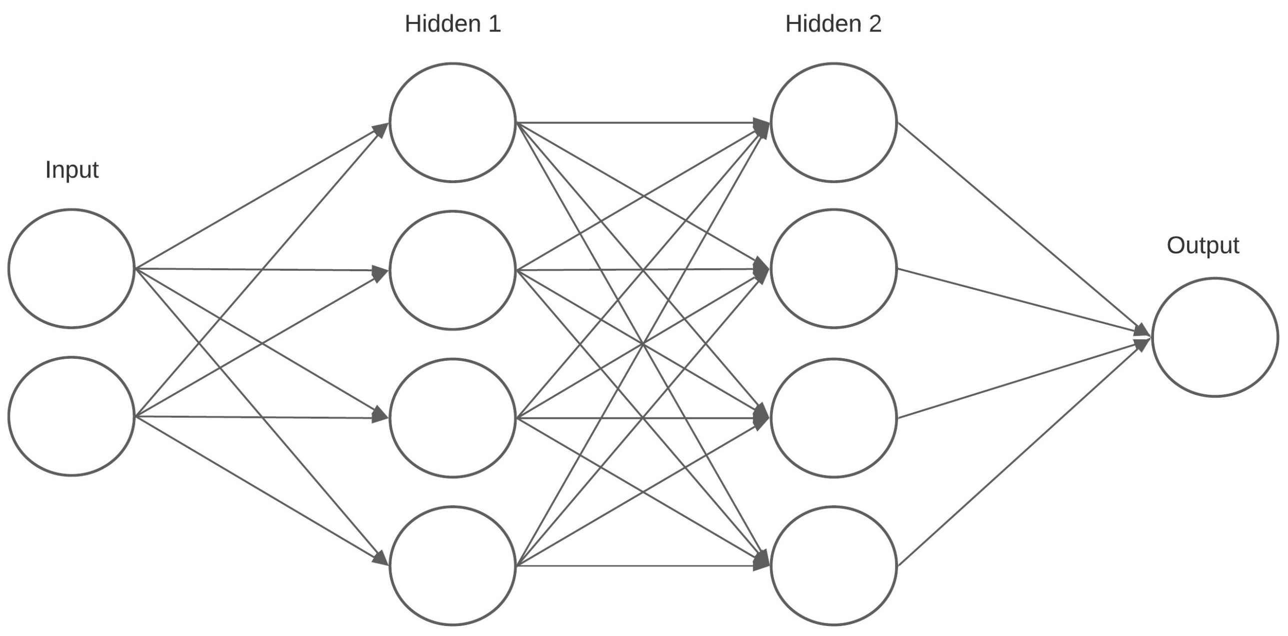


Figure 3.1: Topology of network defined for the Sensor System example.

in section 2.5.2, the loss function to use for such problems, is BCE. The methods *model.compile* and *model.fit* applies the loss function and remaining training parameters, and the training is carried out through *model.fit*.

```python
Dense(units=4, activation='relu')
```

Code listing 3.2: Example of incompatible activation function declaration.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense, InputLayer
from tensorflow.keras.optimizers import Adam

model = Sequential([
    InputLayer(input_shape=(2,)),
    Dense(units=4),
    Activation('relu'),
    Dense(units=4),
    Activation('relu'),
    Dense(units=1),
    Activation('sigmoid')
])

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',
     metrics=['accuracy'])
model.fit(x=train_samples, y=train_labels, validation_split=0.1, batch_size=64,
     epochs=10, shuffle=True, verbose=2)
```

Code listing 3.3: Defining ANN model for Sensor System example.

3.3.2 Converting Model for MCU Using NNoM

The NNoM library is divided in two parts; one part implemented in Python, and one in standard C. The Python package handles the conversion from Tensorflow

model to an NNoM model, by generating a *.h* header file containing the network structure, weights and biases. The C library serves as the inference framework on the MCU. Listing 3.4 shows the function call for generating the NNoM model. It takes the Tensorflow model object as its first argument, and as its second argument it takes a set of representative data. The latter refers to a subset of the data used to either train or test the network, which is representative of data the network could expect as input. The NNoM documentation claims 1000 training examples to be optimal [24]. The last argument is an optional save path to store the generated header file. This is typically a path to the MCU project folder.
A limiting factor one might encounter during this implementation step, is the limitations of the built-in quantization functionality in NNoM. The library does convert all weights and variables in the model to int8 types, but does not ensure that node output values will not exceed the limits of 8-bit resolution. This means that the network architect has to manually ensure values in the network will never exceed the range of $[-127, 127]$. If a node output exceeds this range, the int8 variables will overflow. The *generate_model* function in listing 3.4 provides useful console printouts to monitor exactly this, which displays the maximum and minimum output values for each layer. Using this information, one can remedy the problem through a number of design choices, some of which will be presented in section 5.4.

```python
from nnom.scripts import nnom_utils

nnom_utils.generate_model(model, test_samples[:1000], mcu_path, quantize_method='
    max_min')
```

Code listing 3.4: Function call for converting Tensorflow Keras models into an MCU friendly format.

3.3.3 Running Model on MCU

Once the NNoM ANN model is created, it can be implemented on the MCU through the NNoM C API. This is done by including the NNoM folder in the desired MCU project, specifically the subfolders *inc*, *port*, and *src*. The includes in the library are referenced from the root directory, so the *inc* and *port* folders must be manually added to the project include paths. As mentioned in section 3.3.2, the neural network model generated by NNoM is defined within a *.h* header file. The ANN model can be defined on the MCU by including this header file, and executing the embedded *nnom_model_create* function. Listing 3.5 shows how to define the model, enter input data, run the model, and finally read out the resulting model prediction. Due to the simplicity of the example problem, this model performs at an accuracy level of 100%. This performance level is much harder to achieve in more complex models.

```c
#include <avr/io.h>
#include "weights.h"
#include "nnom.h"

int main(void)
{
  nnom_model_t  model;
  model = nnom_model_create();

  nnom_input_data[0] = 64;
  nnom_input_data[1] = -64;
  model_run(model);

  uint8_t result = nnom_output_data[0];

  while(1);
}
```

Code listing 3.5: Example C code for running inference on MCU.

Chapter 4

Testing Using MNIST Classification Models

In order to test the feasibility of ANNs on MCUs, the MNIST dataset of handwritten digits will be implemented. This dataset is commonly referred to as the "Hello World" of machine learning, and is often used as an example application when showcasing certain machine learning methods. Several tests were performed using this application as a reference point, which will be presented in this chapter.

4.1 Implementing MNIST Handwritten Digit Classifier

The MNIST dataset [25] contains a total of 60 000 images, all depicting a handwritten digit. The images are comprised of a 28x28 pixel grid, containing only a single color channel (grayscale). The objective of the model in question will be to recognize which digit is written in each image. All models were trained using Tensorflow Keras, and forward-pass to MCUs were performed using NNoM. This process is outlined in section 3.3, and will not be detailed in the following sections.

Densely Connected Network

The densely connected networks trained during this chapter vary in structure and size. What they do share, however, is the output layer, and the training parameters. Each MNIST model has an output layer with ten nodes, one for each digit. This indicates that the problem is a multiclass classification problem. As presented in section 2.4, this means the output activation function should be *softmax*. The loss function has been similarly set according to the problem type, to *SCCE* (see section 2.5.2).
The raw MNIST data is formatted as a two dimensional array, with a shape of 28x28 pixels. Dense layers cannot handle multi-dimensional data, and so the data needs to be "flattened" before training. This refers to converting the two-dimensional array, to a one-dimensional array, by appending each row of the original array to a Python list. Eventually this will result in a list of 784 elements.

Listing 4.1 shows the function call of *load_flat_images*, which loads a preflattened dataset. An alternative to this is to use the *Flatten* layer available in Keras. The numerical pixel values are originally in the range $[0, 255]$, and are therefore subtracted 127, in order to fit the data within int8 type variables (NNoM requirement). The data is then divided by eight, to avoid integer overflow in the model. The reasoning behind the last step will be explored further in section 5.4.

```python
import numpy as np
from utils import load_label_data, load_flat_images

#----------------- Load datasets ----------------------#
train_data = np.array(load_flat_images('training'))
train_data = ((train_data) - 127)/8
train_data = train_data.reshape(50000, 784, 1)
train_labels = np.array(load_label_data('training', 50000, 0))

valid_data = np.array(load_flat_images('valid'))
valid_data = ((valid_data - 127))/8
valid_data = valid_data.reshape(10000, 784, 1)
valid_labels = np.array(load_label_data('training', 10000, 50000))

test_data = np.array(load_flat_images('test'))
test_data = ((test_data - 127))/8
test_data = test_data.reshape(1000, 784, 1)
test_labels = np.array(load_label_data('test', 1000, 0))
#------------------------------------------------------#
```

Code listing 4.1: Loading and preprocessing MNIST data for densely connected networks.

4.2 Testing and Results

4.2.1 Testing Framework

A framework for testing the MCU implementations was developed, in order to efficiently asses the accuracy and performance of the different ANN models. The framework consists of two components; one running on the PC, and one running on the MCU. Each image in the MNIST dataset consists of 784 bytes. In order to reliably test the accuracy of ANN models on the MCU, one needs to run inference on a few hundred such images. The ATmega4809 has a maximum program memory of 48kB, which would quickly be exhausted should the MNIST images be stored directly on the MCU. This has been solved by opening a communication channel between the PC and the MCU, utilizing the USART protocol. The PC transfers MNIST image data to the MCU, and the MCU returns the model prediction after running inference on the received data. In addition to model prediction, the MCU will return the execution time of the inference routine. This is done by using the Timer0 module on the ATmega4809. A timestamp is recorded before starting the model inference routine, and another timestamp is recorded after finished execution. The total execution time is thus calculated as the difference between the two timestamps. Software drivers for the USART and Timer module were generated using Atmel Start; a code generation tool for SAM and AVR devices. The PC Python implementation uses the PySerial package [26] to perform perform the serial communication. The Python code can be found in Appendix A, and the MCU C code can be found in Appendix B. In addition to the code designed for testing ANN models on the MCU, a number of purpose made scripts were developed to test certain specific metrics (e.g. pruning efficiency). These scripts will be referenced as their respective test results are presented later in this chapter.

4.2.2 Dense Keras Models - Size and Accuracy

The results in figure 4.1 and figure 4.2 were obtained by training a number of densely connected networks, each with similar network parameters, only varying the number of nodes in the hidden layers. This was done twice; once with two hidden layers (figure 4.1), and once with three hidden layers (figure 4.2). The networks are trained to predict handwritten digits from the MNIST dataset. The common training parameters for all networks can be seen in table 4.1. The network structure can be seen in table 4.2. Note that the accuracy is measured using the raw Keras model, and does not necessarily reflect the accuracy each respective model would achieve on the MCU. The diameter of each data point reflects the relative size of the models, so that a large circle represents a large model, and a small circle represents a small model. The Python code used to generate these results can be seen in Appendix C.

Model Type	Sequential
Epochs	10
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Loss Function	Sparse Categorical Crossentropy
N Training Examples	50 000
N Validation Examples	10 000
N Test Examples	1000

Table 4.1: Training parameters for networks in figure 4.1 and figure 4.2.

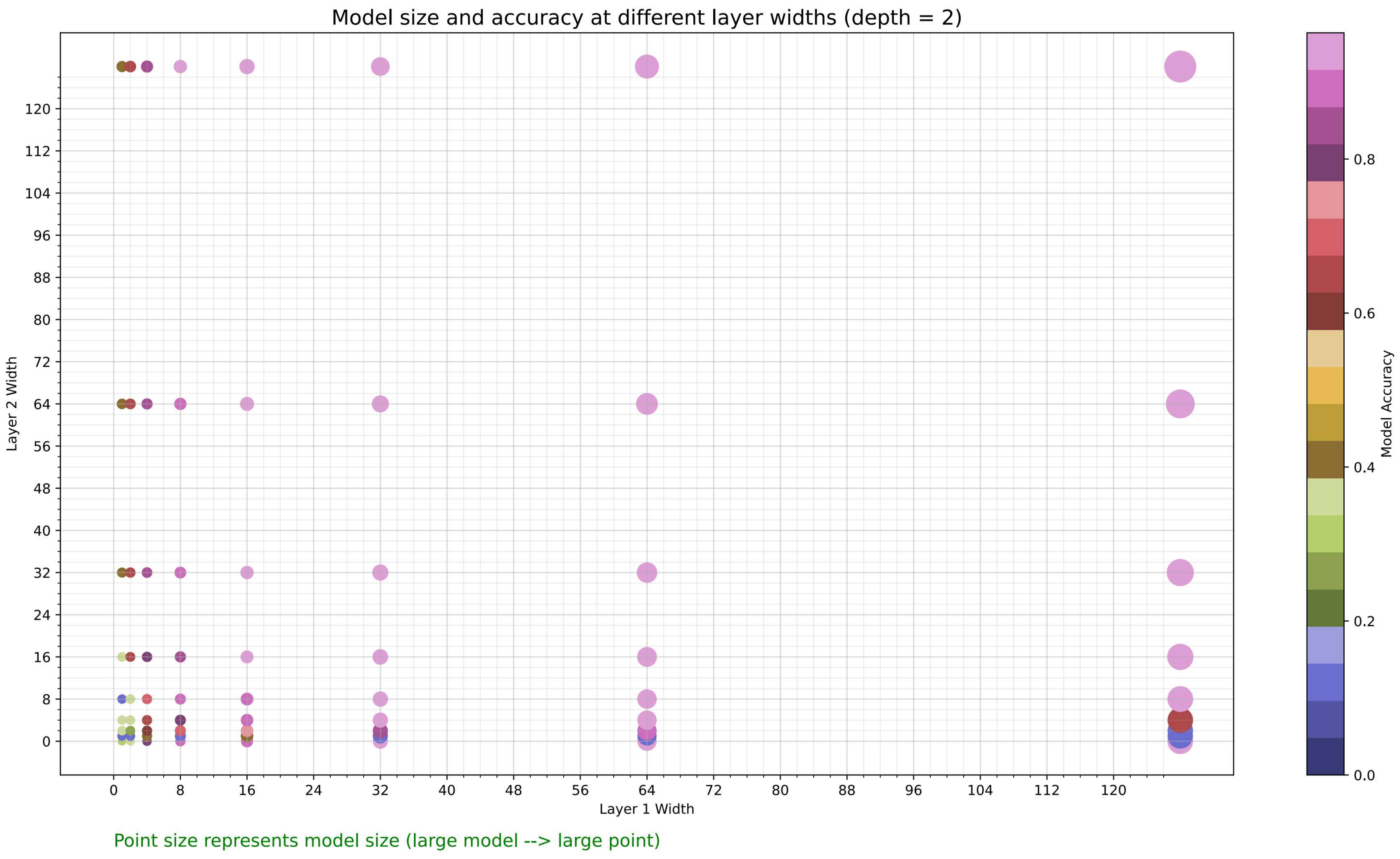


Figure 4.1: Size and accuracy of densely connected networks, with two hidden layers and varying layer widths.

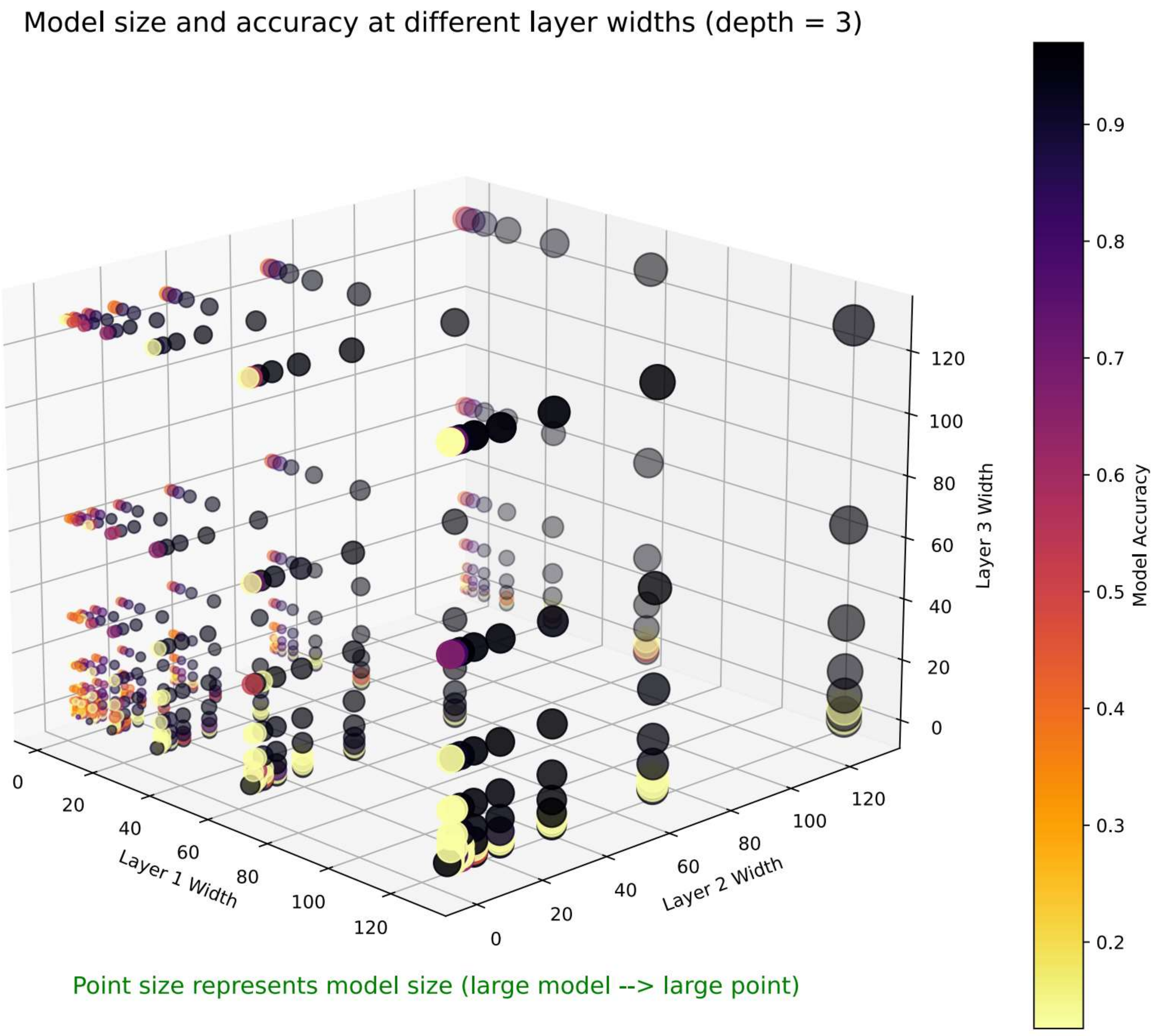


Figure 4.2: Size and accuracy of densely connected networks, with three hidden layers and varying layer widths.

InputLayer(input_shape=(28,28,1))
MaxPool2D(pool_size=(2,2), strides=2)
Flatten()
Dense(units=width_layer_1)
Activation('relu')
Dense(units=width_layer_2)
Activation('relu')
Dense(units=10)
Softmax()

Table 4.2: Network Structure for figure 4.1. Figure 4.2 has a similar structure, but with an additional hidden layer.

4.2.3 MCU Network Accuracy

Figure 4.3 displays the relative accuracy achieved on the MCU, compared to the accuracy achieved from the unoptimized Keras model on the PC. The MCU model is quantized, but not pruned. The relative accuracy metric is calculated from the following equation, where A is the estimated model accuracy:

$$A_{MCU\_Relative} = \frac{A_{MCU}}{A_{Keras}}$$

Each data point was produced by training a MNIST handwritten digit classifying model using Tensorflow Keras, and then using the NNOM library to convert said model into an MCU format. A_{MCU} is the estimated accuracy achieved on the MCU, after performing inference on 300 test examples. A_{Keras} is the estimated accuracy achieved on the PC, after performing inference on 1000 test examples. The relative MCU accuracy plotted in figure 4.3 is produced by applying the above equation to each trained model. Each network shares the same training parameters and structure, which can be seen in table 4.3 and table 4.4 respectively. The only varying factor is the number of nodes in the single hidden layer, which are in the range $[4, 32]$, with an increment of one. The test was run twice, with the blue curve representing the results from the first run, and the orange curve representing the second run.

4.2.4 Compiler Optimizer Settings

Code for the ATmega4809 was compiled using the AVR GCC compiler, which provides a set of different code optimization routines. Model size and inference execution time was measured, after compiling and uploading the ANN solutions to the MCU with different optimization settings. The optimization settings are invoked by the following compiler flags: [*-O1*, *-O2*, *-O3*, *-Os*]. Three different ANNs were used in the test, all with a single hidden layer, and with similar training parameters. The only varying network parameter is the width of the hidden layer. Flash usage was measured using a Python tool called *pymcuprog*, and inference

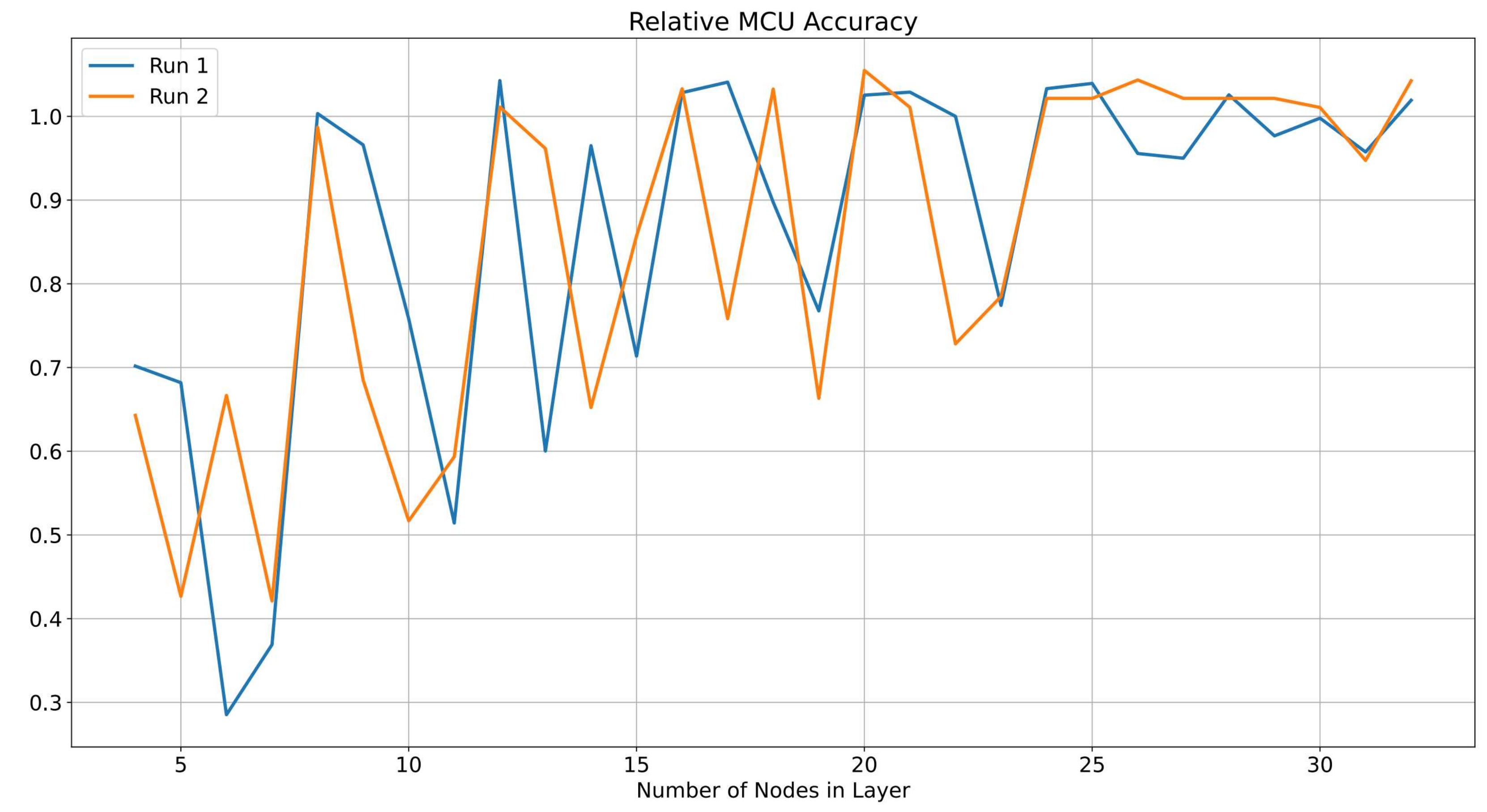


Figure 4.3: The relative accuracy achieved on the MCU, compared to the accuracy achieved from the raw Keras model on the PC.

Model Type	Sequential
Epochs	10
Optimizer	Adam
Learning Rate	0.001
Batch Size	64
Loss Function	Sparse Categorical Crossentropy
N Training Examples	50 000
N Validation Examples	10 000

Table 4.3: Training parameters for networks in figure 4.3.

InputLayer(input_shape=(784))
Activation('tanh')
Dense(units=Hidden_layer_width)
Activation('relu')
Dense(units=10)
Softmax()

Table 4.4: Network Structure for figure 4.3.

time was measured using the ATmega4809 on-board Timer0 module. Table 4.5 displays the test results.

4.2.5 Pruning Impact on Accuracy and Compression

Tests were performed to investigate the accuracy loss, and compression gain, that follows a pruning operation. The tests were performed on both wide and deep networks, without any significant variation in results. The pruning routine used followed an example presented by Tensorflow [27]. The pruning parameters "initial_sparsity" and "final_sparsity" were varied incrementally, and the model accuracy and compression gain were recorded for each increment. Figure 4.4 and 4.6 show the results of the tests. The compression factor is calculated by dividing the initial model file size, by the resulting model size after finished optimization. Note that no attempts were made to optimize the initial model accuracy, and that the results must be interpreted relatively. It is also noteworthy that no accuracy values were estimated using an MCU, and that the accuracy potentially achieved on an MCU might differ. The code used to generate the test results can be seen in Appendix D.

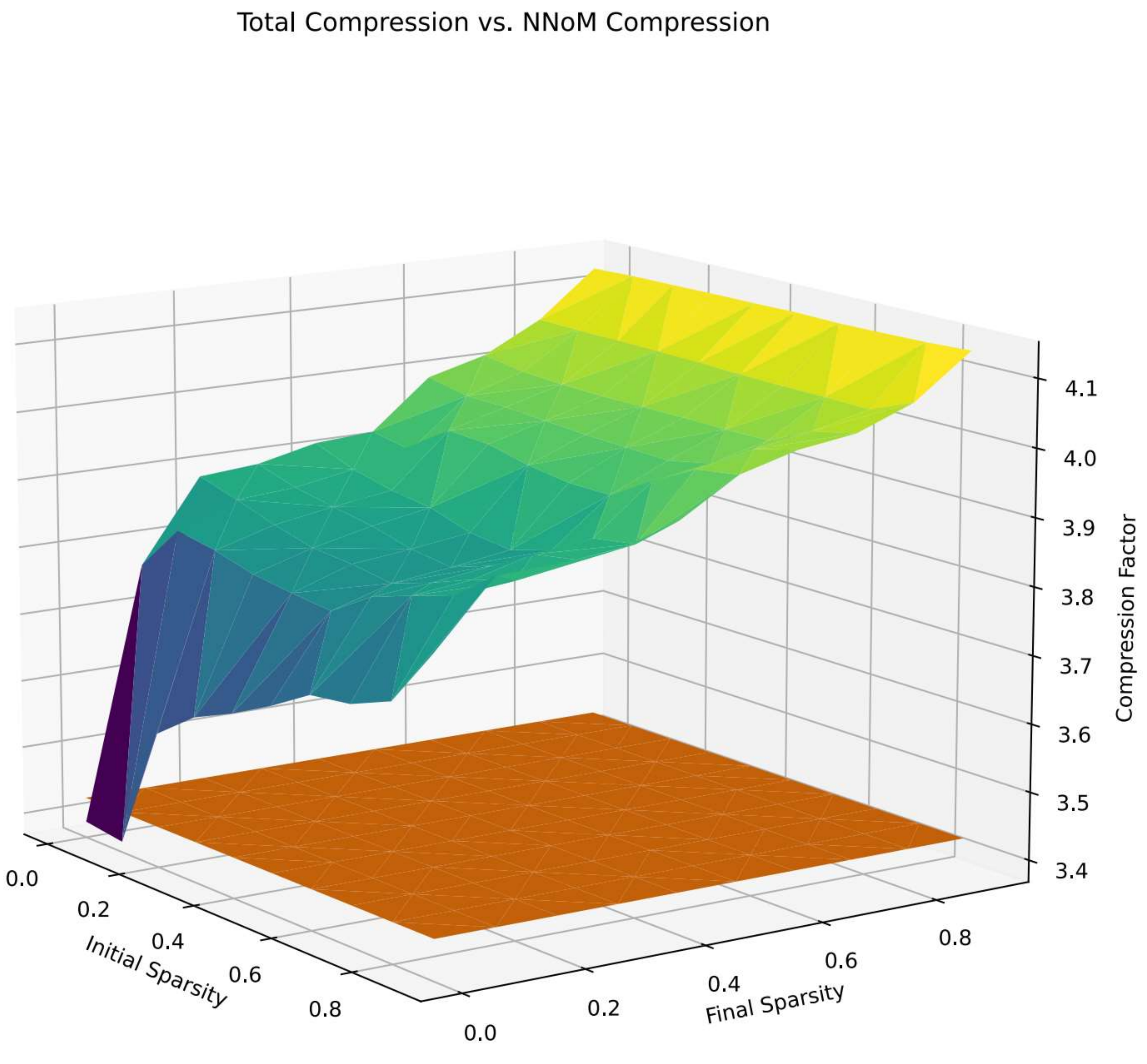


Figure 4.4: The orange plane represents the compression achieved from the NNoM conversion alone. The color varied surface plot represents the total compression achieved by both pruning the model, and performing the NNoM conversion. Initial and final sparsity (X and Y-axes) refer to pruning settings.

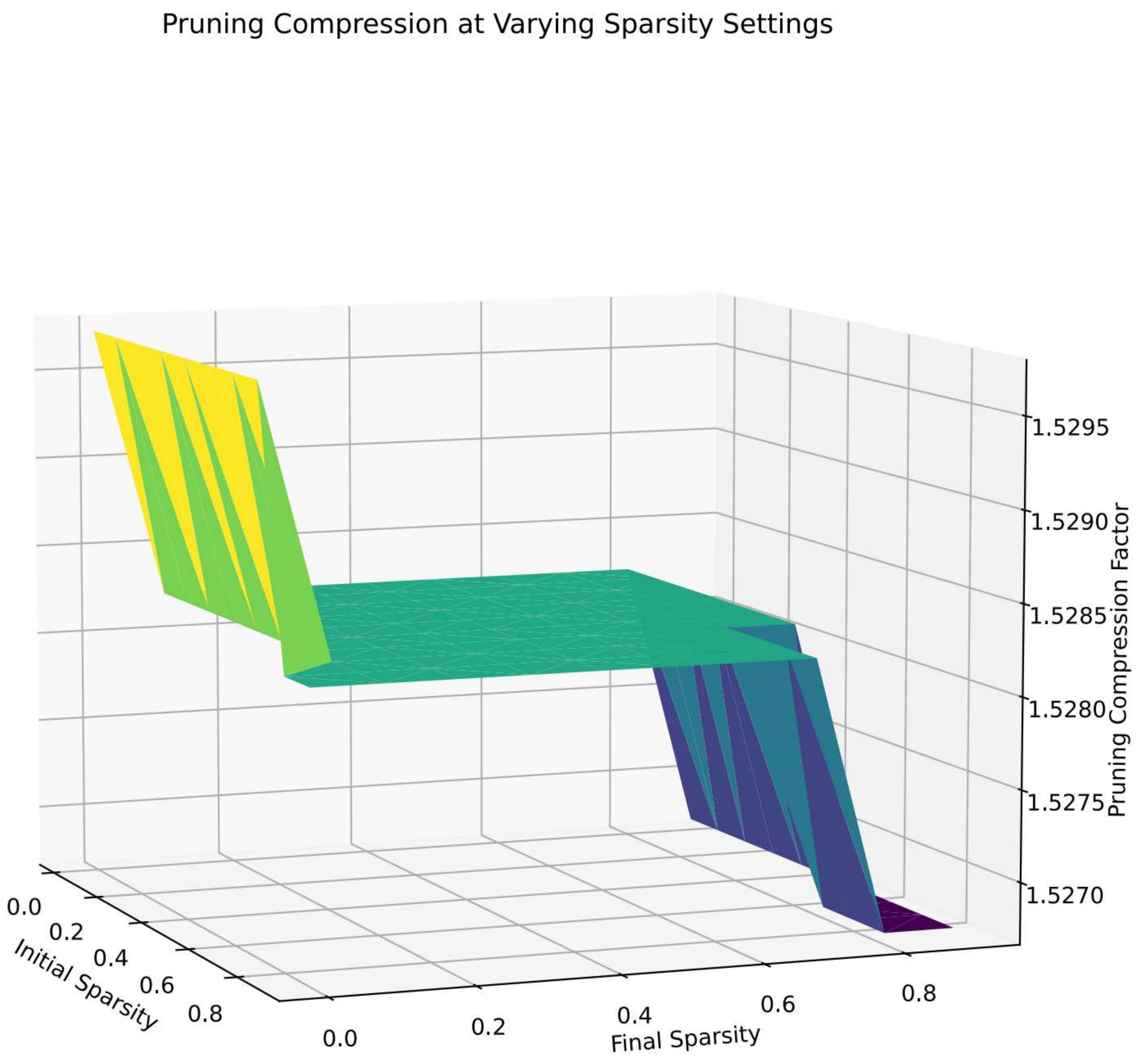


Figure 4.5: This surface plot represents the compression factor achieved by applying pruning to an MNIST classification model. The sparsity settings of the pruning operation are varied along the X and Y-axes.

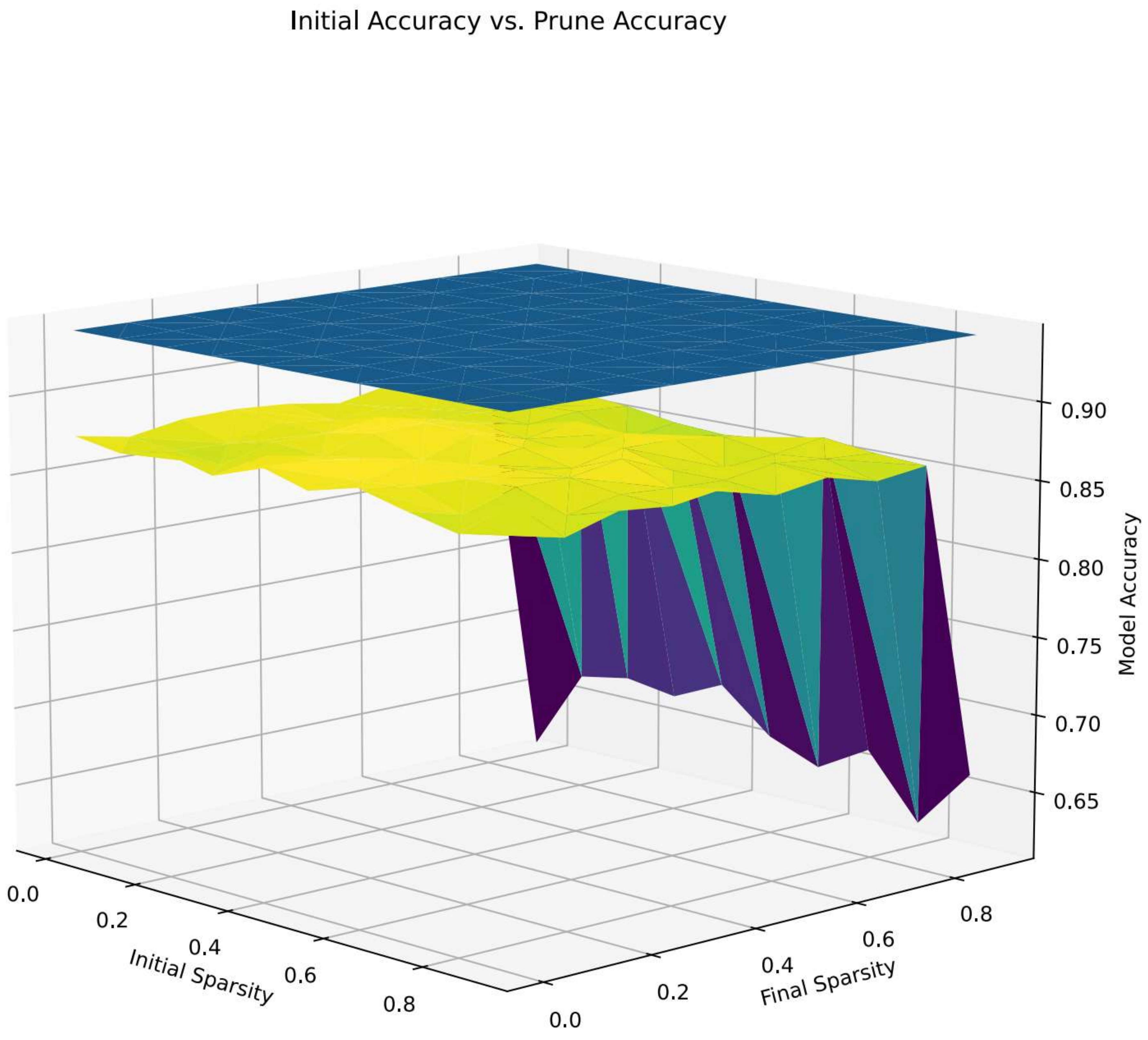


Figure 4.6: The blue plane represents the initial model accuracy achieved before performing any optimization routine. The color varied surface plot represents the accuracy achieved after the model has been pruned. Initial and final sparsity (X and Y-axes) refer to pruning settings.

Layer Width = 8	O1	O2	O3	Os
Inference Time [ms]	9.06	8.04	8.09	7.01
Program Memory Usage [B]	23120	22976	25152	21984

Layer Width = 32	O1	O2	O3	Os
Inference Time [ms]	31.44	27.39	27.44	23.14
Program Memory Usage [B]	42192	42064	44240	41072

Layer Width = 40	O1	O2	O3	Os
Inference Time [ms]	38.91	33.89	-	28.51
Program Memory Usage [B]	48560	48416	-	47424

Table 4.5: Inference time and flash usage statistics for different GCC compiler optimization settings.

Chapter 5

Optimization and Design Considerations Discussion

There are several different optimization techniques and design considerations one can deploy to utilize as much of the MCU resources as possible, with as few trade-offs as possible. These did however prove less than trivial to implement. Several issues related to this arose during the coarse of writing this thesis. The issues were addressed, and guidelines on how to work around said issues were developed. The following chapter will present and discuss these guidelines, as well as the different optimization techniques and design considerations. The performance of different techniques will be discussed in reference to the results presented in chapter 4.

5.1 Quantization

Quantization is an absolutely necessary optimization technique for ANNs on 8-bit devices. This technique converts all variables in the desired model from float32 types, to int8 or uint8 types. Tensorflow normally performs quantization through the Tensorflow Lite subpackage, but this functionality also exists within the *Tensorflow Model Optimization* package, commonly referred to as *tfmot*.
Tensorflow Lite has the drawback of using its own *tflite* file format. NNoM does not support the *tflite* format, and so this quantization functionality cannot be used with said inference framework. The tfmot package does not work with NNoM either. When tfmot quantizes a model, it prepends the layer names with "quant_" , and seemingly alters other structure names in the quantization process. NNoM relies heavily on the layer names for identification, and will therefore not be able to process such models. Attempts were made to revert these altered identifiers, but the efforts did not results in a successful NNoM compilation. NNoM does however provide a quantization routine itself, which works as advertised. This is both a positive and a negative aspect of NNoM. This built-in quantization routine simplifies the implementation process, and abstracts away the quantization operation to such a degree, that the user can practically disregard this step. It does how-

ever remove the customizability from the quantization. As mentioned in section 2.7, one can either quantize a model during training, called quantization aware training; or after the training is finished, called post-training quantization. Due to the NNoM incompatibility issues, only post-training quantization is implementable. This can be a significant drawback, as quantization aware training has been shown to produce more accurate models than post-training quantization [28].
There is no apparent technical reason why the optimization functionality in Tensorflow Lite should be incompatible with 8-bit MCUs, this is simply an issue with NNoM. It is therefore useful to note that other frameworks, such as AIfES, might support *tflite* models. It is hard to determine whether or not Tensorflow Lite would improve model compression significantly, however, the compression achieved using the built-in quantization routine does reduce the model size by a factor greater than three. This result is achieved consistently by training a model using Keras, then converting said model using NNoM, and finally comparing the file size of the keras *.h5* file, and the NNoM *.h* file.

5.2 Pruning

Figures 4.4, 4.5, and 4.6 show results that explore the benefits potentially achieved through pruning. It is apparent from figure 4.4 that one can gain a significant improvement to the model compression by performing a pruning operation. Figure 4.6 does however showcase the cost of such a compression gain. There is a seemingly constant accuracy drop, regardless of the sparsity settings used. In the case of this particular MNIST model, the post pruning accuracy drop is at 5%. This test was performed on several different networks, both wide and deep, and each test showed similar results. It is worth noting that the test was only performed on MNIST classification models, and that these metrics might show different results for datasets with smaller input data.
Another interesting result is that seen in figure 4.5. This plot shows that the compression factor achieved from pruning alone is, in a practical sense, constant. By inspecting the Z-axis of the graph one can see that the increments are practically negligible. The sparsity settings are seemingly irrelevant for the model compression factor. By closer inspection of figure 4.6 it seems like the same can be said for the model accuracy drop. It is therefore interesting to see the significant impact sparsity settings has on the results in figure 4.4. The significance of the sparsity settings only become apparent when further compressing the model through the NNoM quantization routine. Due to the quantization functionality being embedded in NNoM, and the fact that NNoM defines its output model in a *.h* header file, the pruning had to be performed before the quantization. The dependency shown in figure 4.4 would likely not be present if the order of these operations were reversed, but this is not possible when using the NNoM library. Whether or not this dependency is reflected in the post-quantization model accuracy, is unknown. This was not tested due to difficulties in automating programming of the MCU, which would be necessary to test the accuracy of NNoM generated models.

Pruning might allow for larger models to fit on the MCU, but these large models would need to have a high initial accuracy to justify the accuracy drop following a pruning operation. These pruning metrics might however change for different network configurations that have not been tested here. There are many different variables that factor in when training such networks, any of whom might affect the outcome of these tests. More testing is therefore needed before concluding whether or not pruning is beneficial for ANNs on MCUs.

5.3 Network Topology and Memory Consumption

A central question in the MCU ANN feasibility debate is how deep, and how wide networks can be, before the MCU memories are saturated. The short answer is that, it depends on the size of the input data, the size of the network, and the size of the program and data memory on the MCU in question. The long answer requires a deeper look into which network aspects impact the network size. Figures 4.1 and 4.2 showcase the results from training several networks with varying depth and width, and then measuring the model size and accuracy of each. An interesting takeaway from these figures is that the model size increases significantly more when increasing the width of the first hidden layer (X-axis in figure 4.1), compared to increasing the width of subsequent layers. This is due to the size of the images in the MNIST dataset. As a result of using densely connected layers, each node in the first hidden layer will receive all 784 bytes of image data, which in turn will be reduced to a single output value from each node. Each of the input connections to the first hidden layer holds a weight. This means that increasing the layer width with a single node, means adding 784 weights to the network. In turn, the the number of weights associated with the input of the second hidden layer, will be dependent on the number of nodes in the first layer. For example; since each node only has one output, if the first layer has 32 nodes, then the input to each node in the next layer will carry 32 weights. This is significantly less than 784 weights per node, and as a result, for models with large input data (like MNIST), the first hidden layer will be the most significant layer in terms of model size. This will mostly affect the program memory of the MCU. The data memory seems to be affected too, but attempts at measuring this proved to be unsuccessful. The data memory limitations have proved to be a bottleneck when loading large networks, but further details of these limitations have not been identified successfully.

For an MNIST model with a single hidden layer, and unaltered input data size, the ATmega4809 has been shown to fit networks with up to 40 nodes in the hidden layer, before running out of program memory. Subsequent layers can however be much wider, since the memory cost per node is lower. The memory cost of these layers can be roughly calculated using the logic presented above. The program memory increases in an expected manner when creating deeper networks, and it seems that the limiting depth factor is data memory. However, as stated before, more testing is needed before concluding with any statements about data memory.

5.4 Avoiding Integer Overflows

One of the more limiting factors when running ANNs on MCUs is the resolution available in 8-bit integer types. A large portion of this problem is handled by quantizing the model before deployment, but the quantization routine does not account for the output values of individual nodes. Even though each weight in the model is scaled to fit inside 8-bit integers, there is no guarantee that the node input function (equation 2.1) will output a value limited by the same range. This can potentially cause integer overflows, effectively making the nodes in question output vastly different values from what is intended. This could in turn cause the network to drop significantly in prediction accuracy. Luckily, in the case of NNoM, there are mechanisms in place to warn the user when this happens. NNoM provides useful printouts when converting the Keras model using the NNoM Python API. These printouts show the maximum and minimum values output from each layer in the model. Since the NNoM C-framework is implemented using int8 types, these layer outputs must be within the range of $[-127, 127]$.

5.4.1 Remapping Dataset Value Range

There are a few steps one can take, should a model exceed this 8-bit limit. One such step is to remap the values in the dataset to a smaller range. The MNIST dataset has values in the range $[0, 255]$, which at first glance seems to be fitting for 8-bit applications. However, this leaves no wiggle room for the internal model calculations, which means that any positive contribution from a node will cause an integer overflow. The first adjustment one should make is to center it around zero, with equal parts positive and negative values[1], which for the MNIST dataset would grant the range $[-127, 127]$. The next adjustment should be to downscale each data point by a constant factor. This factor depends on the original scale of the data range, and on the structure of the network being trained. Listing 4.1 shows how the data was scaled for the tests performed in chapter 4. For the case of the MNIST classification model, a scaling factor of 8 was deemed suitable. Choosing this scaling factor is a matter of balance. A larger scaling factor means a smaller numerical range. In other words, this leads to a loss in numerical resolution, which might negatively impact the network performance. In non-resource-constrained devices it is common to scale the training data between zero and one. In the case of systems using float32 types, this would provide great accuracy due to its ability to represent decimal places. This is not possible with integer types, so a range between zero and one would grant a resolution of exactly two values. The selection of this scaling value thus has to be balanced for each network, so that the numerical resolution is as large as possible, while still avoiding integer overflows. Some applications are more affected by a resolution loss than others. The MNIST classification model is minimally affected by the drop in resolution, but a fine-

[1]This step is specifically for NNoM, which uses int8 type variables, and is not relevant for frameworks implemented with uint8 types.

tuned sensor network might suffer significant consequences as a result. In such cases it is wise to retain as large a resolution as possible, and rather utilize the technique presented in section 5.4.2 to avoid integer overflows.

5.4.2 Selective Choice of Activation Functions

Another step to prevent integer overflows is to be selective of which activation functions are used in the hidden layers of the network. The ReLU activation function (see figure 2.2) will return a value equal to the input value, for all values greater than zero; and will return zero for all negative input values. This could potentially create a sort of cascading effect, where an initial large value contributes to a large node output, which then becomes a significant factor in the input function of the next node. This could lead to increasing output values for each successive layer, eventually causing an integer overflow. Tanh and Sigmoid on the other hand have output values in ranges $[-1, 1]$ and $[0, 1]$ respectively. This will prevent the aforementioned cascading effect, since node output values are restricted to an upper activation limit of magnitude one. Section 2.4 presents theory of which activation functions to choose for different use-cases. However, due to to the integer overflow challenge, these recommendations and best-practices might need to be set aside in order to account for this factor. When using ReLU on non-resource-constrained devices, it is recommended to scale the data points of the dataset to a range of $[0, 1]$ [6], which would also cancel the cascading effect. As discussed earlier however, this is not possible when using integer data types. This is an indication that ReLU might be unsuited for use on 8-bit MCUs. It is important to note that this cascading effect is in no way guaranteed to occur, and in several situations ReLU could be used without causing an integer overflow. Another note is that Tanh and Sigmoid produce values between -1 and 1, which are unsuited for use on 8-bit MCUs. The author has not been able to decipher how NNoM handles the non-integer nature of these activation functions. Through testing it has been confirmed that these activation functions work as expected when running on an MCU, and are therefore presented as a good and functional tool for preventing integer overflows.

5.5 Optimization Through Data Preprocessing

As previously discussed in section 5.3, it is clear that for networks with large input data (like MNIST), that the width of the first hidden layer is the most significant factor in the total model size. This effect can be reduced by downsampling the input data before feeding it to the network. This would reduce the number of data points for each training example, which in turn means fewer data points input to each node in the first hidden layer. This strategy is not necessarily applicable to all use-cases, and one has to ensure no vital data is filtered out. It can be very efficient for use in image recognition tasks, where removing every other pixel does not necessarily take away any crucial information from the image. The same can be said for audio recognition, where downsampling would have the effect of removing high-frequency information. This is a consideration the network architect needs to address before utilizing such a technique. E.g. in the case of a large sensor network, where each sensor carries vital information; then a downsampling operation would filter out irreplaceable data. This is in contrast to e.g. image recognition, where the data is only reduced in resolution. Tensorflow Keras does have built-in downsampling layers, called *pooling layers* (see section 2.6.2). A similar technique to downsampling, often used in image recognition, is to reduce the dimensionality of the image data. Images are normally structured with three separate color channels (e.g. red, green and blue), which all carry the same number of pixels. This effectively triples the necessary image size. Unless the application is dependent on color information, it is common to convert the image to an array of grayscale values. This retains the releavant image data, with color information filtered out, at a size reduction of factor three.

5.6 Compiler Optimizers

Table 4.5 presents the results of two different performance metrics, using four different compiler optimization settings. The first observation when reviewing the results is that, the choice of compiler optimizer could have a significant influence on both model inference time, and on the program memory usage. The one compiler setting that proves superior over the other three, based on the metrics at hand, is the *Os* setting. It shows both the lowest inference time, and the lowest program memory usage. Accuracy tests were also performed for each of the optimizer settings, which at no point showed signs of change. As a disclaimer it is worth noting that the author has very limited knowledge on the inner workings of compilers and its optimization algorithms. Different optimization settings might bring different side-effects and trade-offs, which are not reflected in the chosen metrics of this test. Such potential side-effects did not present themselves during testing, and the author has no reason to believe different optimizer choices will cause problems.

Chapter 6

Further Discussion

6.1 Alternative Embedded AI Solutions

Section 1.2.1 presented motivations for exploring the feasibility of 8-bit edge computed AI, where key arguments were privacy, latency, and power consumption. No tests performed throughout this thesis challenged the arguments of privacy or power consumption, so there is no basis to conclude whether or not the two hypotheses still stand. It is however very likely that they do stand, considering [1] concluded that edge computed AI does reduce the total power consumption of the MCU, and that data privacy is more secure. This work was done for 32-bit processors, but there is no reason to believe it does not hold true for 8-bit. Especially considering power consumption, and the fact that 8-bit MCUs are generally more power efficient than 32-bit devices. The argument of latency is however worth discussing more in detail. Table 4.5 presents the results of the GCC compiler optimizer tests, which feature both program memory usage, and model inference time. The latter gives a clear indication to how inference time changes with model width. Tests have also shown that the inference time increases very little with depth, and that once again, the size of the input data, and the width of the first hidden layer, are the most significant factors in deciding the magnitude of the inference time. Considering that the results in table 4.5 reflect the performance of an MNIST classification model, which is a resource intensive application; and that a layer width of 40[1] shows a maximum inference time of *38.9ms*; it is safe to assume that the inference time of a typical MCU application (using densely connected layers), with a clock speed of *20MHz*, is well below *50ms*. The question of whether or not this is an improvement to the latency achieved through cloud computed AI, is still difficult to answer. The latency of cloud computed AI varies vastly with a number of factors, such as the connection speed between MCU and cloud server; severity of interference in the area of operation; and the processing power of the cloud server. It is therefore hard to get an accurate estimate of this latency value. Something that is clear, however, is that the MCU inference time is

[1]40 is the maximum width possible with the current MCU, and the MNIST dataset. See section 5.3 for details.

very consistent. Edge computed AI is not reliant on networking or other external factors, and as a result it has an inference time almost deterministic in nature. This is highly advantageous for applications where a predictable inference time is important.

6.2 MCU Network Accuracy

The results presented in figure 4.3 attempts to demystify the behavior of the MCU model accuracy drop, when converting the Keras model to an MCU model. The test was initially designed to determine whether or not there is a predicable nature to the accuracy drop, but what can be observed is that the behavior is seemingly erratic. The test was performed twice, under the exact same conditions, with very different results. What becomes clear from these results is that, if the same network is trained several times, the resulting accuracy of the MCU models can vary vastly between each cycle. Another noteworthy observation is that several results show signs of non-existent accuracy drops. The results that do show signs of significant accuracy drops can, seemingly, be retrained to improve the relative accuracy score. It is also observable that the erratic behavior seems to diminish with increasing layer width. In summary, the MCU models can be implemented with close to no accuracy drop, so long as the network architect is aware of this behavior.
Whether or not this erratic trend is an issue directly linked to the inner workings of NNoM, is unknown. It is worth noting that some of the results in figure 4.3 indicate a relative accuracy above one, seemingly having gained accuracy in the NNoM conversion process. This is believed to be due to the accuracy tests on the MCU being performed on 300 test examples, and the Keras model being tested on 1000. This is a weakness in the test procedure that was discovered post-testing. The trend of the results are however not affected by this, and still remains valid.
In general, the accuracy of an ANN is decided by the design quality of the network. Throughout this thesis focus was not on achieving the best possible accuracy for a network, but rather optimize it as best possible. The accuracy estimates presented in this thesis should therefore be regarded in a relative fashion, and not be seen as an indication to an MCUs ability to solve the MNIST classification problem. There are several techniques that solve this problem in a much more impressive manner, some of which are discussed in section 6.3.

6.3 Additional NNoM Functionality

Throughout this thesis the main target of investigation has been densely connected networks. Although very useful, they can be limiting in certain applications. For instance, densely connected layers are not equipped to handle temporal data, but rather just a snapshot in time. It has no memory of previous predictions. Dense networks have proven to be well equipped to handle the MNIST classification problem, so recognizing single digits or letters are well within the realm of feasibility. The problem arises when faced with the task of recognizing full words, which would require the network to string together several individual character predictions. This would require a form of short-term memory in the network, and this is exactly what RNNs provide. NNoM does support recurrent functionality, but this has not been tested due to time constraints. If proved to be functional, this could open a door to much more complex tasks for the MCU. This includes tasks such as speech and sound recognition, temporal analysis of sensor data, and general time series predictions. Recurrent functionality is reportedly also planned for implementation in AIfES [22].

Although dense networks are able to handle image data, it does not mean it is the best suited layer type for the job. CNNs are widely known for their ability to generalize on image data. NNoM does have full support for convolutional layers. This functionality has been tested thoroughly, and works just as advertised. The MCU managed to achieve accuracy levels upwards of 99% for the MNIST classification problem, using CNNs. In the authors experience, CNNs require less program memory to run compared to dense networks, but require a greater amount of data memory. These layer types are much more computationally demanding during inference, due to the convolutional calculations (see section 2.6.1), which in turn means longer inference times. A rough estimate of a typical inference time is *300ms*. Image recognition is not a typical task for an 8-bit MCU, but it is clearly capable of handling the task, should a suitable use-case present itself. AIfES have reported CNNs as a planned future implementation[22]. In addition to RNNs and CNNs, NNoM does also support a variety of pooling layers.

The inclusion of this extra functionality further points to a positive feasibility report for ANNs on 8-bit MCUs. The RNN functionality has not been tested, and could potentially require a greater amount of data memory, or it could pose other unforeseen issues. If it proves to work as hoped, it would open a door to a new world of possibilities, with more complex functionality on the table. The inclusion of CNNs do speak in favor of feasibility, but less so due to the limited use-cases for image recognition in the 8-bit market.

Chapter 7

Conclusion

This thesis has explored the feasibility of running ANNs on 8-bit MCUs. The feasibility was evaluated through implementation of several MNIST handwritten digit classifying models, which was run on an ATmega4809 AVR MCU. As previously discussed, this device is neither a top of the line product for ANN applications, nor a low end product. The achieved performance on this device is therefore representative for the performance one expects on an 8-bit MCU. Solving the MNIST classification problem requires an input data length of 784 values. It is uncommon for 8-bit MCUs to handle applications with input data of such length, and it is therefore important to note that the MNIST classifier represents a computationally heavy task compared to a typical 8-bit MCU application.

The size of the input data, and the width of the first hidden layer, are the two major factors impacting the size of the final model. Reducing these factors are the most efficient model compression measures. Quantization reduces the model size by a factor of approximately 3, with seemingly no accuracy drop. This is a promising result, seeing as quantization is a vital step in transferring models to an 8-bit system. Pruning reduces the model size by a factor of approximately 1.5, but comes at the cost of a significant accuracy drop. As a result, the use of pruning might be hard to justify, seeing as other optimization techniques can grant better compression results at a smaller accuracy cost. Different compiler optimization settings (using AVR GCC) have also been shown to impact both program memory usage, and inference execution time.
Avoiding integer overflows has proved to be an important consideration when designing ANNs for 8-bit MCUs. This issue is a matter of balance between input data resolution, and the maximum/minimum output values from each layer in the model. In order to minimize layer outputs, one has to reduce the resolution of the input data. This issue can also be remedied by strategically selecting activation functions for layers with output values on the edge of the 8-bit scale.

Converting the pre-trained Tensorflow Keras models to MCU compatible NNoM models impacted the prediction accuracy in a seemingly erratic manner. In some cases the accuracy would drop by more than 40%; but it was shown that retraining the network, without altering any parameters, could remove this drop completely, granting a relative accuracy of 100%. In conclusion; as long as the network fits within the available MCU program and data memory, one can achieve the same model accuracy on the MCU, as on the non-resource-constrained device it was trained on. Model inference time was shown to be a function of mainly input data size, and width of the first hidden layer. Table 4.5 shows the effect of varying the latter. Network depth is also a factor in this regard, but with a relatively small impact. Based on the tests presented in this thesis, a typical inference time for a dense ANN running a MNIST classifier on an MCU, with a clock speed of *20MHz*, using the NNoM framework, is in the rough range of *1ms* to *50ms*. Considering that the MNIST application is, as discussed, computationally demanding, it is safe to say that no typical 8-bit MCU application will exceed an inference time of *50ms*[1]. It is difficult to generalize a conclusion for the achievable network depth and width, since these parameters are dependent on the MCU memory sizes, the application at hand, and the choice of layer types. The results and discussions in chapters 4 and 5 do however provide indications as to what is achievable.

It is clear that 8-bit MCUs are computationally capable of performing ANN inference. Other embedded ANN implementation methods, as presented in section 1.2, can provide better performance for different applications. 8-bit MCUs do however prove to be a contender for a range of use-cases. Key areas where 8-bit MCUs are strong contenders, are application that require predictable inference times, low power consumption, and data privacy. Such requirements rule out any cloud computed solutions. 32-bit MCUs will provide greater computing power, allowing for larger networks, and lower inference times. 8-bit MCUs are cheaper, consume less power, and provide low system complexity. The latter is therefore a superior alternative, if it satisfies the application requirements for memory size and inference time.

The results and implementations presented in this thesis strongly indicate that running ANNs on 8-bit MCUs is well within the realm of feasibility.

[1]This applies to the software and hardware setup stated, and would change for decreased clock speeds, and MCUs with larger memories running larger networks.

Bibliography

[1] M. Merenda, C. Porcaro **and** D. Iero, 'Edge machine learning for ai-enabled iot devices: A review,' *Sensors*, **jourvol** 20, **number** 9, 2020, ISSN: 1424-8220. DOI: `10.3390/s20092533`. **url**: `https://www.mdpi.com/1424-8220/20/9/2533`.

[2] A. Laudani, F. R. Fulginei, A. Salvini, G. M. Lozito **and** F. Mancilla-David, 'Implementation of a neural mppt algorithm on a low-cost 8-bit microcontroller,' **in** *2014 International Symposium on Power Electronics, Electrical Drives, Automation and Motion*, 2014, **pages** 977–981. DOI: `10.1109/SPEEDAM.2014.6872101`.

[3] S. Russel **and** P. Norvig, *Artificial Intelligence: A Modern Approach*. Upper Saddle River, New Jersey 07458: Pearson, 2010.

[4] M. Nielsen, *Neural networks and deep learning*. **url**: `http://neuralnetworksanddeeplearning.com/index.html` (**urlseen** 19/04/2021).

[5] Paperswithcode, *Image classification on mnist*. **url**: `https://paperswithcode.com/sota/image-classification-on-mnist` (**urlseen** 12/04/2021).

[6] J. Brownlee, *How to choose an activation function for deep learning*. **url**: `https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/` (**urlseen** 15/04/2021).

[7] G. Sanderson, *Neural networks*, 2017. **url**: `https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi` (**urlseen** 19/04/2021).

[8] J. Brownlee, *How to choose an optimization algorithm*. **url**: `https://machinelearningmastery.com/tour-of-optimization-algorithms/` (**urlseen** 26/04/2021).

[9] J. Brownlee, *How to choose loss functions when training deep learning neural networks*. **url**: `https://machinelearningmastery.com/how-to-choose-loss-functions-when-training-deep-learning-neural-networks/` (**urlseen** 27/04/2021).

[10] Peltarion, *Loss functions*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions` (**urlseen** 29/04/2021).

[11] Peltarion, *Mean squared error*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error` (**urlseen** 29/04/2021).

[12] Peltarion, *Mean squared logarithmic error (msle)*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-logarithmic-error-(msle)` (**urlseen** 29/04/2021).

[13] Peltarion, *Mean absolute error*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-absolute-error` (**urlseen** 29/04/2021).

[14] Peltarion, *Binary crossentropy*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy` (**urlseen** 29/04/2021).

[15] Peltarion, *Squared hinge*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/squared-hinge` (**urlseen** 01/05/2021).

[16] Peltarion, *Categorical crossentropy*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy` (**urlseen** 03/05/2021).

[17] Peltarion, *Run a model*. **url**: `https://peltarion.com/knowledge-center/documentation/modeling-view/run-a-model` (**urlseen** 03/05/2021).

[18] J. Brownlee, *How do convolutional layers work in deep learning neural networks?* **url**: `https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/` (**urlseen** 06/05/2021).

[19] J. Brownlee, *A gentle introduction to pooling layers for convolutional neural networks*. **url**: `https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/` (**urlseen** 06/05/2021).

[20] Tensorflow, *Post-training quantization*. **url**: `https://www.tensorflow.org/lite/performance/post_training_quantization` (**urlseen** 20/05/2021).

[21] J. Ma, *A higher-level Neural Network library on Microcontrollers (NNoM)*, **version** v0.4.2, **october** 2020. DOI: `10.5281/zenodo.4265016`. **url**: `https://doi.org/10.5281/zenodo.4265016`.

[22] P. Gembaczka, personal communication, 16 **march** 2021.

[23] Tensorflow, *Tensorflow lite for microcontrollers*. **url**: `https://www.tensorflow.org/lite/microcontrollers` (**urlseen** 29/05/2021).

[24] J. Ma", *A higher-level Neural Network library on Microcontrollers (NNoM)*. **url**: `https://github.com/majianjia/nnom`.

[25] Y. LeCun, C. Cortes **and** C. Burges, *The mnist database of handwritten digits*. **url**: `http://yann.lecun.com/exdb/mnist/` (**urlseen** 28/01/2021).

[26] C. Liechti, *Pyserial*. **url**: `https://pypi.org/project/pyserial/`.

[27] Tensorflow, *Pruning in keras example*. **url**: `https://www.tensorflow.org/model_optimization/guide/pruning/pruning_with_keras` (**urlseen** 09/03/2021).

[28] Tensorflow, *Quantization aware training*. **url**: `https://www.tensorflow.org/model_optimization/guide/quantization/training` (**urlseen** 01/06/2021).

Appendix A

MNIST MCU Inference Test Code - PC Python Implementation

```python
import serial
import numpy as np
from tqdm import tqdm
from random import randint
from utils import load_image_data, load_label_data, flatten_images

START_TOKEN = 0xAA
END_TOKEN = 0xAB

CONV = 1
DENSE = 0

class MCU:
    """
    The MCU class handles all low-level communication between the PC and the MCU.
    It handles all data returned from the MCU, and sorts said data into appropriate data containers.
    Different relevant parameters like MCU Clock Speed, and USART baud rate, are also stored in the MCU class.
    """

    def __init__(self):
        self.clock_speed = 20000000
        self.baud_rate = 115200

        #----- Timer/Counter TCA (16-bit) -----#
        # Size of the count register in the TCA module (16-bit)
        self.TCA_CNT_sz = np.power(2, 16)
        # Clock prescaler for the TCA module
        self.TCA_clk_prescaler = 1024
        # Time period of the TCA. Time needed to overflow timer once (in seconds)
        self.TCA_period_s = self.TCA_CNT_sz*(self.TCA_clk_prescaler/self.clock_speed)

        self.mcu = self.mcu = serial.Serial('COM3', self.baud_rate, timeout=10)
        self.DATA_PKG_SZ = 5
        self.data_package = bytearray()

        self.prediction = 255
        self.mnist_execution_time = 0

        self.indices = {
            "prediction": 1,
            "exec_time": [2,3]
        }

    def initiate_transfer(self, image_flat_bytes, conv_or_dense):
        """
        This function handles sending and reception of a pre-structured data package to/from the MCU.
        The PC sends an image from the MNIST dataset to the MCU, which runs inference on said image,\
            as well as measure other desired parameters (e.g. execution time). The MCU responds \
            with a data package containing the various measured parameters.\
            Every data parameter received from the MCU is stored in appropriate class variables.

        :param image_flat_bytes: Image from the MNIST dataset. \
            The image must data must be in the bytes format, and must be a one-dimmensional vector.
        """
        #--------- Message Construction ----------#
        message = bytearray()
        message.append(START_TOKEN)
        message.append(conv_or_dense)
        message.extend(image_flat_bytes)
        message.append(END_TOKEN)

        #----------Serial Transmission -----------#
        self.mcu.write(message)
        self.data_package = self.mcu.read(self.DATA_PKG_SZ)

        #--------- Extract received data ---------#
        if (self.data_package[0] == START_TOKEN) and (self.data_package[-1] == END_TOKEN):
            self.predction = self.data_package[self.indices["prediction"]]

            # The 16-bit value received from the TCA is in register values,
            # and needs to be converted to seconds
            exec_time_msb = self.data_package[self.indices["exec_time"][0]]
            exec_time_lsb = self.data_package[self.indices["exec_time"][1]]
            CNT_diff = (exec_time_msb << 8) | exec_time_lsb
            self.mnist_execution_time = (CNT_diff * self.TCA_period_s) / self.TCA_CNT_sz

        else:
            print(self.data_package)
            raise Exception("Invalid data returned from MCU. Missing START or END token.")


class MCUMnistModel:
    """
    MCUMnistModel provides a high level interface to the MNIST Model running on the MCU. \
    This includes both model inference, and model testing.

    :param conv_or_dense (int): 1 for cnn, 0 for dnn
    """
    def __init__(self, conv_or_dense):
        self.conv_or_dense = conv_or_dense
        n_test_data = 1000
        self.test_labels = np.array(load_label_data('test', n_test_data, 0))
        self.test_data = np.array(load_image_data('test', n_test_data, 0))

        # Cannot transfer negative values over USART.
        # The values are therefore adjusted to int8 range on the MCU
        if conv_or_dense == CONV:
            self.test_data = self.test_data / 4
        elif conv_or_dense == DENSE:
            self.test_data = self.test_data / 8

        self.test_data_flat = flatten_images(self.test_data)
        self.test_data_bytes = []
        for image in self.test_data_flat:
            self.test_data_bytes.append(bytearray(image))

        self.mcu = MCU()

    def run_complete_test(self, n_accuracy_smpls):
        """
        Run a full test of the MNIST model on the MCU. Results are printed to stdout.
        """
        _, accuracy_percentage = self.evaluate_model_accuracy(n_accuracy_smpls)
        execution_time = self.evaluate_execution_time()
        print("\n\n Test Results")
        print("----------------")
        print("Accuracy:\t {:.2f}%".format(accuracy_percentage))
        print("Execution time:\t {:.2f}ms".format(execution_time*1000))

    def predict_single_image(self, image_index):
        """
        Takes a single image from the MNIST dataset, sends it to the MCU for inference, \
        and receives the model prediction in return from the MCU.

        :param image_index (int): Index of the desired image from the MNIST dataset (values in range [0, 999])
        :return prediction (int): Number predicted by the MCU
        :return label (int): Label for the image in question, displaying which number was written in the image
        """
        if image_index >= 1000:
            raise Exception("Image index out of range.")

        self.mcu.initiate_transfer(self.test_data_bytes[image_index], self.conv_or_dense)
        label = self.test_labels[image_index][0]

        return self.mcu.predction, label

    def evaluate_model_accuracy(self, n_test_samples):
        """
        Runs inference on the MCU with a desired number of test samples. \
        The MCU returns its prediction for each image, which is then compared to its corresponding label. \
        Finally the total model accuracy is calculated.

        :param n_test_samples (int): Number of samples to include in accuracy test
        :return accuracy: Model accuracy
        :return accuracy_percentage: Model accuracy in percentage
        """
        correct_predictions = 0
        incorrect_predictions = 0

        print("Evaluating model accuracy...")
        for i in tqdm(range(n_test_samples)):
            model_prediction, label = self.predict_single_image(i)
            if model_prediction == label:
                correct_predictions += 1
            else:
                incorrect_predictions += 1

        accuracy = (correct_predictions/n_test_samples)
        accuracy_percentage = accuracy*100
        return accuracy, accuracy_percentage

    def evaluate_execution_time(self):
        """
        Returns the execution time of the model inference on the MCU. \
        The MCU measures this using a 16-bit onboard timer module, and transmits the resulting execution time. \
        This value represents the execution time of the model inference alone, and excludes any conditioning code.

        :return mnist_execution_time: Execution time of model inference on MCU (in seconds)
        """
        print("Evaluating model inference execution time...")
        self.predict_single_image(randint(0,999))
        return self.mcu.mnist_execution_time
```

Appendix B

MNIST MCU Inference Test Code - MCU C Implementation

```c
#include <stdio.h>
#define F_CPU 20000000UL
#include <util/delay.h>

// weights.h is the header file generated by NNoM,
// which contains the ANN model definition
#include "weights.h"
#include "atmel_start.h"
#include "nnom.h"

#define START_TOKEN 0xAA
#define END_TOKEN 0xAB

uint8_t predict_digit(nnom_model_t  model, int8_t  image_data);

nnom_model_t  model;
uint16_t image_size = 784;
uint8_t prediction = 255;

uint16_t timer_start = 0;
uint16_t timer_end = 0;
uint16_t execution_time = 0;

int main(void)
{
  atmel_start_init();
  _delay_ms(20);

  nnom_model_t  model;
  model = nnom_model_create();

  uint8_t receiving_transmission = 0;
  int8_t image_data[784];
  uint8_t serial_data = 0;

  while(1)
  {
    serial_data = USART_0_read();

    // If received data contains start token,
    // start collecting data
    if (serial_data == START_TOKEN)
    {
      receiving_transmission = 1;
      uint16_t i = 0;

      // Read data packet header byte informing whether or not data
      // should be processed for a dense network, or a convolutional
    one
      uint8_t cnn_or_dnn = USART_0_read();

      // Collect data until end token is received
      while (receiving_transmission)
      {
        serial_data = USART_0_read();
        if (serial_data != END_TOKEN)
        {
          image_data[i] = cnn_or_dnn ? serial_data-32 : serial_data-16;
          i++;
        }
        else {receiving_transmission = 0;}
      }

      // Perform inference on ANN model, and measure execution time
      timer_start = TCA0.SINGLE.CNT;
      prediction = predict_digit(model, image_data);
      timer_end = TCA0.SINGLE.CNT;
      execution_time = timer_end - timer_start;

      // Transmit results to PC over USART
      USART_0_write(START_TOKEN);
      USART_0_write(prediction);
      USART_0_write((execution_time&0xff00)>>8);
      USART_0_write(execution_time&0xff);
      USART_0_write(END_TOKEN);
    }
  }
}

uint8_t predict_digit(nnom_model_t  model, int8_t  image_data)
{
  // Place image data in model input array, and execute inference
  memcpy(nnom_input_data, image_data, image_size);
  model_run(model);

  // Extract the index of the node with the greatest output value.
  // The numerical index value represents the predicted digit
  uint8_t highest_value = 0;
  uint8_t high_value_index = 0;
```

```c
  for (uint8_t i = 0; i < 10; i++)
  {
    if (nnom_output_data[i] > highest_value)
    {
      highest_value = nnom_output_data[i];
      high_value_index = i;
    }
  }
  return high_value_index;
}
```

Appendix C

Dense Keras Models - Size and Accuracy

This appendix displays the Python code used to generate results seen in section 4.2.2.

```python
import numpy as np
import tensorflow as tf
import os
from utils import load_image_data, load_label_data
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, MaxPool2D, InputLayer, ReLU,
    Softmax
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import load_model

physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)

learning_rate = 0.001
batch_size = 64
epochs = 10

#------------ Load datasets -------------------------#
n_train_data = 50000
train_data = np.array(load_image_data('training', n_train_data, 0))
train_data = ((train_data) - 127)/8
train_labels = np.array(load_label_data('training', n_train_data, 0))

n_valid_data = 10000
valid_data = np.array(load_image_data('training', n_valid_data, 50000))
valid_data = ((valid_data) - 127)/8
valid_labels = np.array(load_label_data('training', n_valid_data, 50000))

n_test_data = 1000
test_data = np.array(load_image_data('test', n_test_data, 0))
test_data = ((test_data) - 127)/8
test_labels = np.array(load_label_data('test', n_test_data, 0))
#--------------------------------------------------------#
```

```python

def train_dense(save_path, n_layers, width_l1=0, width_l2=0, width_l3=0):
    if n_layers == 1:
        model = Sequential([InputLayer(input_shape=(28,28,1)),
                            MaxPool2D(pool_size=(2,2), strides=2),
                            Flatten(),
                            Dense(units=width_l1), ReLU(),
                            Dense(units=10), Softmax()])
    elif n_layers == 2:
        model = Sequential([InputLayer(input_shape=(28,28,1)),
                            MaxPool2D(pool_size=(2,2), strides=2),
                            Flatten(),
                            Dense(units=width_l1), ReLU(),
                            Dense(units=width_l2), ReLU(),
                            Dense(units=10), Softmax()])
    elif n_layers == 3:
        model = Sequential([InputLayer(input_shape=(28,28,1)),
                            MaxPool2D(pool_size=(2,2), strides=2),
                            Flatten(),
                            Dense(units=width_l1), ReLU(),
                            Dense(units=width_l2), ReLU(),
                            Dense(units=width_l3), ReLU(),
                            Dense(units=10), Softmax()])
    else:
        raise Exception("Invalid number of layers.")

    model.compile(optimizer=Adam(learning_rate=learning_rate), loss=’sparse_categorical_crossentropy’, metrics=[’accuracy’])
    model.fit(x=train_data, y=train_labels, validation_data=(valid_data, valid_labels), batch_size=batch_size, epochs=epochs, verbose=2, shuffle=True)

    model.save(save_path)

    model.evaluate(test_data, test_labels)
    _, model_accuracy = model.evaluate(test_data, test_labels, verbose=0)

    return model, model_accuracy

def dense_structure_test():
    folder_path = ’tests/model_structure/dense/’
    save_path = folder_path + ’mnist_dense.h5’
    f = open(folder_path + ’DenseStructureTest.csv’,’w’)
    f.write(’n_layers,Layer1 Width,Layer2 Width,Layer3 Width,’)
    f.write(’Model Accuracy,Model Size,Learning Rate,Batch Size,Epochs\n’)

    layer_widths = [1,2,4,8,16,32,64,128]
    layer2_widths = [0]
    layer3_widths = [0]
    progress = 0
    l = len(layer_widths)
    total_models_to_train = l + l**2 + l**3
    for n_layers in range(1,4):
        if n_layers == 2:
            layer2_widths = layer_widths
        elif n_layers == 3:
            layer3_widths = layer_widths

        for width_l1 in layer_widths:
            for width_l2 in layer2_widths:
                for width_l3 in layer3_widths:
                    _, model_accuracy = train_dense(save_path, n_layers, width_l1, width_l2, width_l3)
                    model = load_model(save_path)
                    model_size = os.path.getsize(save_path)

                    progress += 1
                    print("Evaluated {} models out of {}".format(progress, total_models_to_train))
                    print("n_layers:", n_layers)
                    print("width_l1:", width_l1)
                    print("width_l2:", width_l2)
                    print("width_l3:", width_l3)
                    f.write("{},{},{},{},{},{},{},{},{}\n".format(n_layers,
                                                                  width_l1,
                                                                  width_l2,
                                                                  width_l3,
                                                                  model_accuracy,
                                                                  model_size,
                                                                  learning_rate,
                                                                  batch_size,
                                                                  epochs))
    f.close()
```

Appendix D

Pruning Test Code

This appendix displays the Python code used to generate results seen in section 4.2.5.

```python
import os
import numpy as np
import tensorflow as tf
from utils import load_label_data, load_flat_images
from optimization import prune_model
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Activation, Dense, InputLayer
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.models import load_model
from nnom.scripts import nnom_utils
from tqdm import tqdm

physical_devices = tf.config.experimental.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)

#----------------- Load datasets ----------------------#
train_data = np.array(load_flat_images('training'))
train_data = ((train_data) - 127)/8
train_data = train_data.reshape(50000, 784, 1)
train_labels = np.array(load_label_data('training', 50000, 0))

valid_data = np.array(load_flat_images('valid'))
valid_data = ((valid_data - 127))/8
valid_data = valid_data.reshape(10000, 784, 1)
valid_labels = np.array(load_label_data('training', 10000, 50000))

test_data = np.array(load_flat_images('test'))
test_data = ((test_data - 127))/8
test_data = test_data.reshape(1000, 784, 1)
test_labels = np.array(load_label_data('test', 1000, 0))
#-------------------------------------------------------#

def train_dense(save_path):
    model = Sequential([
        InputLayer(input_shape=(784)),
        Activation('tanh'),
        Dense(units=32),
        Activation('relu'),
        Dense(units=32),
        Activation('relu'),
        Dense(units=32),
        Activation('relu'),
        Dense(units=32),
        Activation('relu'),
        Dense(units=10),
        Activation('softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=0.001), loss='
     sparse_categorical_crossentropy', metrics=['accuracy'])
    model.fit(x=train_data, y=train_labels, validation_data=(valid_data,
     valid_labels), batch_size=64, epochs=10, verbose=2, shuffle=True)

    model.save(save_path)

    model.evaluate(test_data, test_labels)
    _, model_accuracy = model.evaluate(test_data, test_labels, verbose=0)

    return model, model_accuracy


def prune_dense_model(model, initial_sparsity, final_sparsity, batch_size, epochs,
     save_path):
    prune_model(model, 'dense', initial_sparsity, final_sparsity, save_path,
     batch_size, epochs)
    pruned_model = load_model(save_path)

    pruned_model.compile(optimizer='adam',
                        loss=tf.keras.losses.SparseCategoricalCrossentropy(
     from_logits=True),
                        metrics=['accuracy'])
    _, pruned_model_accuracy = pruned_model.evaluate(test_data, test_labels,
     verbose=0)
    print('Pruned model accuracy:', pruned_model_accuracy)

    pruned_model.save(save_path)

    return pruned_model, save_path, pruned_model_accuracy

def dense_nnom_model_converter(model, header_path):
    nnom_utils.generate_model(model, test_data[:100], header_path, quantize_method=
     'max_min')


def get_folder_size(start_path = '.'):
    total_size = 0
    for dirpath, dirnames, filenames in os.walk(start_path):
        for f in filenames:
            fp = os.path.join(dirpath, f)
            # skip if it is symbolic link
            if not os.path.islink(fp):
                total_size += os.path.getsize(fp)

    return total_size


def run_test():
    model_path = 'tests/pruning/sparsity_test/dense/run_3/dense_sparsityTest_model.h5'
    nnom_model_path = 'tests/pruning/sparsity_test/dense/run_3/weights_dense_sparsityTest.h'
    pruned_model_save_path = 'tests/pruning/sparsity_test/dense/run_3/dense_sparsityTest_pruned_model'

    _, model_accuracy = train_dense(model_path)
    model = load_model(model_path)
    _, model_accuracy = model.evaluate(test_data, test_labels, verbose=0)

    unpruned_file_size = os.path.getsize(model_path)
    dense_nnom_model_converter(model, nnom_model_path)
    nnom_compression_factor = unpruned_file_size / os.path.getsize(nnom_model_path)

    f = open("tests/pruning/sparsity_test/dense/run_3/PruningSparsityTest.csv", "w")

    f.write("Initial Model Accuracy,Pruned Model Accuracy,Pruning Compression Factor,NNOM Compression Factor,Total Compression Factor,Initial Sparsity,Final Sparsity,Batch Size,Epochs\n")

    batch_size = 64
    epochs = 10
    for final_sparsity in tqdm(range(10)):
        final_sparsity = final_sparsity/10
        for initial_sparsity in range(10):
            initial_sparsity = initial_sparsity/10

            pruned_model, prune_path, prune_accuracy = prune_dense_model(model, initial_sparsity, final_sparsity, batch_size, epochs, pruned_model_save_path)

            pruned_file_size = get_folder_size(pruned_model_save_path)
            pruning_compression_factor = unpruned_file_size/pruned_file_size

            dense_nnom_model_converter(pruned_model, nnom_model_path)
            tot_compression_factor = unpruned_file_size/os.path.getsize(nnom_model_path)

            f.write("{},{},{},{},{},{},{},{}\n".format(model_accuracy,
                                                       prune_accuracy,
                                                       pruning_compression_factor,
                                                       nnom_compression_factor,
                                                       tot_compression_factor,
                                                       initial_sparsity,
                                                       final_sparsity,
                                                   batch_size,
                                                   epochs
                                                   ))
    f.close()
```