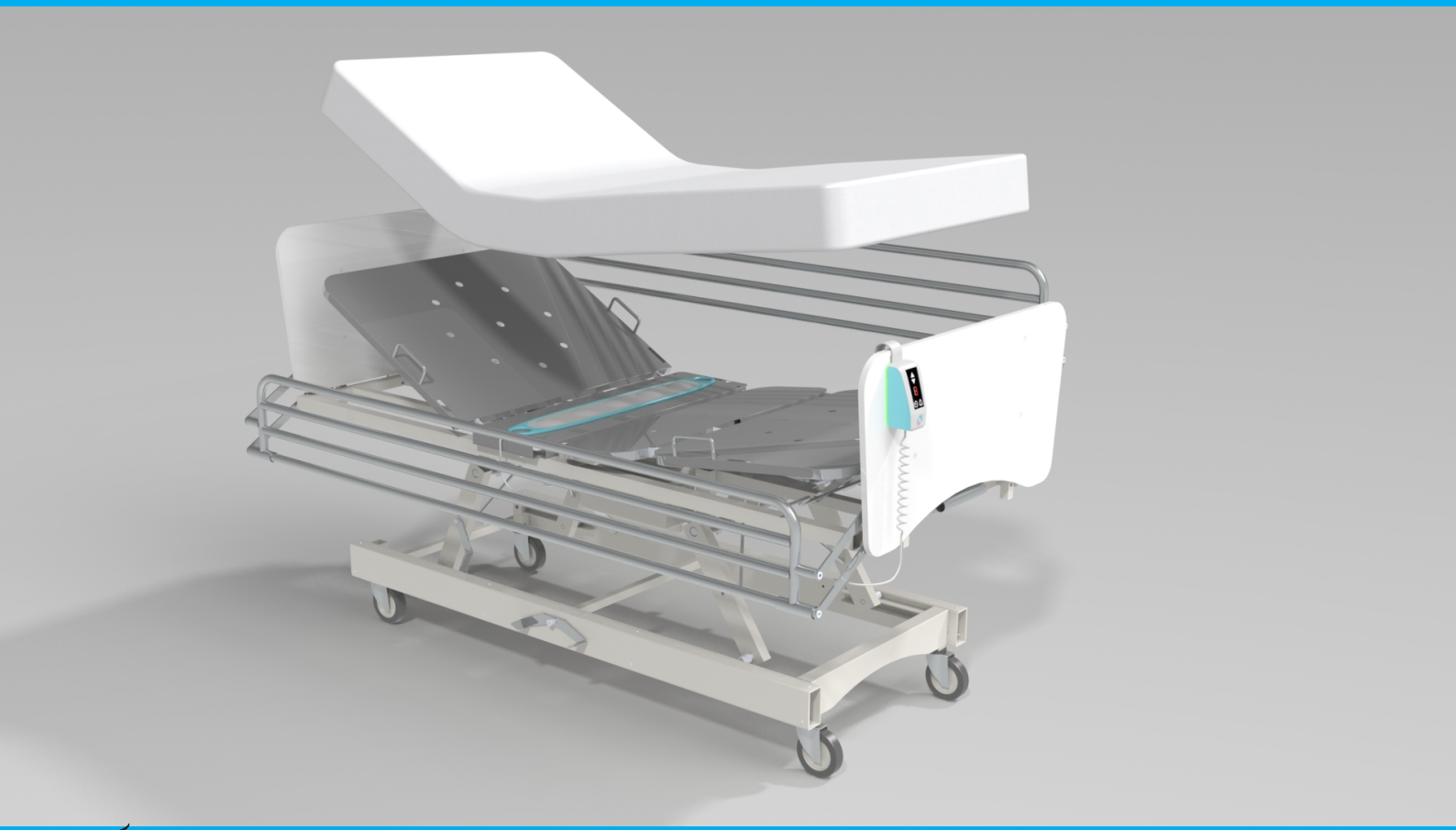
Embedded Neural Networks for Continuous Patient Posture Classification

Vincent Koeten



TU Delft

Embedded Neural Networks for Continuous Patient Posture Classification

by

Vincent Koeten

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday November 27, 2018 at 2:30 PM.

Student number:	4633393	
Project duration:	November 13, 2017 – November 27, 2018	
Thesis committee:	Prof. dr. ir. F. A. Kuipers,	TU Delft
	Prof. dr. P. Pawełczak,	TU Delft, supervisor
	Prof. dr. J. Gonçalves,	TU Delft

This thesis is confidential and cannot be made public until November 31, 2021.

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Preface

Current hospital protocols dictate patients be turned at least every three hours in the effort of preventing pressure ulcers. To reduce the workload of nurses, Momo Medical has created an embedded sensing device to track the patient's posture and notify nurses when it is time to turn them. The challenge presented and the focus of this thesis is classifying the posture of the patient based on the sensor data sampled, specifically, utilizing neural networks on an embedded platform. Furthermore an optimization inspired by recurrent neural networks and ensemble neural networks is proposed, implemented, and compared against vanilla neural networks and pruned variants.

I would like to thank many people for the support and assistance provided. From TU Delft, I would like to thank my professor, supervisor, and thesis committee member, Dr. Przemysław (Przemek) Pawełczak, thesis committee member Dr.ir. Fernando Kuipers, and thesis committee member Dr. Joanna Gonçalves. From Momo Medical, I would like to thank my supervisors Menno Gravemaker and Ide Swager, and fellow MSc student Yanick Mampaey. Finally I would like to thank my family and friends for the wonderful support and keeping me on track.

Vincent Koeten
Delft, November 2018

Contents

List of Figures

List of Tables

1

Introduction

Healthcare solutions have evolved in recent years to take advantage of the advances in technology. Specifically, improvements in telecommunication and embedded systems have allowed for a rapidly growing network of connected and smart healthcare systems. This has made it easier for more diagnostic information to be available to healthcare providers. However, it has raised the demand for automated diagnostic tools that can use the abundance of information, perform primary analysis on the data, and potentially alert healthcare providers.

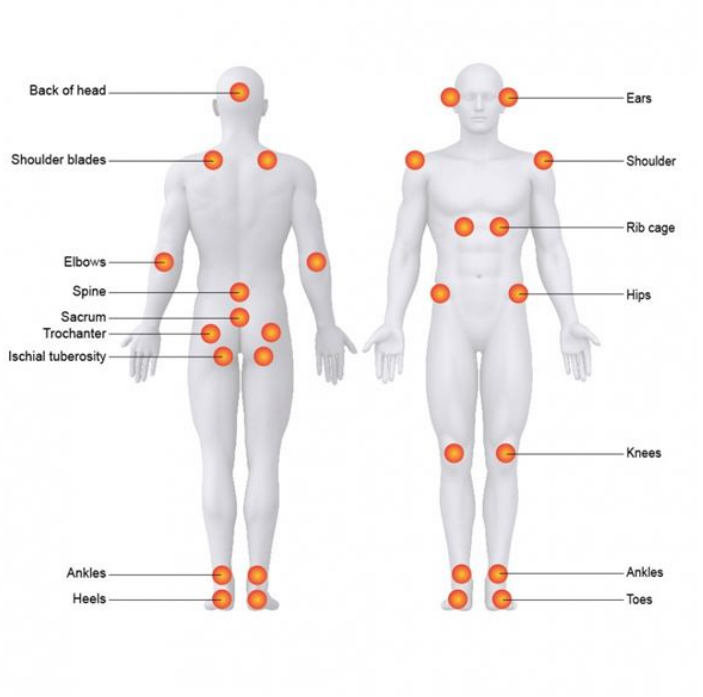


Figure 1.1: Common locations of pressure ulcers [1]

Neural networks have seen significant popularity in recent research projects dealing with medical diagnosis. They perform well because they can generalize and capture underlying relationships between diagnostic parameters and medical conditions that may not be expressly obvious through simple linear relationships. Since the the diagnostic information is usually recorded using embedded platforms it is beneficial to have a neural network that can run on the same embedded systems and perform on-site analysis of sensory data to provide real time diagnostic information rather than having to rely on remote computation

platforms. It is therefore the aim of this thesis to find optimizations in the implementation of neural networks that makes them suitable for use on embedded platforms.

Analysis and diagnostic tools that can augment the work of healthcare providers is one of the motives behind which Momo Medical was founded. Momo Medical is a startup that has designed a sensory platform that assists nurses by detecting the posture of patients lying on a bed in a hospital environment. Posture detection is vital for preventing pressure ulcers in patients which can occur in locations on the body shown in Figure 1.1. This thesis was performed in conjunction with Momo Medical to explore neural networks as an approach to classifying the posture of the patient lying on the bed using the data from sensors.

This thesis will be broken down into seven chapters. In the next chapter, on background, the problem statement will be expanded on and the scope of the thesis will be defined. Chapter 3 explores research relating to neural networks that are the inspiration of this thesis. Chapter 4 will explain the design of the proposed optimization. Chapter 5 discusses challenges and choices made during implementation. Chapter 6 examines the experiments and analyzes the results. Finally, the thesis will conclude in chapter 7 and briefly touch on potential avenues for future work.

2

Background

This chapter will explain the background information that will lay the foundation for the problem statement and provide the motivation for this thesis work.

2.1. Problem Requirements

This section is a brief introduction to the problem of pressure ulcers from a medical standpoint and discusses the challenges in solving this problem.

2.1.1. Need for Posture Detection

Pressure ulcers occur due to a number of factors. The largest being the application of constant pressure on the skin over a long duration. This condition prevelant with patients who are bedridden and lack mobility. Cell death occurs due to the weight of the body on certain pressure points leading to death of tissue under the areas of high mechanical loading [24]. Pressure ulcers worsen the condition of the patient, prolonging the recovery process, increasing the likelihood of developing further pressure ulcers, resulting in a vicious cycle.

To combat pressure ulcers, preventative clinical procedures rely on the presence of hospital caregivers that can reposition patients to relieve the pressure on the body on a single side and evenly distribute it over time across the patient. Determining if this method of pressure ulcer prevention is effective requires studying the relationship between repositioning and reduction in pressure ulcers. However, the data indicating how often the patients were repositioned or repositioned themselves is based purely on input from the nurses and is thus subject to human bias and error. A major factor of bias is that the nurses more strictly follow protocol when they are asked to record information for a study. This skews the result as recorded data deviates from ordinary behavior.

Due to the nurses incredibly busy schedules, they find it difficult to accurately track and remember all patients. This is compounded when the shifts change as the nurses may not pass all the necessary information along to the new shift. This opens up the possibilities for an automated system to assist the nurses in adhering to the protocol.

2.1.2. Restrictions

Momo Medical in this pursuit must comply to the restrictions of the environment that they operate in. These restrictions come from four main sources, the hospital, the patients, the nurses, and Momo Medical. It is beneficial to both the hospital and Momo Medical if the costs to manufacture the sensor system are kept low. For the comfort of the patient and to ease installation by the nurses the sensor plate is placed below the mattress and must maintain a low profile. As the beds in a hospital are mobile, the entire system will be run from an internal battery not relying on constant grid power. Therefore in-order to maximize battery lifetime the sensor plate utilizes as few sensors as possible, designed to sample in a single dimension across the width of the bed and the micro-controller is selected with limited computational power and intended for low-power usage. The previously mentioned requirements limiting

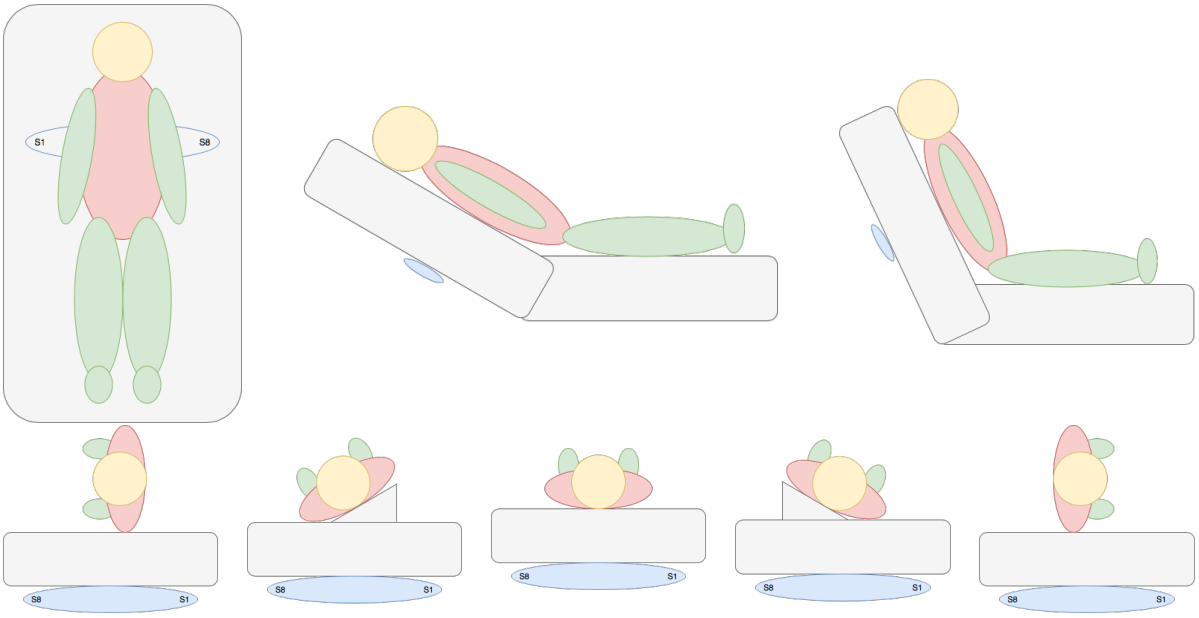


Figure 2.1: From left to right, top row: top-down view of back posture showing the position of the sensor plate in relation to the patient and mattress, side view with mattress elevated to $30°$ seated posture, side view with mattress elevated to $60°$ seated posture; bottom row: $90°$ left laying posture, $30°$ left laying posture with support pillow, back posture, $30°$ right laying posture with support pillow, $90°$ right laying posture

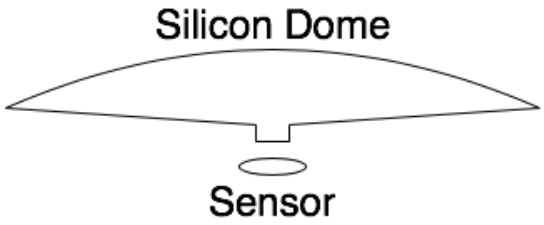


Figure 2.2: Silicon dome placement above piezo-resistive sensor to increase area sensed.

the hardware place restrictions on the software such that if it is able to be executed more quickly, with fewer instructions, it may be possible to change the processor to a cheaper and lower-powered version, reducing costs and extending battery life.

2.2. Sensor Based Solutions

There have been a number of previous attempts at posture classification. Not only for pressure ulcer prevention but also sleep analysis. The sensor systems used in the literature utilize a two-dimensional sensor arrays, for example Yousefi et al. [43], Heydarzadeh et al. [12], Yip et al. [42], Boughrobel et al. [32], and Hsia et al. [13]. These two-dimensional sensor arrays create a heat-map detailing the pressure exerted by the person laying on them. This heat-map is similar to an image and thus image recognition algorithms could easily be applied to determine the posture. On the other hand, Momo Medical arranges the sensors in a single array across the width of the bed to reduce the number of sensors and complexity of the system lowering the cost. Another difference between the aforementioned papers and Momo Medical's sensor plate is the use of silicon domes to increase the surface area affecting each pressure sensor an shown in Figure 2.2. In addition to the pressure sensors, Momo Medical also utilizes piezoelectric sensors and an accelerometer to capture motion and the angle of the backrest for seated postures.

Other possibilities besides pressure sensing arrays under the patient also exist. For example a sensor could be attached on the patient's body. However, this solution is uncomfortable to the patient. Another possibility is a camera focused on the bed and image recognition to

determine patient posture. The cameras will need to be adjusted for changes in bed layout, can not be easily moved to another bed, and are perceived as an invasion of privacy.

Momo Medical aims to use their sensor system to assist hospital caregivers by providing feedback on the repositioning of patients in regards to pressure ulcer prevention. However, the sensor plate requires an algorithm that can translate the given sensor values to a posture. This problem is more complex than can be modeled with a simple linear relationship when factors such as mattress types, patient height, and patient weight effect the sensor output.

2.3. Scope of Thesis

Given the background information, the thesis question can be formalized as: Can neural networks be used for posture classification of the Momo Medical sensor data on an embedded platform?

In addition to the quest for the answer to this question, a deeper goal was defined: to optimize neural network execution for this problem scope.

2.3.1. Why Neural Networks

Seeing the complexity of the problem of classifying postures from the non-linear sensor inputs, Momo Medical wanted to investigate neural networks as a possible solution. Thus this thesis started along this path of inquiry and found a grasp in optimization. Data analysis is available in section 4.1 for additional support for why neural networks are an applicable approach.

3

Related Work

This chapter covers topics that relate to the thesis and which this thesis builds upon in the pursuit of using neural networks for posture classification on an embedded platform.

3.1. Floating vs Fixed Point for Neural Network Computations

Low-powered and low-cost micro-controllers very rarely have hardware floating point operations. When they do, the floating point operations are much slower compared to the fixed point or integer versions. In the absence of floating point hardware there are two options. Software to emulate floating point or use integer arithmetic and bit shifting to implement fixed point. Obviously, the software version will require many instructions to execute as it must manipulate the inputs and outputs multiple times. This will cause programs using floating point arithmetic to slow down considerably. For the quickest execution a fixed point version is the best compromise.

It is therefore important that neural networks can be executed with fixed point arithmetic using integer operations on an embedded system. Gupta et al. explores explores reducing the size of fixed point integers to 16-bit representation [37], Warden states that neural networks can be run using as few as 8 bits [40], Lai et al. utilize a combination of floating and fixed point numbers [19], Langroudi et al. propose using the posit number system [20], and in an extreme case Courbariaux et al. use binary weights and bit-wise operations instead of other arithmetic operations [8].

The ability to use less precise connection weights comes from the idea that the neural network is making generalizations.Thus minor differences in the input should result in the same or very similar outputs. So the lowered precision of the calculations used to produce the output have a minimal consequence. Taking this one step further Binas et al. implement neural networks in analog circuits to reduce the power consumption [3], while others use FPGAs, DSPs, and GPUs as hardware based solutions and optimizations for executing neural networks more efficiently [16] [17] [27] [44]. However, these solutions are beyond the scope of this thesis.

3.2. Ensemble Neural Networks

Ensemble neural networks are the combination of multiple neural networks used together to achieve the desired output. Similar to a musical ensemble the ensemble neural networks can each play a different role or there can be some redundancies to backup and reinforce the primary. For example, in image recognition, there may be a neural network that can detect house cats as the main subject in the image. Another neural network may be able to detect larger cats like lions and tigers. By combining these with other networks that can detect other animals accurately a result is produced if there is a cat, or other animal present in the image. This method can be more accurate than a single more generalized neural network which may confuse more similar subjects like cats and dogs from the example above.

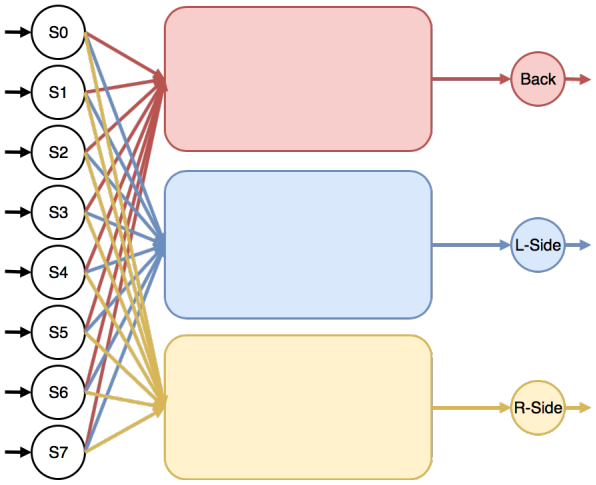


Figure 3.1: Example 'black box' ensemble neural network for posture classification

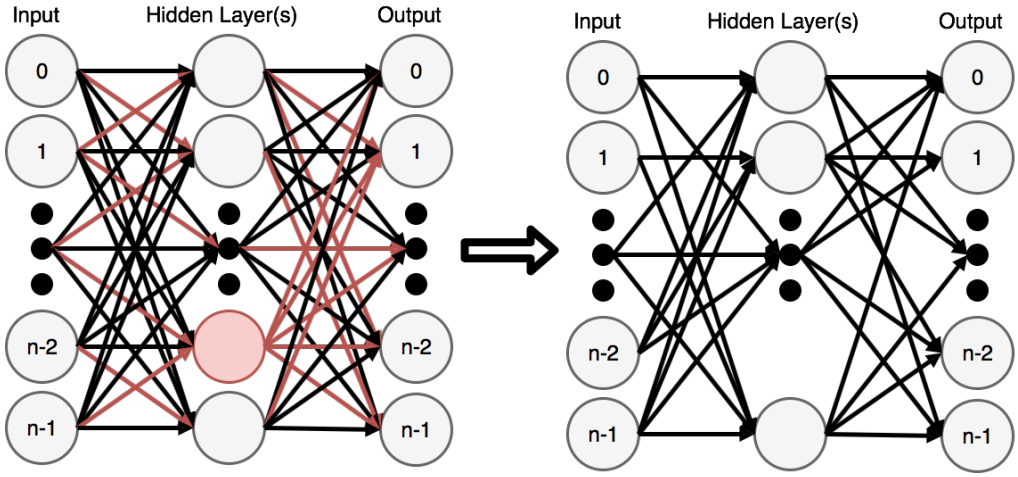


Figure 3.2: Example of pruning connections, in red, from a neural network. Also as one of the nodes would have no outgoing connections it is pruned.

The results of all of the separate classifications can summed together to generate a simple output or as shown by Rogovaa [31], the results can be analyzed to find the strengths of each classifier to further increase the accuracy with probabilities and beliefs. In searching for a good combination of classifiers, Lee et al. have provided some basic properties and conditions, for example reducing the redundant portions [22]. Zhou et al. narrow down ensembles of neural networks from all of the available ones to a selection that is able to generalize better with a smaller size [45].

Figure 3.1 shows an example ensemble of neural networks, with the internals left as a 'black box', working together to classify posture. In this example the networks are all specialized for a single posture.

3.3. Pruning Neural Network Connections

Pruning neural networks removes connections with weights that have the least significance to the resulting output. For example, in Figure 3.2, the red connections are deemed to be unimportant and are thus pruned. Also as one of the nodes would no longer have any outgoing connections it is pruned resulting in the network shown on the right.

Usually the pruned connections have weights which are very close to zero and thus when multiplied by the previous node's value result in a subsequently small value which will have very little effect on the the next node's output. In some cases like with Babaeizadeh et al.,

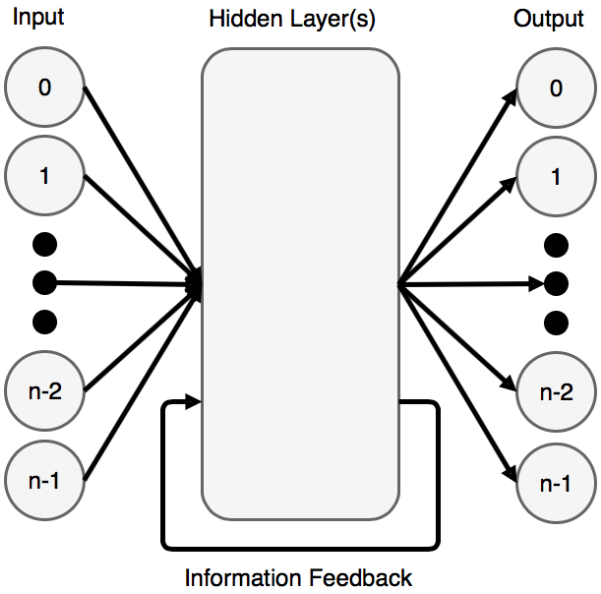


Figure 3.3: General idea of recurrent neural networks where new inputs come in and ouputs are given but also some information is retained as it is fed back into the network.

there is more focus to prune the redundancies [2], while Bondarenko et al. use pruning to reduce over-fitting, improve generalization, and decrease the size of the network [4]. Golub et al. and Manessi et al. continuously prune during training for maximal reduction of the connections and maximizing the possible accuracy of the network [10] [25].

The method of pruning chosen in this thesis is outlined further in subsection 5.2.3 and follows the work of Han et al. to train a neural network, then reduce the unnecessary connections in the network, and finally continue to train the network to reinforce the remaining connections and retain accuracy [11].

3.3.1. Sparsity of Connections

In relation with pruning connections from neural networks, initializing networks with sparse connections can be a useful tool to help train more efficient networks. Various papers discuss methods for structuring sparsity in neural networks along with the potential gains [5] [35] [36] [41]. Though the neural network library chosen, discussed further in subsection 5.1.2, already implements a function to deal with sparsity during initialization of the connections and weights before training so none of the mentioned techniques needed to be implemented.

3.4. Recurrent Neural Networks

Recurrent neural networks are interesting because they can remember information from previous inputs and correlate how that affects the current inputs. Thus they are not focused on a single sample but they work through time and can utilize the history of events to better determine an output. Recurrent neural networks achieve this with connections that flow in the opposite direction of input to output as shown in Figure 3.3.

The downsides are that they are much more difficult to train properly, require an immense amount of labeled training data, and require many more resources for computing the results. This sentiment can be found in the literature [6] [15] [30] [14]. Solutions to simplify the training process of recurrent neural networks exist [6] [7] [18] [28] [30].

Finite-state automata can be modeled in recurrent neural networks as shown by Omlin et al. and Siegelmann and would be very applicable to posture classification [29] [34]. Finally, Shin et al. use fixed point in recurrent neural networks to reduce the hardware complexity [33].

Despite the applicability of recurrent neural networks to the problem of posture classification as a finite-state automata capable of capturing the time-series aspect of posture

changes, the complexity of combining multiple techniques from some of the mentioned papers outweighs the potential benefits. Therefore this thesis will only draw inspiration from the information feedback and time-series aspects of recurrent neural networks.

4

Design

This chapter covers the analysis of the sensor data and how it relates to the classification of postures using neural networks, proposes the design for the optimization at the core of this thesis, and touches on the theoretical bounds.

4.1. Data Analysis

The first step was to analyze the data provided by the sensor system at Momo Medical. At the beginning of the thesis the sensor data was incredibly noisy and inconsistent. This can be seen in Figure 4.1 where the sensor value is the milli-volt reading from an ADC that corresponds to the force exerted on the sensor. The output values from the piezo-resistive sensors vary drastically with weights from 10 grams up to 500 grams. A portion of this can be attributed to the differences in the sensors themselves, however, the majority of the inconsistency is due to the design and manufacturing process of the sensor plates.

The minor differences with the sensors is exacerbated by compounding the inconsistency due to the use of a silicon dome placed over the sensor to increase the surface area being sensed. These silicon domes could vary in stiffnesses and could be tightened down on the sensor to create tension or could be left loose leaving an air-gap between the sensor and the dome as shown in Figure 2.2. Thus some sensors would be preloaded with tension from the silicon dome and have high zero-weight values while other sensors with the air-gap between the dome and sensor resulting in lower sensitivity. In addition the silicon domes could move around slightly in their socket and end up not perfectly oriented over the sensor, further causing varying results even with the same sensor and weight value.

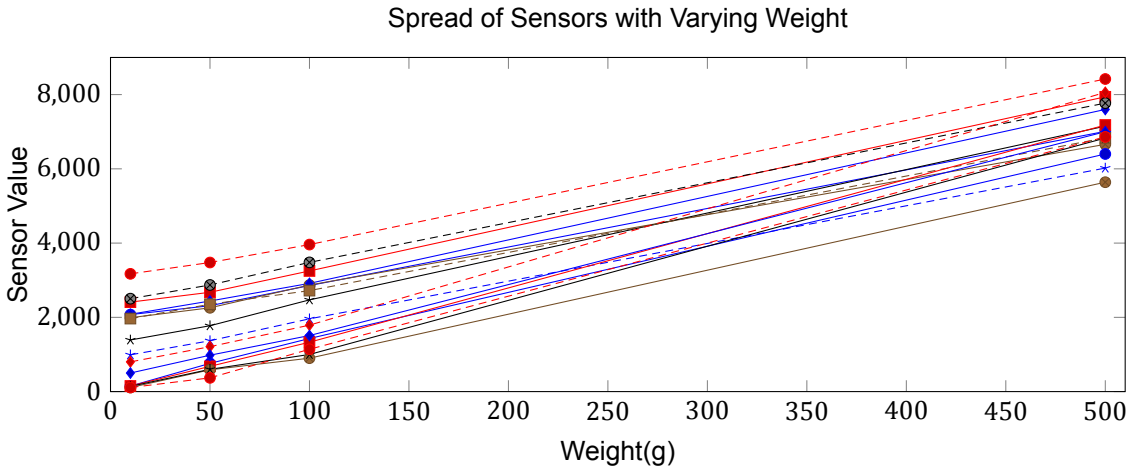


Figure 4.1: Selection of 16 sensors loaded with weight showing inconsistent results

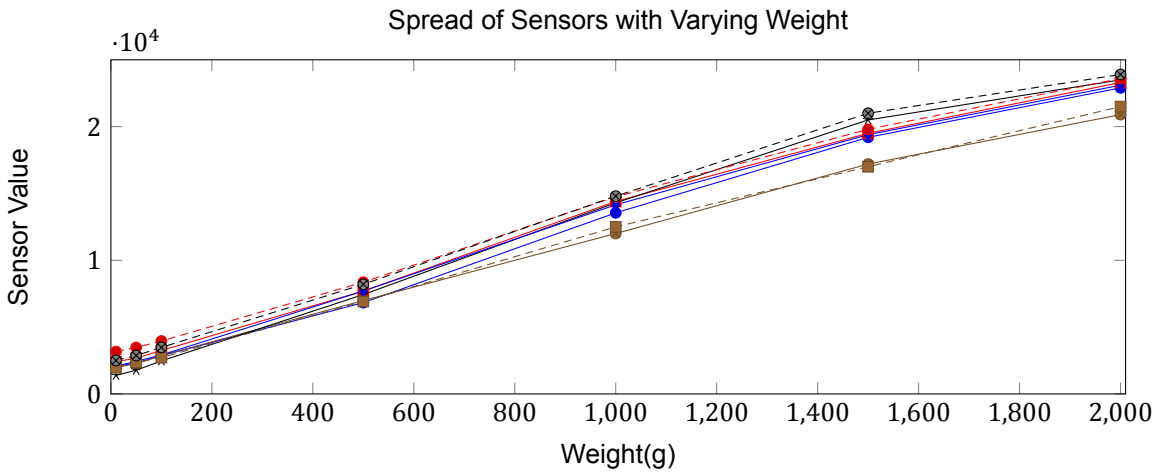


Figure 4.2: Selection of 8 sensors loaded with weight beyond specifications

It was possible to measure the offsets of each sensor and to measure the differences in slopes of the sensor as more weight was applied to create calibration values that could be used to map the sensors to a more similar normalized scale. Unfortunately the calibration values are only valid for a single set of tests. The sensor plates were vulnerable to being jostled, changing the positions, orientation, and contact patches of the sensors and silicon domes. Additionally the sensors are also only rated to work linearly up to $1lb$, approximately $\frac{1}{2}kg$, while they were under potential loads up $12lbs$, approximately $5\frac{1}{2}kgs$ [39]. This can be partially seen in the curve of Figure 4.2 as the sensors become less sensitive to increasing weight beyond approximately $1kg$.

After obtaining the calibration values for a sensor plate a few simple tests were run. Using some care, results were manageable. Figure 4.3 shows two samples of three people across an air mattress and foam mattress laying on their back. Figure 4.4 shows a single sample of two people laying in the back posture and 90° right posture on both the air mattress and foam mattress. These showed indications that a classification between the three main postures, 90° right, 90° left, and back, would be possible, as for a single person, on a single mattress, laying in the same position of the bed, the postures were distinct. However, when combining one person's samples with samples of other people using a variety of mattress types the distinctions become much less clear.

A few observations of these initial samples are listed below. The foam mattress and air mattress react differently to the same person laying with the same posture. In general the foam mattress slightly dampens the pressure exerted on the sensors compared to the air mattress. The 90° right side and 90° left side postures are close mirrors around the center of the bed, thus only the 90° right side posture is shown. The 90° side postures generally have a slightly higher peak than the back posture as can be seen in three of the four samples of Figure 4.4. The 90° side postures have an asymmetrical shape while the back posture has a more symmetrical curve.

The initial sensor data appears optimistic and shows promise that neural networks could successfully classify postures. With the promise of improvements to the sensor plate and sensor consistency and hope for a labeled dataset, work continued forwards.

4.2. Optimization Design

Inspiration was drawn from the research performed and applied to the problem at hand to create an optimization.

During the analysis of the problem of determining a patients posture, it is noted that in the expected case the patient's posture changes very rarely, approximately once every three hours, compared to the sampling rate of $10Hz$. This allows for a very simple predictive solution that will be correct much more often than it is incorrect. In short, predict that the

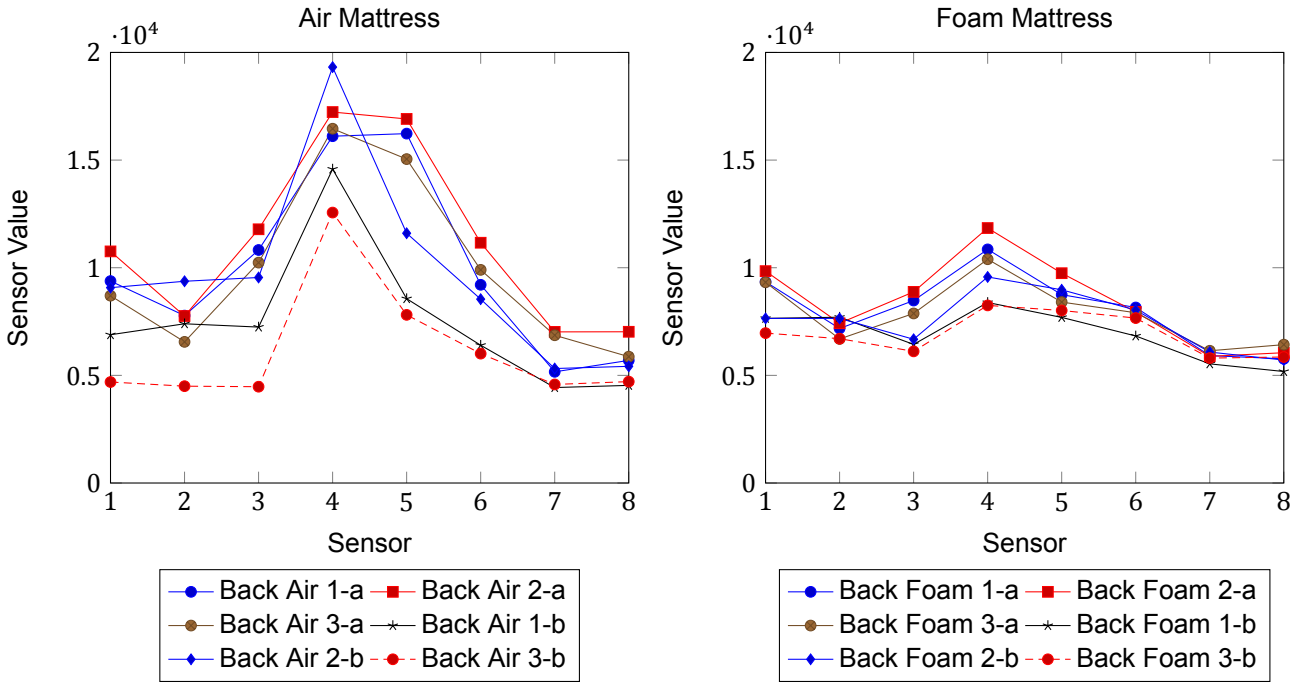


Figure 4.3: Sample sensor plate data of three people on an air mattress and foam mattress

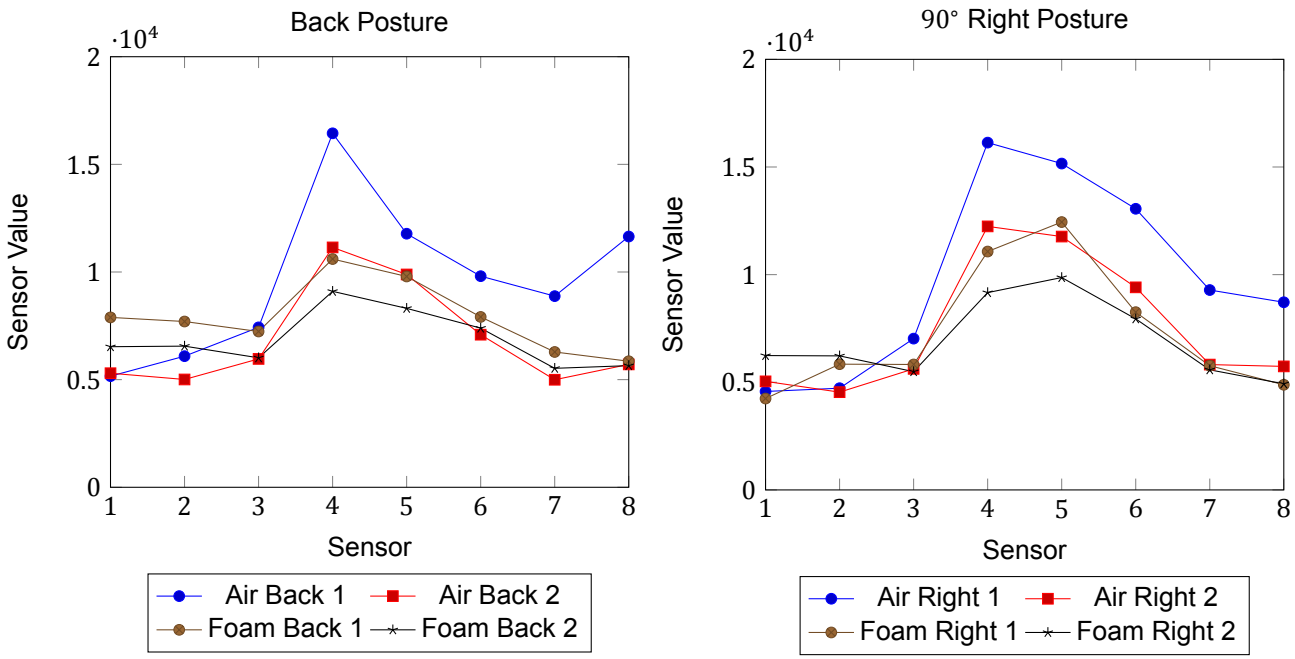


Figure 4.4: Sample sensor plate data of two people on an air mattress and foam mattress laying in both back and right side postures

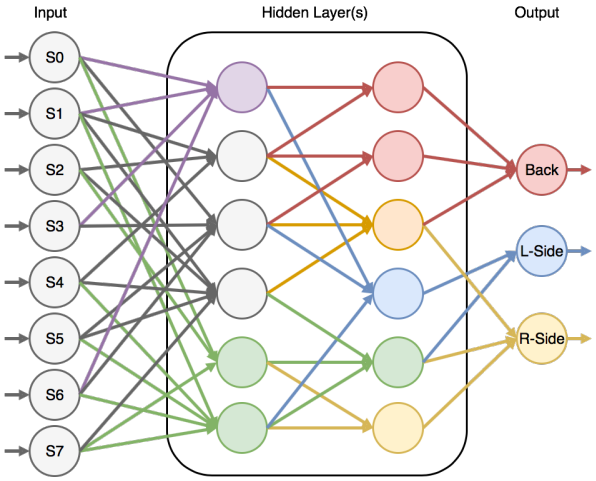


Figure 4.5: Example breakdown of neural network for optimization

patient's posture is unchanged, using the answer from the previous sample. This can be further expanded to check the patients posture over the last few seconds or minutes and use that as a prediction for increased stability. So, until the patient's posture is changed, after possibly three hours or approximately $108,000$ samples, the heuristic will remain correct. Then when the posture of the patient does change the prediction will be incorrect for a few moments, depending on the stability of the prediction, before becoming correct again if there is a proper classification to verify.

Combining this prediction idea to the ensemble neural network shown in Figure 3.1, would allow for computing only the network corresponding to the predicted posture. This output of the computed network could then be compared against a threshold. A result above the threshold and the prediction is accepted as correct. If the output is below the threshold then the remaining branches of the ensemble neural network can be processed and the result from the entire neural network is used. In the vast majority of samples, compute time could be reduced as the remaining networks would not be needed, however, in the case where the patient changes posture, it would still be possible to determine the new posture by computing all of the networks.

Incorporating the sparsity of connections for a single neural network would result in a similar optimization as with the ensemble neural network but reduces the penalty for the prediction being incorrect as some nodes, and therefor computations, overlap between the output classes. Figure 4.5 shows an example where red, blue, and yellow nodes and connections represent each of the three output classes, purple, orange, and green represent the overlap between two output classes, and grey represents the overlap between all three output classes. Then if the back posture was predicted then the grey, purple, orange, and red nodes would need to be computed to check against the threshold. If the prediction threshold is not met then the optimization only needs to fill in the nodes that were skipped over previously, in this example it would be the remaining green, blue, and yellow nodes.

Teerapittayanon et al. and Leroux et al. mention similar principles in trying to reduce the number of computations during execution by trying to avoid portions of the neural network [38] [23]. Though Teerapittayanon et al. exit the evaluation early based on thresholds and Leroux et al. ignore some nodes at the loss of accuracy.

4.3. Optimization Theory

To see how viable the potential optimization theoretically could be, a few equations were used. As the main interest is the connectivity of nodes in the neural network the equations used are similar to the equations of the Erdős-Rényi model of random graphs [9]. However the nomenclature used here differs so that the terms are more relatable to the structure and application of neural networks.

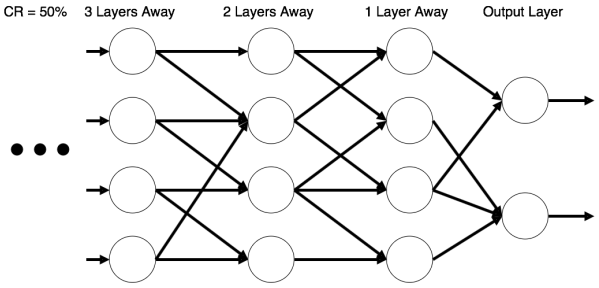


Figure 4.6: Example Neural Network with Connection Rate of 50%

First CR is the connection rate of nodes from one layer to the next in the neural network, $P_C(l)$ is the probability that a node l layers away is connected to an output node, T_l is the total number of nodes in layer l, and $P_D(l,n)$ is the probability that n nodes in layer l are connected to the same output node.

Then the probability that a node is connected one layer away is

$$P_C(1) = CR.$$

Exploring further to a node in the second layer away gives

$$P_C(2) = 1 - (1 - CR \cdot P_C(1))^{T_1}.$$

As the connectivity of nodes in the first layer must also be accounted for. The generalized recursive form is

$$P_C(l) = 1 - (1 - CR \cdot P_C(l-1))^{T_{l-1}},$$

where $T_0 = 1$ to fulfill the base case. Applying these equations, so far, with a randomly connected neural network constrained by a constant sparsity or connection rate, the probability of a node connected to an output can be calculated.

Note these equations only consider some of the most basic rules of neural network structure, that it is formed of nodes in layers with connections going from one layer to the next. In this model it is possible for a node to not be connected to a previous layer or a following layer or both. These equations are not meant to model a perfect representation but to give a close estimation. Figure 4.6 illustrates an example neural network and shows how layers are counted from the output layer.

Using the previous formulas a probability distribution can be generated to view the range of how connected a neural network is as nodes and layers are varied. The following equation is for calculating the probability distribution of n nodes in layer l connected to an output with connection rate CR and connection probabilities as shown in the previous equations,

$$P_D(l,n) = P_C(l)^n \cdot (1 - P_C(l))^{T_l - n} \cdot \binom{T_l}{n},$$

where $n \leq T_l$.

The more completely connected the neural network, the less effective the potential optimization will become, as the optimization takes advantage of the disconnect of nodes. On the flip side, however, the more connected the neural network is the less the penalty is for incorrect predictions because there would be more overlapping nodes already computed for the prediction.

These equations can also be used to generate large tables representing the probability distribution of how many nodes in a particular layer are theoretically connected to an output based on the layers and number of nodes in a network and the connection rate. From these tables, three main factors increase the chance of nodes being connected to an output. They are an increase in the number of layers or depth of the network, the increase in the number of nodes in the layers, and the connection rate. In general, the accuracy of a neural network is also dependent on these same factors so they must be balanced such that the neural network can capitalize on the optimization and yet maintain proper results.

5

Implementation

This chapter provides insight into decisions made and steps taken during the initial stages of this thesis project, the creation and training of neural networks, the implementation of the optimization, and generation of the final dataset used.

5.1. Initial Choices

This section briefly delves into the reasoning behind two of the major initial choices that needed to be made to begin working.

5.1.1. Dataset

Finding an alternative dataset was critical as the sensor plate was under constant development and a large, labeled dataset did not yet exist. The MNIST dataset [21], a set of numeric handwritten characters arranged near the center of a 28 by 28 pixel greyscale image, was chosen. Other datasets were available elsewhere online, however, the MNIST dataset was sufficient. The MNIST dataset is used in benchmarks and teaching neural networks, it contains more than two distinct output classes, and it is sufficiently complex with some similar output classes yet is simple enough that learning about the training process does not overwhelm the thesis project. Also, the order of the data can easily be rearranged to mimic the expected data stream of a patient's posture. For example a sequence of 0's can be followed by a sequence of 1's, and then 2's, etc., imitating a patient moving from laying on their back, to their left side, to their right side. This will allow testing of the optimization using a prediction that is computed from the previous samples rather than feeding it in as additional information.

5.1.2. Neural Networks Library

The neural network library selected is a community driven, open source fork of Fast Artificial Neural Network (FANN) [26]. The decision was based on the facts that FANN is built in C, it is fairly lightweight but contains the necessary set of features, works for both floating point and fixed point, and contains an algorithm to convert floating point to fixed point. Thus this library can easily be modified to run the optimization and ported to a micro-controller platform.

5.2. Training Neural Networks

Training neural networks took tremendous effort, patience, and trial and error. There were four initial variants of neural networks to train, test, and compare. The following section will discuss the training of the neural networks used and explore the major differences and similarities between them. First, vanilla neural networks, the simplest form of neural networks. Second, ensemble neural networks, a combination of neural networks working together. Third, pruned neural networks, removing connections from vanilla neural networks.

And finally, generating the necessary dependency sets from the vanilla neural network, for use in the proposed optimization.

5.2.1. Vanilla

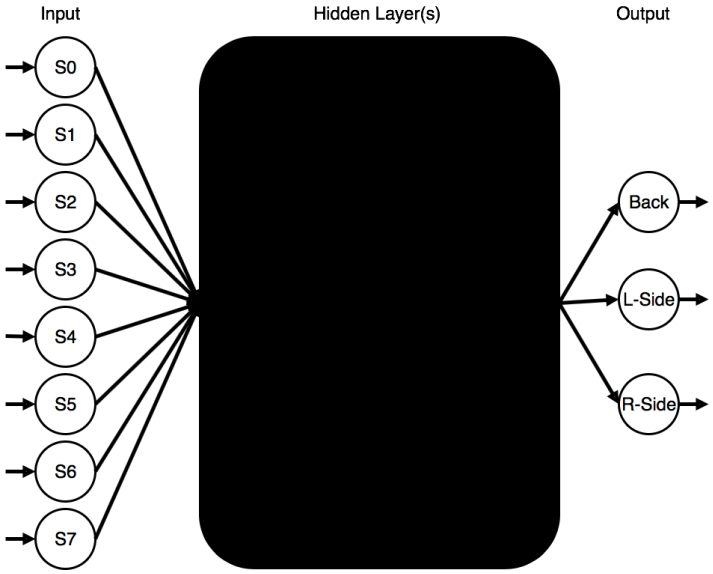


Figure 5.1: Example vanilla neural network with 'black-box' for hidden layers

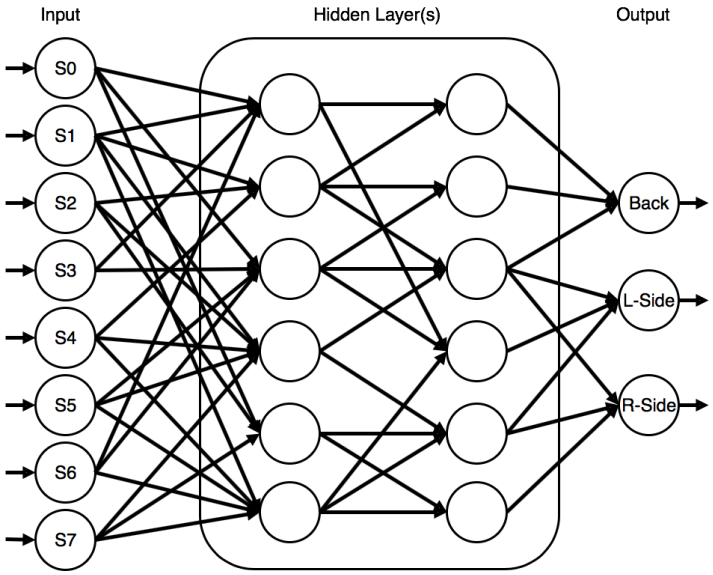


Figure 5.2: Example vanilla neural network with two hidden layers

Vanilla neural networks are the basic form of neural networks. Containing layers of nodes that feed information forward to compute an output. Generally the hidden nodes of a neural network are not examined as in Figure 5.1 though looking into the hidden layers Figure 5.2 could appear. The neural networks trained at the beginning were simple, utilizing back propagation training on a small network. As the size and complexity of the neural networks increased so to did the knowledge put into training them. A forced connection sparsity, L2 regularization, and a validation dataset for determining when best to stop training were included in stages. Using a forced connection sparsity means randomly reducing the connections down to a specified limit, while L2 regularization is a technique for keeping the connection weights low.

The vanilla neural networks form the basis of two of the other alternatives, namely the pruned and optimization variants.

5.2.2. Ensemble

This implementation of an ensemble neural network was done by training individual neural networks for each of the output classes then utilizing them together to compute a complete set of output values for each sample input, see Figure 3.1. It is also possible to use the ensemble neural networks in a rough version of the prediction optimization by computing the predicted output's respective neural network and only if the resulting output is not above a threshold then computing the remaining neural networks to complete the output. This showed in an early test the advantage of the predictive nature of the problem but has a larger overhead as there is no overlap between the ensemble neural networks and thus if the prediction fails every node in the remaining ensemble must be computed.

5.2.3. Pruned

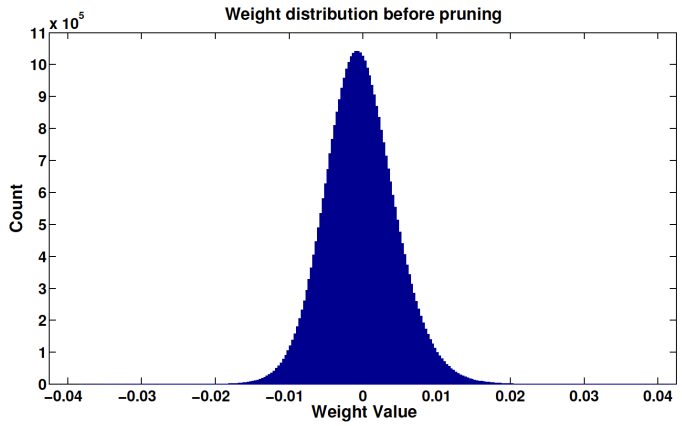


Figure 5.3: Histogram of initial connection weights from [11].

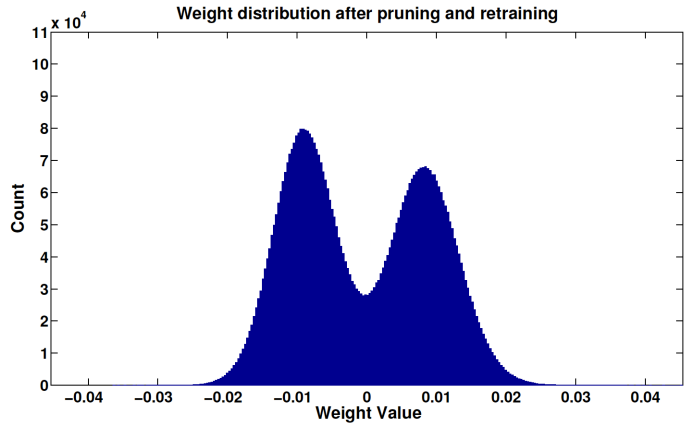


Figure 5.4: Histogram of pruned and retrained connection weights from [11]. Note 10x smaller scale than Figure 5.3

The implementation of the pruned neural networks follows the steps of Han et al. [11]. By training a fully connected vanilla neural network some connections will have a low importance. Then by analyzing the the connection weights a bell curve centered around 0 is created in the connection weight histogram, like in Figure 5.3. Next step is removing the least important connections, those closest to 0. Followed by further training, or retraining, the neural network to strengthen the remaining connections and possibly reveal additional weak connections. Figure 5.5 is the plot of the histogram of the connection weights for one

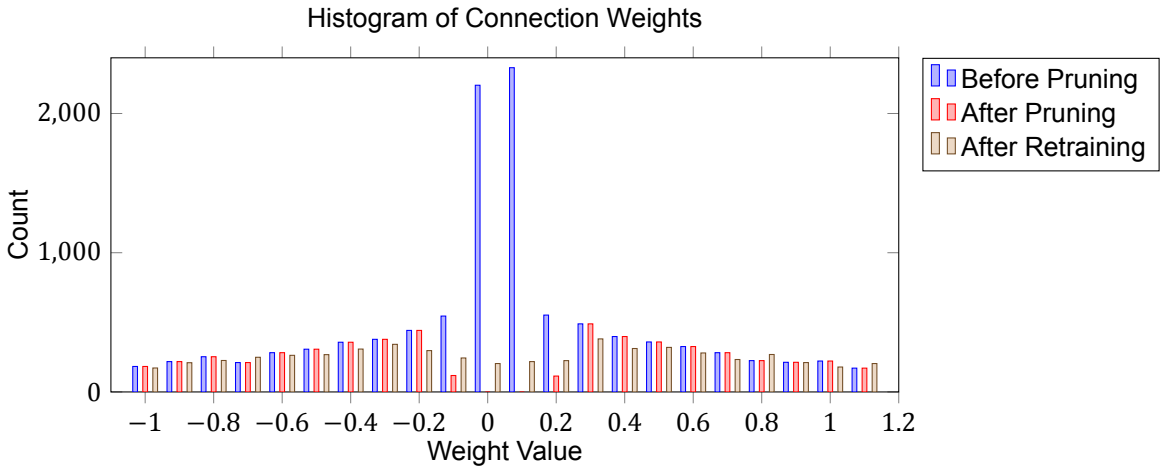


Figure 5.5: Histogram of connection weights of a neural network before pruning, after pruning, and after retraining

of the pruned networks used in this thesis. This is similar to the results shown in the difference between Figure 5.3 and Figure 5.4. Thus a large neural network is pruned down. The benefits being the number of computations needed to evaluate the network has been reduced and also the memory footprint has been made smaller though potentially at the cost of some accuracy.

5.2.4. Generating Dependency Sets

One of the highlight aspects of training neural networks for the proposed optimization is that it is just like training any other neural network so long as the connection rate is well below 100%. Once the network is trained, the sets of dependent nodes for each output need to be found. This is done by treating the neural network backwards as a tree with each output node as a separate root. Then a breadth-first traversal will discover all the nodes that the particular output is dependent on. Creating these lists of dependencies is the only additional step in preparing for the optimization. It is possible to generate these lists as the neural network is loaded during runtime, however, for this thesis they were precomputed and loaded separately.

5.3. Optimization

There were a few design choices to be made in implementing the optimization. First, how the predictive execution knows which nodes to execute.

An option was to explore the nodes during execution, starting from the predicted output with a breadth-first traversal on the connections until reaching the input layer then going backwards through the discovered nodes computing the result. This, however, has the downsides of adding complexity to the already complicated computation of the neural network, has a limit of how many nodes can be recursively explored due to a limited recursive stack, and will need to be run through every time the neural network is executed. It also becomes more complex to determine the remaining nodes to be calculated if the prediction does not reach the threshold.

An alternative is to maintain lists of nodes which each output is dependent on and based on the prediction iterate through the appropriate list for the nodes to compute. The lists would ideally need to be precomputed either once after training is completed or when loading the neural network and will take additional memory. If the prediction fails it is simple to iterate through the entire neural network and skip computing the nodes that are in the list. A possible option, depending on the connectivity of the neural network could also be to use the list of nodes not in the dependency set based on which set is smaller.

The decision was made to precomute the dependencies due to the less restrictive limitations. The additional memory constraint of the second option is not a large concern as the

number of nodes in the neural networks will be well below 65,535. Thus the lists of dependent nodes can be stored using unsigned 16 bit integers. If the neural network size is below 255 then it could be possible to store the listed values with 8 bit integers, however, as the initial dataset, the MNIST dataset, contains 784 inputs, 16 bit integers were used. The lists can also ignore the input nodes, as no computations need to be made and they only need to be loaded into memory. Also, as evidenced by the theory and backed up by most neural networks trained, the output nodes usually have a dependent connection to most or all of the input nodes. It may be possible, but unlikely, to further optimize by including the input nodes in the dependency lists but that was not investigated.

```
// algorithm inputs: nn - the neural network,
//     data_inputs - the input values from the sample,
//     prediction_id - the id of the output that is being predicted,
//     dependency_lists - dependency lists in an array by output node ids,
//     prediction_threshold - the threshold to compare against the output

load data_inputs to input_nodes of nn
for node_id in dependency_lists[prediction_id]:
    compute(nn[node_id])
if output[prediction_id] >= prediction_threshold: // prediction succeeds
    return outputs // predicted output will retain value, others will be 0
else: // the prediction failed
    for node_id not in dependency_lists[prediction_id]:
        compute(nn[node_id])
return outputs // all outputs are computed
```

Listing 5.1: Pseudo-code of optimization algorithm

Now that the optimization has been implemented from the pseudo-code provided in Listing 5.1, there are a few things to consider for executing on an embedded platform. If this optimization was run on a real-time operating system, it would finish before its maximal deadline when the prediction is correct, and would run until the maximal deadline when the prediction fails. This jitter introduced by the optimization task may be unacceptable for the task scheduler and complicates the computation-cost analysis for the system.

In regards to the initial problem of classifying patient posture, an analysis through time of the sensor data will likely provide more accurate results than examining each, single sample. Unfortunately, with neural networks, this becomes too resource expensive for an embedded platform. Thus the neural network chosen does not utilize all available sensors in the sensor plate provided by Momo Medical. Only the piezo-resistive, force sensors were used, as a single sample of these sensors provides a snapshot of the pressure exerted by the patient. The piezo-electric sensor is useful in detecting motion and movements of the patient and can even be used to detect breathing and heartbeats in a time analysis. The accelerometer was also excluded from the neural network as it is used to detect the seated posture and angle, while the backrest of the bed is raised, with simple thresholds.

5.4. Generating Momo data

As the end of the thesis approached the expected labeled dataset from Momo Medical still did not exist. Thus there was a decision to either continue to use the MNIST dataset or generate a dataset based on some characteristics of the actual Momo sensor plate. In respect to the initial problem of patient posture classification the decision was made to move forward with creating a dataset based on samples from the sensor data.

The first step was to analyze the current sensor's data and find characteristics that could be reproduced programmatically. The sensors and samples have become much more consistent as shown in Figure 5.6, where the two subjects got in and completely out of bed between each sample, compared to the previous data in Figure 4.4. The curves are now much more similar and appear to be shifted or displaced in both the x and y axes.

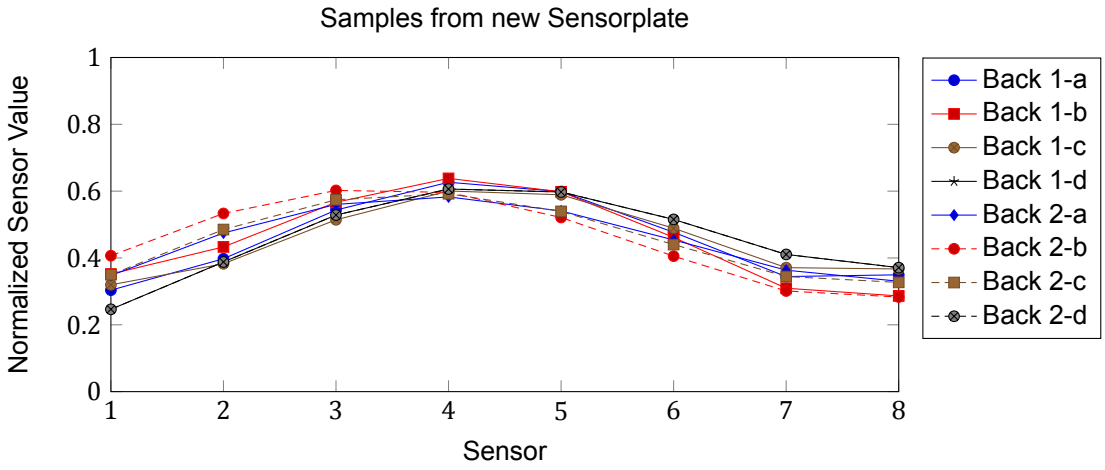


Figure 5.6: Samples of two people laying on Back from the new sensor plate.

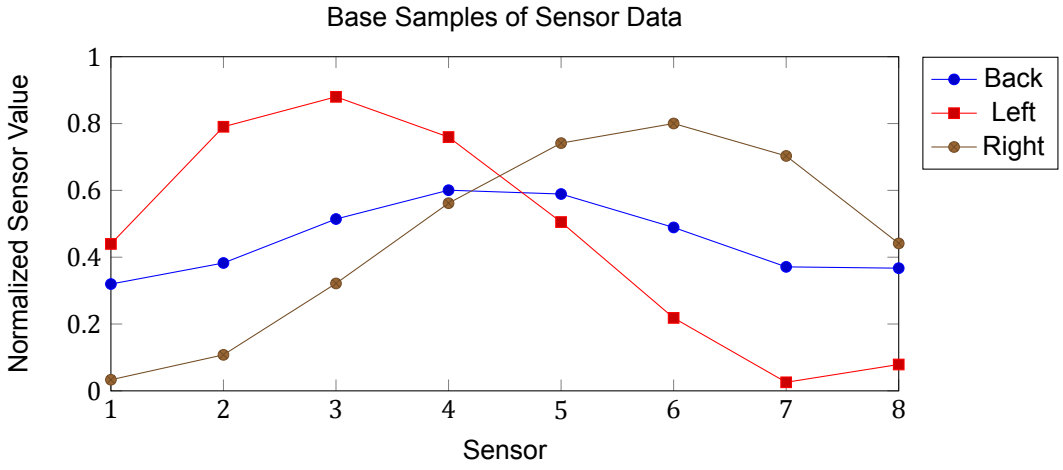


Figure 5.7: The base samples from which the dataset is programmatically generated from

Some minor features of the data are that it is sampled at $10Hz$, there are occasional minor spikes for patient adjustments and small movements, transitions between postures generally last between $\frac{1}{2} - 4seconds$ but could be longer, and the patient is located near the center of the bed.

Due to the similarities between subjects and tests of the same posture, the data generated is based on samples taken from a single person laying on a single type of mattress shown in Figure 5.7. Only three postures will be tested, left side 90°, right side 90°, and back. Upon these three base samples a variety of translations can be applied as shown in Figure 5.8, similar to how the curves in Figure 5.6 seem to be shifted. The translations include a vertical shift, up or down, a horizontal shift, towards sensor 1 or towards sensor 8, a horizontal flip, mirrored across the x-axis, or a combination.

Next noise is added to the generated data. Figure 5.9 shows a noisy 10 seconds and clean 10 seconds from the data gathered. As the noise on the sensors comes from both physical changes, like breathing, minor movements, and electrical disturbances, for example $50Hz$ AC interference, the noise is time dependent. However, as the neural network only focuses on a single sample per execution, the noise added to the generated signals, shown in Figure 5.10, is a random value, up to 5% of the original, added or subtracted. The real-world data also shows some drift as either the patient, mattress, sensor, or a combination, settles. Again this was not added to the generated data as again the neural network only works on a single sample.

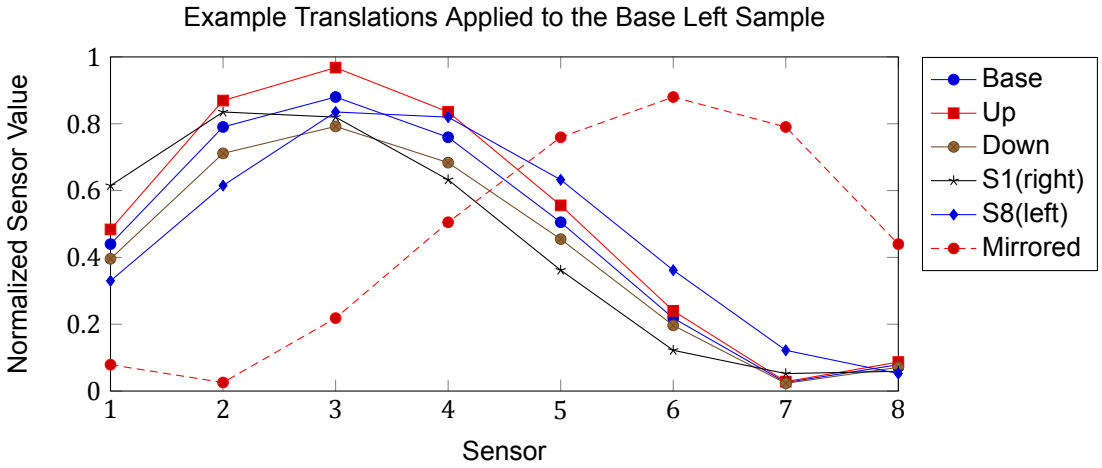


Figure 5.8: Examples of translations used in programmatically generating the dataset

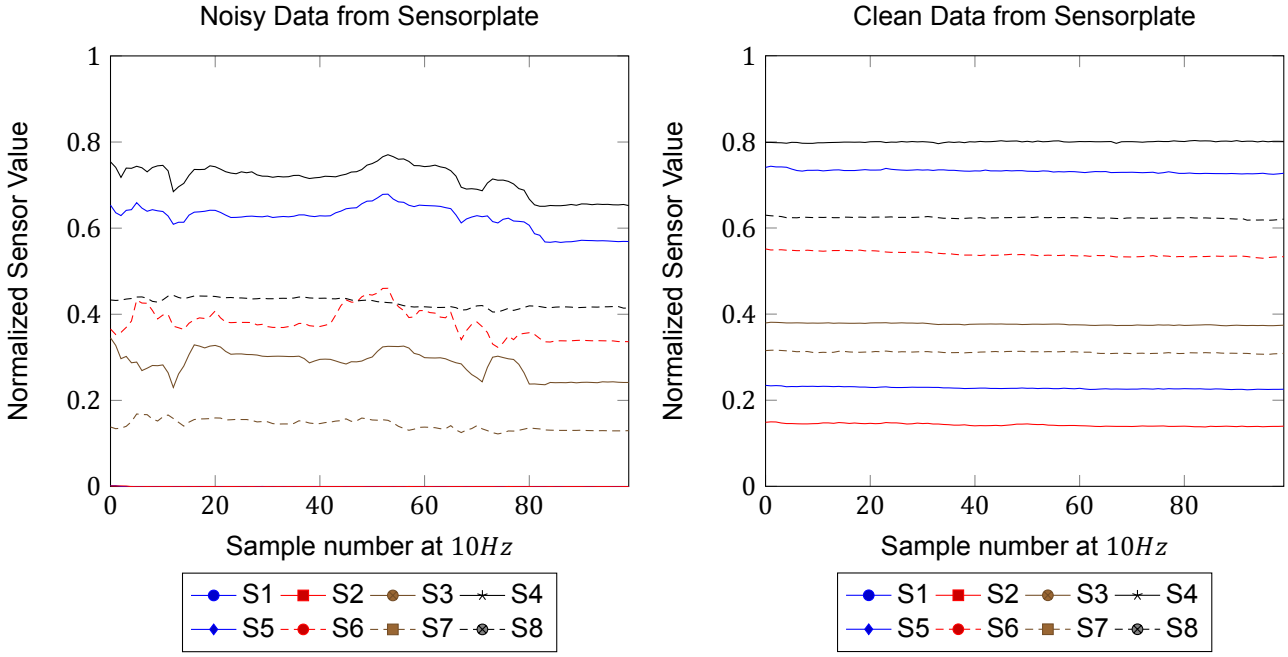


Figure 5.9: Noisy sensor data on left, clean sensor data on right

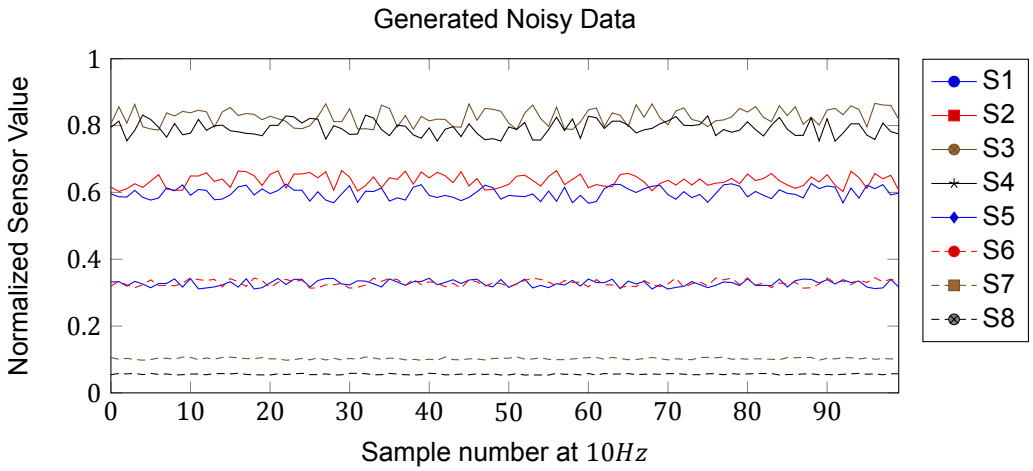


Figure 5.10: Noisy generated data

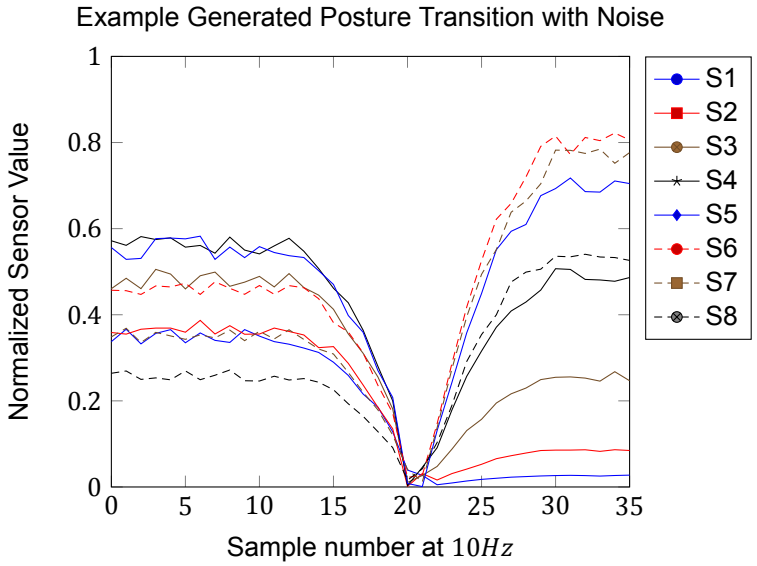


Figure 5.11: Example transition generated with noise

Finally transitions are added between changes in posture. Figure 5.11 shows that transitions go to zero briefly as imitating actual transition programatically was difficult.

One additional note is that all the data generated and used for generation has been normalized to be between 0.0 and 1.0.

6

Experiments and Results

This chapter will cover the experiments performed to test and compare the optimization to known methods and the results from those experiments.

6.1. MNIST Experiments

The experiments in this first section were performed with the MNIST dataset on a PC using floating point. The purpose being to test that the accuracy is acceptable, verify that the optimization works as designed, and to obtain an early indication of execution times.

An accurate vanilla network with a connection rate well below 100% trained on the MNIST dataset was chosen. It had a single hidden layer with 200 nodes and a connection rate of 60%. Further information on the training of this network can be found in subsection 5.2.1.

Ensemble neural networks were trained and were chosen for having a high accuracy with a small number of hidden nodes However, as there is a neural network for each of the ten possible outputs, the exact details will not be discussed though more information can be found in subsection 5.2.2.

A separate vanilla network was chosen for the pruning. In this case a neural network with a 100% connection rate and a lower number of hidden nodes to keep the computation time more similar was chosen. In this case the network contained 100 nodes. This network was then pruned following subsection 5.2.3

Finally, the dependency sets were calculated for the vanilla neural network so a direct comparison between the vanilla execution and the optimization could be made. This was done according to subsection 5.2.4. It is therefore critical that the vanilla neural network have a connection rate below 100% to be able to take advantage of the optimization.

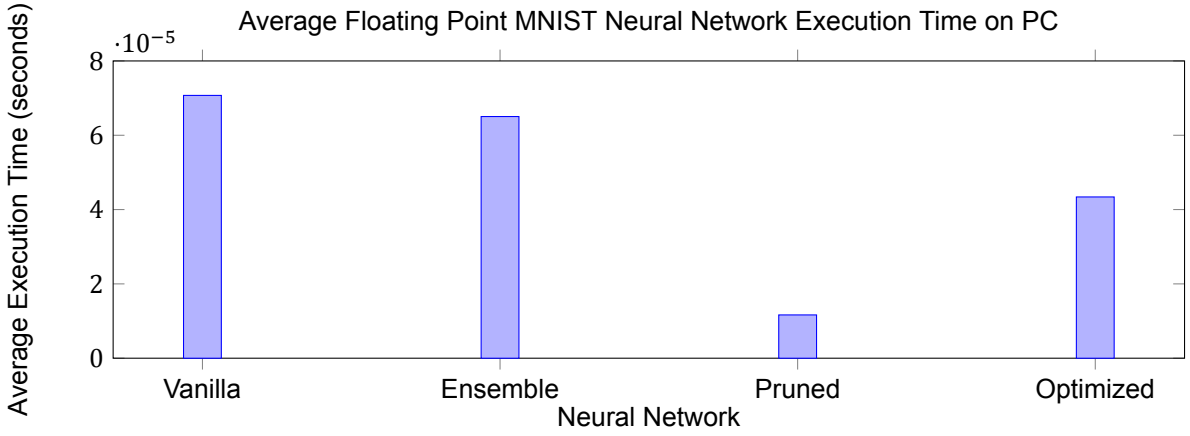


Figure 6.1: Comparison between the average execution time for the four neural network methods using the MNIST dataset

	Vanilla	Ensemble	Pruned	Optimized
Average Execution Time (Microseconds)	70.7256	65.0387	11.6468	43.3975
Accuracy (%)	97.17	94.88	94.78	97.44

Table 6.1: The average execution time and accuracy for the four methods used on the MNIST dataset.

These four methods or sets of neural networks were then tested on the same 10000 MNIST data samples on a PC running at $2.2GHz$ using a single dedicated core. The timing results are shown in Figure 6.1 while Table 6.1 shows both the average execution time and accuracy together. The vanilla neural network acts as a baseline with the hopes that each of the other methods can show improvement upon the execution time while retaining a similar accuracy.

As is evident from Table 6.1 the ensemble neural networks is overall the worst, having the second slowest average execution time and the second worst accuracy. The ensemble neural networks were also the most laborious to train and do not guarantee that they will all use the same decimal place for the eventual fixed point execution. This would then require additional complexity and computations to the inputs of each of the networks in the ensemble further negating any execution time advantage. Thus it was decided not to continue testing the ensemble neural networks in the following experiments.

Unlike the ensemble both the pruned neural network and the optimization show promise. The pruned network executes quickest with an approximate speedup of $6.07x$ but has the lowest accuracy, losing by nearly 3%, while the optimization provides an approximate $1.63x$ speedup and improves slightly on the accuracy of the vanilla neural network. However, that is not the full story for the optimization.

As mentioned during the design and implementation when the prediction threshold is not met the optimization must go back and calculate the nodes that were initially skipped over. A threshold of 70% of the maximum output was used, or 0.7 in this case as the maximum output is 1. Thus for some samples there is overhead in comparison to the computation time of the vanilla neural network. For this subset of test cases the optimization had an average execution time of $69.7188\mu s$ while the vanilla neural network for the same subset of samples had an average execution time of $67.2422\mu s$. Thus the overhead of the failed optimization is $2.4766\mu s$ or approximately 3.7% of the vanilla neural network execution time. However, as mentioned before the optimization's average execution time is much lower at $43.3975\mu s$. This shows that usually the threshold is being met and thus the optimization is able to take an advantage in most cases.

The flip side to this threshold is that when using a correct prediction the accuracy can increase for the optimization. This is already clear from Table 6.1 where the accuracy is increased by 0.27% or 27 samples from the test set. While this is a very small fraction of the total test set, it is still a significant result and changing the threshold could affect the accuracy and average execution time. By increasing it, the neural network must be more sure that the result is correct and may default more frequently, however, by decreasing the threshold, if the predictions are inaccurate, then the accuracy may fall.

6.2. Generated Momo Medical Data Experiments

The experiments in the following section were performed with neural networks trained on the generated Momo Medical sensor data from section 5.4. The purpose of these experiments is to further explore the comparison in computation time of the various neural network options as well as comparing floating point and fixed point on PC and fixed point on the micro-controller. As the sample data can be considered fairly sterile and simple, accuracy will not be discussed in great detail.

New neural networks were trained on this generated Momo Medical dataset following the steps in section 5.2. As mentioned in section 6.1 the ensemble neural networks will not be included in the following experiments.

For this new dataset two vanilla neural networks were chosen to have an additional set of parameters to compare. The two differences were the number of nodes in the single hidden layer and the connection rate. The first neural network, A, had 350 hidden nodes and a

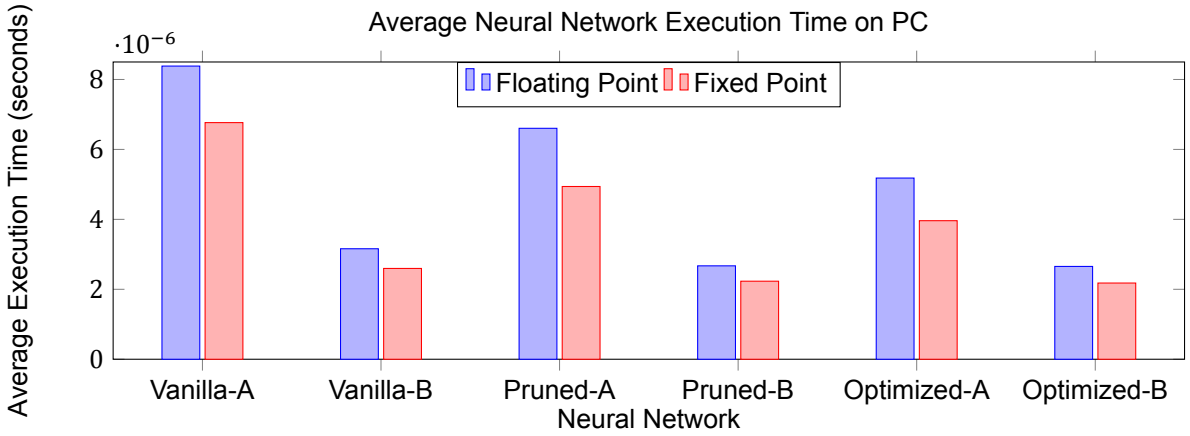


Figure 6.2: Comparison between the average execution time for different neural networks and implementations in both floating point and fixed point on PC

	Vanilla-*A*	Vanilla-*B*	Pruned-*A*	Pruned-*B*	Optimized-*A*	Optimized-*B*
Floating (μs)	8.383	3.159	6.604	2.671	5.181	2.654
Fixed (μs)	6.766	2.597	4.940	2.232	3.962	2.179

Table 6.2: The average execution time for the six neural networks in both floating point and fixed point on the PC.

connection rate of 50%, the second network, *B*, was smaller at 100 hidden nodes but had a higher connection rate of 70%. Additional prefixes will be added to further specify additional distinctions. Both neural networks had very similar accuracies over the 9000 test samples of 99.45% and 99.5% respectively, a difference of only four failed test cases.

The idea behind choosing these two different neural networks is to see if they react to the optimization differently. As one network is much larger than the other but has a lower connection rate while the smaller network has a higher connection rate and may not see as much benefit from the optimization.

Another difference for these sets of experiments is that both the pruned networks and the optimization networks will be based off the two different vanilla neural networks. This will allow for a more direct comparison of execution times.

The pruned networks took the two vanilla neural networks mentioned and followed the pruning method outlined in subsection 5.2.3. However, with the goal that accuracy after the pruning and retraining is much closer to the accuracy of the vanilla networks at the sacrifice of removing fewer connections resulting in a potentially higher than necessary computation time and memory size. Still, approximately half of the connections were able to be pruned away from both vanilla networks. In the case of *A* the number of connections went from 2278 to 1093 while with *B* they went from 873 to 445. After retraining the accuracies of both pruned networks remained in the $99.45 - 99.5\%$ range.

The optimized networks remain unchanged from the vanilla networks while the dependency lists are generated for each output according to subsection 5.2.4. The prediction value is based on only the single, immediate, previous iteration though as mentioned in section 4.2 this could be expanded to include more samples. However, for these tests, using only the previous output worked well and was very simple.

6.2.1. PC Floating and Fixed Point

These floating point tests on the PC were to see how the generated data and new neural networks worked while the fixed point tests were used to show that the converted fixed point versions of the neural networks worked properly. The tests were executed on the same PC as before with the MNIST experiments. As mentioned in section 3.1 the fixed point version of a neural network is a close estimate on the floating point thus the accuracy from floating to fixed point for the same dataset should remain very close to the original.

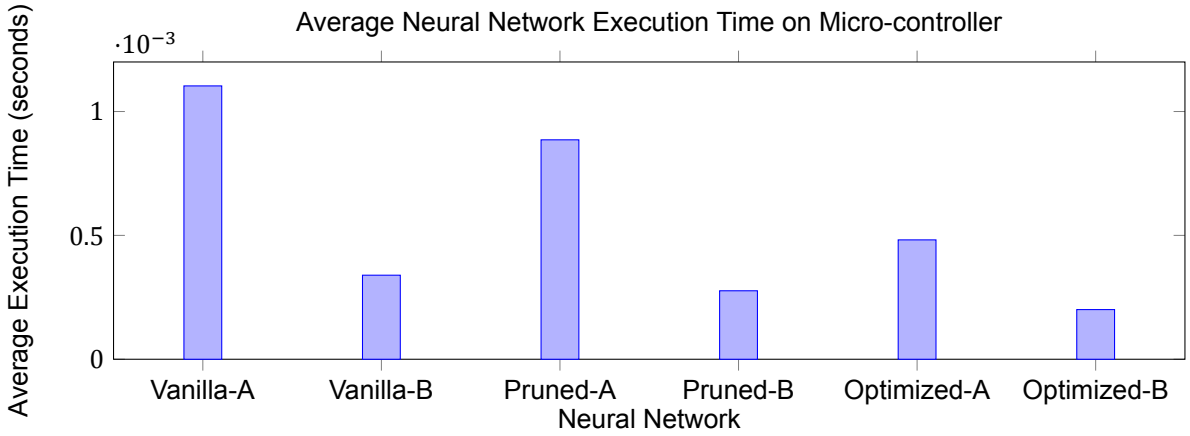


Figure 6.3: Comparison of average execution time on the micro-controller of the different neural networks and implementations.

	Vanilla-*A*	Vanilla-*B*	Pruned-*A*	Pruned-*B*	Optimized-*A*	Optimized-*B*
Average Execution Time (μs)	1103.387	339.306	885.743	276.408	481.858	200.461
Speedup Over Respective Vanilla	N/A	N/A	1.246	1.228	2.290	1.673

Table 6.3: The average execution time for the six neural networks in fixed point on the micro-controller and the speedup over the respective vanilla network.

Figure 6.2 shows the average execution time for the six neural networks examined comparing both floating point and fixed point execution. The first item of note is that across the board the fixed point calculations are faster than the floating point. This is completely expected as generally integer math requires fewer compute cycles compared to single precision and double precision floating point arithmetic.

The next noticeable trait is that the B networks require about half the execution time of the A networks and can be further seen in Table 6.2. This is mainly due to B having less than half the connections that A has as most of the calculations performed in evaluating a neural network are due to the connections. While the number of nodes in a network also plays a role, with fewer nodes requiring fewer computations.

Another characteristic from this test is that the optimized networks are quickest with a speedup of $1.618x$ and $1.190x$ for floating point and $1.708x$ and $1.192x$ for fixed point with respect to the vanilla A and B. Second is the pruned networks with speedups of $1.269x$ and $1.183x$ for floating point and $1.370x$ and $1.164x$ for fixed point over the respective vanilla networks.

These results also show that A optimized network was able to gain more of an advantage compared to the B optimized network as the less connected network, A contains relatively fewer nodes in the dependency sets for each output. However in the grand-scheme of execution time the B networks still perform quicker.

6.2.2. Micro-controller Fixed Point

The final experiment was to run the neural networks on an embedded platform to see if the results still held up. The embedded device contained an ARM Cortex M4 processor running at $120MHz$ and the timing was taken from the cycle count register of the Data and Watch Trace peripheral (DWT) and is thus cycle accurate.

Figure 6.3 shows the average execution time for the different neural networks and follows a similar spread to the results from the previous experiments in Figure 6.2 with a different

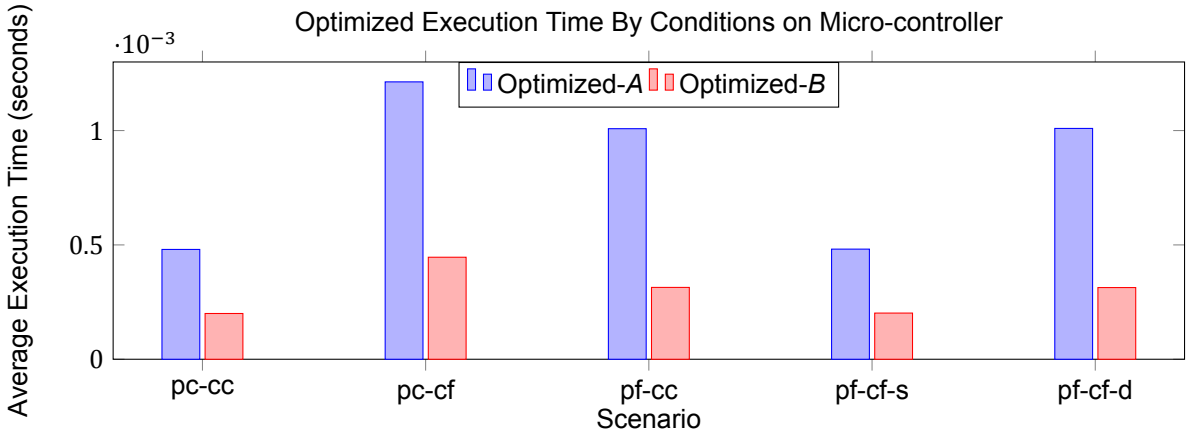


Figure 6.4: Graphical comparison of average execution times of differing scenarios of the optimization between two neural networks on the micro-controller. **pc-cc** -> Prediction Correct, Calculation Correct; **pc-cf** -> Prediction Correct, Calculation Failed; **pf-cc** -> Prediction Failed, Calculation Correct; **pf-cf-s** -> Prediction Failed, Calculation Failed, Prediction and Calculation are Same; **pf-cf-d** -> Prediction Failed, Calculation Failed, Prediction and Calculation are Different

	pc-cc	pc-cf	pf-cc	pf-cf-s	pf-cf-d
Optimized-*A* (μs)	480.244	1212.967	1008.265	481.595	1009.406
Optimized-*B* (μs)	200.099	446.133	314.178	201.934	313.515

Table 6.4: Table comparison of average execution times of differing scenarios of the optimization between two neural networks on the micro-controller. **pc-cc** -> Prediction Correct, Calculation Correct; **pc-cf** -> Prediction Correct, Calculation Failed; **pf-cc** -> Prediction Failed, Calculation Correct; **pf-cf-s** -> Prediction Failed, Calculation Failed, Prediction and Calculation are Same; **pf-cf-d** -> Prediction Failed, Calculation Failed, Prediction and Calculation are Different

scale. Numerical values for the execution time and speedup over the vanilla networks can be found in Table 6.3. Once again the optimized execution for both the A and B networks is quickest with the pruned networks coming in second.

More in-depth timing was performed on the optimized networks to explore the distribution of execution time across five different conditions. These are when the prediction is correct and the calculation is also correct (**pc-cc**), the prediction is correct but the calculation fails (**pc-cf**), the prediction fails but the calculations is correct (**pf-cc**), the prediction fails and the calculation fails and both the prediction and calculation are the same output (**pf-cf-s**), and the prediction fails, the calculations fails, but the prediction and calculation are different outputs (**pf-cf-d**). These scenarios are interesting as they give further insight into the performance of the optimization. Figure 6.4 and Table 6.4 show the results of this experiment.

If the predictive threshold is not met then the remaining nodes must be computed and additional calculations need to be made in order to provide a complete output. Thus when the prediction is different from the calculated result, as in **pc-cf**, **pf-cc**, and **pf-cf-d**, the execution time is greater. While in **pc-cc** and **pf-cf-s** the prediction and calculation are the same so the prediction threshold has very likely been met and the execution time significantly is less. Therefore in the interest of obtaining the maximum performance from the optimization the ideal scenarios are **pc-cc** and **pf-cf-s**, while keeping the occurrences of **pc-cf**, **pf-cc**, and **pf-cf-d** to a minimum. For the sake of maintaining accurate results, **pc-cc** and **pf-cc** are preferred.

6.3. Discussion of results

From these experiments some conclusions can be drawn using the vanilla neural networks as a baseline to improve upon. Ensemble neural networks appear to be the worst method tested. It had a poorer combination of execution speed and accuracy than any of the other techniques tested, hardly improving the execution time over the vanilla neural network. The

pruned networks were able to find some speedup, at the cost of accuracy. Though through pruning they also reduce the memory requirements needed to run them which can be a major advantage in reducing costs. The optimization retains the accuracy of the vanilla neural networks and has the quickest average execution time, though this is dependent on the accuracy of the prediction and the threshold. In the worst case the optimization only requires a small computational overhead of around 3−4%. Thus for posture classification using Momo Medical's sensor plate, from the options explored, the optimization shows positive results and good potential.

7

Conclusion

Based on the experiments tested and presented in chapter 6, an answer can be provided for the questions at the core of this thesis. With a sufficiently large labeled dataset, I am confident that neural networks can be trained to accurately classify posture from the sensor data provided by Momo Medical's sensor-plate using a low-cost and low-powered micro-controller. Also, the optimization proposed does reduce the average execution computational time of a neural network without any deterioration to the accuracy of the classification by utilizing the predictability of the fairly static data.

Looking beyond the posture classification problem presented by Momo Medical the optimization proposed quickly loses value. While there are numerous other problems this optimization could also be applied to, such as predictive maintenance monitoring of machinery, in a more general investigation of neural networks for embedded platforms, of the methods tested, pruning would clearly have the most advantages. While reducing the execution time of evaluating the neural network, pruning also reduces the size of the network in memory allowing for lower cost hardware in more generalized applications of neural networks.

Finally a conclusion directed towards Momo Medical. While neural networks are a valid technique to classify postures from the sensor data, I believe that the time spent gathering and labeling a large dataset and training neural networks is better put towards improving the capabilities of the sensor system and investigating other, simpler classification methods. Because the other tasks performed by the embedded platform are fairly simple, a neural network would take the largest overhead for such a system. Thus if another method can be found that moves away from the paradigm of neural networks, it would significantly reduce the computational requirements of the embedded platform and allow migrating to a lower cost and lower power platform.

7.1. Future Work

As the proposed optimization has novelty, there are a number of parameters that can be explored further. First, applying the optimization to the pruned networks could yield further reductions in both execution time and network size. Second, investigating the number of previous samples the prediction is based on would affect the stability and may skew the average execution time depending on the original accuracy of the neural network. Perhaps in coordination with the prediction, changing the threshold value with a less precise neural network could also be examined. Next, a waterfall style selection of predictions could be implemented for problems with larger number of classification outputs, such that if the first prediction fails then the optimization would only compute the nodes necessary for the second prediction rather than the remainder of the network and so on for further predictions. Finally, including additional postures, for example those in Figure 2.1 like 30° left, 30° right, 30° or 60° seated, or stomach could be attempted though some of the postures may be easier to classify, like sitting, using the accelerometer and a threshold.

Bibliography

[1] 4INSPIRATION, 2018. URL http://4inspiration.ca/pressure-ulcers/.

[2] Mohammad Babaeizadeh, Paris Smaragdis, and Roy H. Campbell. Noiseout: A simple way to prune neural networks. *ArXiv*, 2016.

[3] Jonathan Binas, Daniel Neil, Giacomo Indiveri, Shih-Chii Liu, and Michael Pfeiffer. Precise deep neural network computation on imprecise low-power analog hardware. *ArXiv*, 2016.

[4] Andrey Bondarenko, Arkady Borisov, and Ludmila Alekseeva. Neurons vs weights pruning in artificial neural networks. *Environment. Technology. Resources. Proceedings of the International Scientific and Practical Conference*, 3:22, jun 2015. doi: 10.17770/etr2015vol3.166.

[5] Helmut B□lcskei, Philipp Grohs, Gitta Kutyniok, and Philipp Petersen. Optimal approximation with sparsely connected deep neural networks. *ArXiv*, 2017.

[6] Shiyu Chang, Yang Zhang, Wei Han, Mo Yu, Xiaoxiao Guo, Wei Tan, Xiaodong Cui, Michael Witbrock, Mark Hasegawa-Johnson, and Thomas S. Huang. Dilated recurrent neural networks. *NIPS*, 2017.

[7] Minmin Chen. Minimalrnn: Toward more interpretable and trainable recurrent neural networks. *NIPS*, 2017.

[8] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *NIPS*, 2016.

[9] P. Erdős and A Rényi. On the evolution of random graphs. In *PUBLICATION OF THE MATHEMATICAL INSTITUTE OF THE HUNGARIAN ACADEMY OF SCIENCES*, pages 17–61, 1960.

[10] Maximilian Golub, Guy Lemieux, and Mieszko Lis. Dropback: Continuous pruning during training. *ArXiv*, 2018.

[11] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *NIPS*, 2015.

[12] Mehrdad Heydarzadeh, Mehrdad Nourani, and Sarah Ostadabbas. In-bed posture classification using deep autoencoders. In *2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society (EMBC)*. IEEE, aug 2016. doi: 10.1109/embc.2016.7591565.

[13] Chi-Chun Hsia, Yu-Wei Hung, Yu-Hsien Chiu, and Chia-Hao Kang. Bayesian classification for bed posture detection based on kurtosis and skewness estimation. In *HealthCom 2008 - 10th International Conference on e-health Networking, Applications and Services*. IEEE, jul 2008. doi: 10.1109/health.2008.4600129.

[14] *An Empirical Exploration of Recurrent Network Architectures*, volume 37, 2015. ICML.

[15] Herbert Jaeger. A tutorial on training recurrent neural networks, covering bppt, rtrl, ekf and the ”echo state network” approach. Technical report, Fraunhofer Institute for Autonomous Intelligent Systems (AIS), December 2013. URL http://minds.jacobs-university.de/uploads/papers/ESNTutorialRev.pdf.

[16] Shyam Jagannathan, Mihir Mody, and Manu Mathew. Optimizing convolutional neural network on DSP. In *2016 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, jan 2016. doi: 10.1109/icce.2016.7430652.

[17] Seul Jung and Sung su Kim. Hardware implementation of a real-time neural network controller with a DSP and an FPGA for nonlinear systems. *IEEE Transactions on Industrial Electronics*, 54(1):265–271, feb 2007. doi: 10.1109/tie.2006.888791.

[18] Jan Koutník, Klaus Greff, Faustino Gomez, and Jürgen Schmidhuber. A clockwork rnn. *ICML*, 2014.

[19] Liangzhen Lai, Naveen Suda, and Vikas Chandra. Deep convolutional neural network inference with floating-point weights and fixed-point activations. *ICML*, 2017.

[20] Seyed H. F. Langroudi, Tej Pandit, and Dhireesha Kudithipudi. Deep learning inference on embedded devices: Fixed-point vs posit. *ArXiv*, 2018.

[21] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[22] Dar-Shyang Lee and S.N. Srihari. A theory of classifier combination: the neural network approach. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*. IEEE Comput. Soc. Press, 1995. doi: 10.1109/icdar.1995.598940.

[23] Sam Leroux, Steven Bohez, Cedric De Boom, Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Pieter Simoens, and Bart Dhoedt. Lazy evaluation of convolutional filters. *ArXiv*, 2016.

[24] Yanick Mampaey. Prevention of pressure ulcers: A scoping review. resreport, Delft University of Technology, November 2017.

[25] Franco Manessi, Alessandro Rozza, Simone Bianco, Paolo Napoletano, and Raimondo Schettini. Automated pruning for deep neural network compression. *ArXiv*, 2017.

[26] Steffen Nissen. *Implementation of aFast Artificial Neural NetworkLibrary (fann)*. PhD thesis, University of Copenhagen, 2003. URL `http://leenissen.dk/fann/wp/2015/11/fann-in-research/`.

[27] Eriko Nurvitadhi, Suchit Subhaschandra, Guy Boudoukh, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, and Duncan Moss. Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*. ACM Press, 2017. doi: 10.1145/3020078.3021740.

[28] O. Olurotimi. Recurrent neural network training with feedforward complexity. *IEEE Transactions on Neural Networks*, 5(2):185–197, mar 1994. doi: 10.1109/72.279184.

[29] Christian W. Omlin and C. Lee Giles. Constructing deterministic finite-state automata in recurrent neural networks. *Journal of the ACM*, 43(6):937–972, nov 1996. doi: 10.1145/235809.235811.

[30] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *MLR*, 2012.

[31] Galina Rogova. Combining the results of several neural network classifiers. In *Classic Works of the Dempster-Shafer Theory of Belief Functions*, pages 683–692. Springer Berlin Heidelberg, 1994. doi: 10.1007/978-3-540-44792-4_27.

[32] Koen de Groot Sabri Boughorbel, Fons Bruekers. Pressure-sensor system for sleeping-posture classification. In *Measuring Behavior 2012*, Measuring Behavior. Philips Research Laboratories, Eindhoven, The Netherlands, August 2012.

[33] Sungho Shin, Kyuyeon Hwang, and Wonyong Sung. Fixed-point performance analysis of recurrent neural networks. *National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIP)*, 2016. doi: 10.1109/MSP.2015.2411564.

[34] Hava T. Siegelmann. RECURRENT NEURAL NETWORKS AND FINITE AUTOMATA. *Computational Intelligence*, 12(4):567–574, nov 1996. doi: 10.1111/j.1467-8640.1996.tb00277.x.

[35] Suraj Srinivas, Akshayvarun Subramanya, and R. Venkatesh Babu. Training sparse neural networks. *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2016.

[36] Yi Sun, Xiaogang Wang, and Xiaoou Tang. Sparsifying neural network connections for face recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, jun 2016. doi: 10.1109/cvpr.2016.525.

[37] Kailash Gopalakrishnan Suyog Gupta, Ankur Agrawal. Deep learning with limited numerical precision. In *32nd International Conference on Machine Learning*, 2015.

[38] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung. BranchyNet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*. IEEE, dec 2016. doi: 10.1109/icpr.2016.7900006.

[39] Tekscan. Flexiforce a301. techreport, Tekscan, 2018. URL `https://www.tekscan.com/products-solutions/force-sensors/a301`.

[40] Pete Warden. Why are eight bits enough for deep neural networks? Online, May 2015. URL `https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/`.

[41] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *NIPS*, 2016.

[42] M. Yip, D.D. He, E. Winokur, A.G. Balderrama, R. Sheridan, and Hongshen Ma. A flexible pressure monitoring system for pressure ulcer prevention. In *2009 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, sep 2009. doi: 10.1109/iembs.2009.5333964.

[43] R. Yousefi, S. Ostadabbas, M. Faezipour, M. Farshbaf, M. Nourani, L. Tamil, and M. Pompeo. Bed posture classification for pressure ulcer prevention. In *2011 Annual International Conference of the IEEE Engineering in Medicine and Biology Society*. IEEE, aug 2011. doi: 10.1109/iembs.2011.6091813.

[44] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '15*. ACM Press, 2015. doi: 10.1145/2684746.2689060.

[45] Zhi-Hua Zhou, Jianxin Wu, and Wei Tang. Ensembling neural networks: Many could be better than all. *Artificial Intelligence*, 137(1-2):239–263, may 2002. doi: 10.1016/s0004-3702(02)00190-x.