MASTER THESIS

Exploring the effect of merging techniques on the performance of merged sparse neural networks in a highly distributed setting

Emiel Steerneman

Faculty of Electrical Engineering,
Mathematics and Computer Science

EXAMINATION COMMITTEE
dr.ir. D.C. Mocanu
dr.ing. G. Englebienne
dr. M. Poel

Enschede
July 2022

UNIVERSITY OF TWENTE.

Contents

Glossary 3

Abstract 4

1 Introduction 5

2 Background and Related Work 9
2.1 History of Artificial Neural Networks 9
2.2 Sparse Neural Networks 10
2.3 Parallelization 14
2.4 NNSTD 17

3 Expected NNSTD between two SNNs 18

4 Proposed Merging Methodology 20
4.1 Merging Methods 20
4.2 Merging method similarity 21
4.3 Possible merging issues 21

5 Data and network architecture 24
5.1 Data 24
5.2 Network architecture 25

6 Phase 1 26
6.1 Methodology 26
6.2 Data 26
6.3 Network architecture 26
6.4 Hyperparameters 26
6.5 Training performance 27
6.6 Performance after merging 28
6.7 Summary 29

7 Phase 2 30
7.1 Phase 1 to phase 2 30
7.2 Methodology 30
7.3 Data 31
7.4 Network architecture 31
7.5 Hyperparameters 31
7.6 Training Results 32
7.7 Merging Results 37
7.8 Performance and NNSTD-original 41

8 Phase 2 - Extended research 51
8.1 Methodology 52
8.2 Training results 52
8.3 Merging results with two networks 55
8.4 Merging results with five networks 55
8.5 Conclusions extended research 59

9 Discussion and thoughts **61**
9.1 Importance of bias merging method 61
9.2 Overall best merging method 61
9.3 Sparsity level 0.99 61
9.4 Parallelizing an optimized version of Dropout 61

10 Future Work **62**
10.1 Phase 1 : Classification preference and data distribution 62
10.2 Training a merged SNN 63
10.3 Impact of bias sparsification on network performance 63
10.4 Performance of different merging techniques 63
10.5 Apply topology transformation before merging 64
10.6 Phase 2 : Improve resparsification by including Sparse Connectivity Pattern 64
10.7 Phase 2 extended research : Exploiting overfitting 64

11 Appendix **66**
11.1 Reproducibility considerations and code bugs 66
11.2 Phase 1 66

Glossary

ANN Artifical Neural Network. This encompasses both DNN and SNN

DNN Dense Neural Network. An ANN with all weights present

NNSTD Neural Network Sparse Topology Distance (see page 17)

SNN Sparse Neural Network. An ANN with weights missing

Sparsity The fraction of weights missing from an SNN

Abstract

With the number of parameters of modern neural networks ranging in the billions, training is only feasible in a highly parallel environment. On the other hand, sparse neural networks can significantly reduce the number of parameters of dense neural networks, thereby memory usage and computational costs. To keep the memory usage and computational costs of sparse neural networks low, these should be sparse throughout the training process. This can be accomplished by algorithms such as SET, which train sparse neural networks from scratch. SET achieves state-of-the-art performance by evolving sparse neural network structures throughout training. This evolution prevents the use of conventional parallelization training techniques in decentralized settings. This research evaluates techniques that may enable the training of sparse neural networks in a parallel decentralized setting. Related work suggests that merging sparse neural networks should boost performance due to the bias-variance tradeoff. Evaluation of these techniques shows that merging sparse neural networks based on the magnitude of their parameters gives the best results. Resparsification of resulting neural networks ensure that memory usage and computational costs stay constant. Under the circumstances of this research, sparse neural networks have been successfully merged both with and without incurring loss in performance. These results, combined with the SET algorithm, strengthen the idea that parameter magnitude is an essential factor in sparse neural networks. Using these techniques, conventional parallelization techniques can once again be applied. This research provides a basis for merging sparse neural networks with different structures. Combining sparse neural networks that evolve with parallel training in a decentralized setting allows many low-performance edge devices to train a single network.

1 Introduction

Artificial Neural Network (ANN) [6] is a term that most have heard at least once at some point in the past few years. ANNs can be found in phones in our pockets, in cars on the road, and behind the ads seen on websites. Pharmaceutical companies use ANNs to create the latest medicines. Business use ANNs to gain advantages in the stock market. Meteorologists apply ANNs to create weather forecasts. While not everyone may realize it, they are everywhere around us. Although ANN usage growth has predominantly occurred in the last decade, the concept has existed for over half a century.

The concept of an Artificial Neural Network was first brought up in 1943, in the paper of Mcculloch and Pitts [24]. Since then, research has been accumulated, leading to the current state of ANNs. A few significant events come into view whenever one looks at the history of ANNs. OpenAI has broken down the force driving the advance of AI into three distinct factors; Algorithmic innovation, data, and the power of hardware [9]. Section 2.1 will provide a short historic overview of these factors.

Where we are now Machine learning has seen exponential growth in network complexity, data volume, and hardware performance over the last few decades. The computational power needed to train state-of-the-art networks is increasing ever so fast. Between 2012 and 2018, OpenAI reported a 300,000-fold increase in computation needed to train state-of-the-art networks [9]. In June 2020, OpenAI released the GPT-3 network, which uses deep learning to produce human-like text. The description given by OpenAI boggles the mind with numbers that are too large to comprehend;

"With GPT-3, the number of parameters has swelled to 175 billion, making GPT-3 the biggest neural network the world has ever seen ... The total compute cycles required [to train] is the equivalent of running one thousand trillion floating-point operations per second per day for 3,640 days ... estimated that it would take a single GPU 355 years to run that much compute, which, at a standard cloud GPU instance price, would cost $4.6 million ... GPT-3's 175 billion parameters require 700GB, 10 times more than the memory on a single GPU." [41]

I take this opportunity to point the reader to a website[1] that hosts a GPT-3 instance capable of generating interactive text-based adventures.

Where can we go The training of this enormous ANN has been made possible by massive parallelization. Still, budget and time constraints can be limiting factors for ANNs. If reducing the parameter count of an ANN without significantly reducing its performance would be possible, then that would certainly push the boundaries of these ANNs. Fortunately, it is, through Sparse Neural Networks (SNN). How an SNN works will be explained in the following paragraph. First, the possible benefits will be listed. The reduced strain placed on the hardware by SNNs brings freedom to follow one or more of the following directions.

Increased complexity SNNs can be grown to require maximum performance from current hardware once again. The increased complexity allows for better results and tackling more complex problems.

[1] https://play.aidungeon.io

Increased inference speed Keeping SNNs on the same hardware allows for faster inference speed. This will benefit especially applications that have a (soft) real-time component. Examples of this are self-driving cars and translation software. I would like to point the reader to an exciting application from NVidia [2]. In this application, faces from a video call are compressed on the sender side and reconstructed on the receiver side using ANNs. This technique reduces data transfer up to a factor of 10 compared to the popular encoding standard h264. SNNs could allow this technique to run faster and consume less battery on mobile devices.

Reduced hardware requirements The reduced strain on hardware allows for cheaper hardware that is more accessible to the everyday consumer. Typically, data is sent from consumer hardware to specialized inference servers, where data is put through an ANN, and the results are sent back to the consumer [8]. Examples are Apple's assistant Siri and Google's translation application Lens. Being able to run ANNs on cheap devices would allow users to apply these without an active internet connection. This could be used in, for example, medical wearables and areas without (stable) internet connections, such as developing countries.

Sparse neural networks A Sparse Neural Network (SNN) is an Artificial Neural Network (ANN) in which not all weights are present. This is in contrast to a Dense Neural Network (DNN), in which all weights are present. The fraction of missing weights is called *sparsity*. The first SNN was introduced by Mozer and Smolensky in 1989 [30]. Their purpose was to measure the impact of a single weight on the performance of a network, by measuring this performance both before and after the weight was removed. Mozer and Smolensky also recognized the performance benefits an SNN brings to the table. SNNs can be obtained by pruning a DNN, called dense-to-sparse training. Another approach is to train a network that is sparse from the start, called sparse-to-sparse training. Sparse-to-sparse training has gained traction in the past few years. The Lottery Ticket hypothesis [13] showed that there exist sparse topologies which can successfully be trained. Since then, approaches have been developed in an attempt to find these topologies. A noteworthy contribution is the SET algorithm [25], a sparse-to-sparse training method that optimizes the topology throughout training. Still, the performance benefits that an SNN can bring are inherently limited to the speed of the hardware it runs on. As the problems we want to solve become more complex, more computational power will be needed to train an SNN. Parallelization can offer an answer to this obstacle.

Parallelization Parallelization can be applied to both the training and inference of an ANN. In training, multiple computers can train an ANN on different parts of a dataset. The resulting gradients of all computers can be merged at a single location, where the weights of the ANN can be updated. In inference, an ANN can be split up into multiple parts. Multiple computers can run different parts of the network, resulting in a reduction in memory usage or a speedup in inference time, depending on how the network is split up. Parallel training and inference are techniques widely used by large companies such as Amazon [3], Google [40], and Microsoft [10].

Motivation All the existing parallel training techniques come with a downside, however. The topology of the ANN that is trained can not be modified while multiple

[2]https://nvlabs.github.io/face-vid2vid/

computers are training it. Gradients or weights of networks with different architectures can currently not be merged together. If modifications are being made to the network topology, which can be the case with e.g. the SET algorithm, this will have to be done on a central location. When modifications to the network happen often, this can incur significant overhead since each computer has to communicate with this central location. On top of that, unless networks are neatly sent to the central location and merged one by one, gradients of multiple machines must be kept in memory at the central location. These are not a problem if bandwidth and memory are not a limiting factors. However, since SNNs aim to address these, we can reasonably assume that they are a limiting factor for a user training an SNN.

The different topologies generated will have to be merged back together to form a single SNN. Unfortunately, there is little to no research on merging networks with different topologies. Solving this problem could lead to better scalability, more extensive networks, faster training, and thus the ability to solve more complex problems.

Goal of the research The goal of the research is to further the usage and development of SNNs, both in practical terms and research areas, by further developing the techniques by which SNNs can be trained.

Research Questions

- How can two SNNs with different topologies be merged together?
- How do merging methods impact the performance of a merged SNN compared to its parents?
- How well does a merged SNN retain performance from two SNNs trained on different datasets?

Methodology The research questions have been divided into two phases, which are, in essence, two smaller instances of research. Both come with their own datasets, hyperparameters, methodology, results, and conclusions. Since phase 2 is a continuation of phase 1, there is a large overlap between the datasets, hyperparameters, and methodology. This will be elaborated further in section 'Data and network architecture' on page 24 and in the phases' respective sections.

Methodology Phase 1 To explore the capability of merged SNNs to capture the most important weights, the 'essence', of its parents, networks have been trained on non-overlapping Fashion-MNIST data subsets and merged. The performance of the merged network is then evaluated on both parents' datasets and their combined dataset. This experiment is done with multiple merging methods and at two sparsity levels. Phase 1 can be found on page 26.

Methodology Phase 2 To explore the capability of merged SNNs to retain the parents' performance, a large amount of SNNs have been trained, evaluated, and merged in pairs. Subsets of the Fashion-MNIST dataset have been used for training and evaluation. 12 sparsity levels have been selected, ranging from 0.00 (all weights present) to 0.99 (missing 99% of all weights). For each sparsity level, networks have been trained and paired. Each pair has been merged using one of the 5x5 proposed merging methods (page 20). The

performances of the merged SNNs are evaluated on the evaluation subset of Fashion-MNIST. Next to these metrics, the distances between the merged SNNs and their original SNN pairs have been measured using the NNSTD metric (page 17). These results are then used to analyse the effectiveness of each proposed merging method. Phase 2 can be found on page 30.

Contributions Results from phase 1 show that merging two SNNs trained on two different datasets can lead to a merged SNN that can predict from the aggregate of both datasets. However, performance is lacking compared to an SNN trained directly on the aggregate dataset. Results from phase 2 show that merging two SNNs trained on the same dataset can also lead to a merged SNN that can predict on that dataset with decent performance. In some cases, accuracy loss is almost negligible. Both results indicate that the merging techniques can capture the discriminative essence of both SNNs and place these into a single SNN. Performance of the merged SNN is, however, strongly dependent on the merging technique used, indicating that some techniques better capture the essence of both SNNs than others. Results from phase 2 extended research show that normalization helps models stabilize at lower sparsity levels, as well as an improvement in performance. While related work suggests that merging multiple neural networks should result in a performance boost, empirical results have not yet fortified this claim.

Thesis overview The structure of this thesis will now be summarized. It will start with a concise history of ANNs. After that, background information will be provided about SNNs and parallelization. This background information should be sufficient to grasp both subjects' core concept and understand the methodology and conclusions of phases 1 & 2. References will be provided in case the reader wants to acquire more knowledge before proceeding. After the introduction to SNNs and parallelization, the work related to this thesis will be provided and discussed. There is little work concerning the direct merging of multiple ANNs, so this section might be shorter than what is normally expected. The thesis will then state the proposed methods by which to merge SNNs, as well as the datasets and networks used by phases 1 & 2. Note that phases 1 & 2 do slightly differ in how they use the datasets. Everything up until here has been preparation for phases 1 & 2, which follow right after. Between the sections of phases 1 and 2, there will be an explanation of how phase 1 influenced phase 2. Phase 2 is extended with additional research into the impact of normalization on networks and their merging, as well as the merging of more than two models. After phase 2, results and conclusions from both phases will be analyzed together to reach the final conclusion. This will be discussed, and finally, possible future work will be described.

2 Background and Related Work

2.1 History of Artificial Neural Networks

The beginning. In 1958, Frank Rosenblatt published a paper describing the perceptron [32], the building block for ANNs. He used the idea of the perceptron, first brought up in 1943 by Mcculloch and Pitts [24], to create a machine capable of recognizing simple geometric shapes in 20x20 images. The machine was able to learn by changing the weights of connections. In the machine, these weights were actualized by rotary potentiometers. Rosenblatt demonstrated that the perceptron brought intelligent machines into the realm of possibility. He received strong criticism, however, from Minsky. Minsky, together with his colleague Papert, argued that the perceptron was too simple to model complex real-world phenomena. This was because backpropagation was not discovered yet, networks could not yet model non-linearity, and the largest networks at the time had no more than two layers. Even if the ideas to build networks with more layers were there, the hardware of the time was insufficient. These limitations led to an influential book named *Perceptrons: an introduction to computational geometry* [33] by Minsky and Papert. The book contributed to the onset of the *AI Winter*, the period between 1970-1980 when research on the topic of ANNs was almost non-existent.

The uprising. The AI winter lasted until around 1980. At that time, the improvement of hardware and the large amount of data provided by the rise of the internet sparked a surge that led to many discoveries.

1986 The interest into ANNs was revived with the rediscovery of backpropagation and the paper *Learning representations by back-propagating errors*[33]. In 1989, the development of Q-learning greatly improved the feasibility of Reinforcement Learning.

1997 LSTMs were discovered, paving the way for tackling temporal problems such as speech recognition.

1998 One of the earliest convolution ANNs was developed, LeNet-5. It consisted of 2 convolutional layers, 3 fully connected layers, and a total of roughly 60,000 parameters [2]. The network was capable of recognizing handwritten digits.

1999 NVidia released the GeForce 256, which they dubbed the "First Graphics Card". Graphics cards would not find widespread use in machine learning until 2012.

2000-now Open source datasets are being created, such as MNIST, COCO, and ImageNet.

2006 Deep Belief Networks are invented by Geoffrey Hinton and colleagues [15]. In contrast to the discriminative ANNs, Deep Belief Networks are generative models.

2009 ImageNet reaches 3 million images.

2010 ImageNet reaches 10 million images.

The era of GPUs. *2012* ImageNet reaches 14 million images. AlexNet made a breakthrough in image recognition, improving state-of-the-art accuracy by more than 10%. This improvement was made possible by creating a relatively deep network. It consisted of 5 convolutional layers, 3 fully connected layers, and a total of roughly 60 million parameters [2]. The network was computationally expensive, but training was made possible

with the use of GPUs. Before 2012, using GPUs for machine learning was uncommon [9]. Since 2012, NVidia's stock price has increased roughly 18-fold.

2014 Facebook made a breakthrough in facial recognition with *DeepFace*, improving state-of-the-art accuracy by more than 27%. GANs are invented.

2015 Deepmind releases AlphaGo. It defeated the world champion of the board game Go, which is considered to be one of the most complex abstract strategy games.

2.2 Sparse Neural Networks

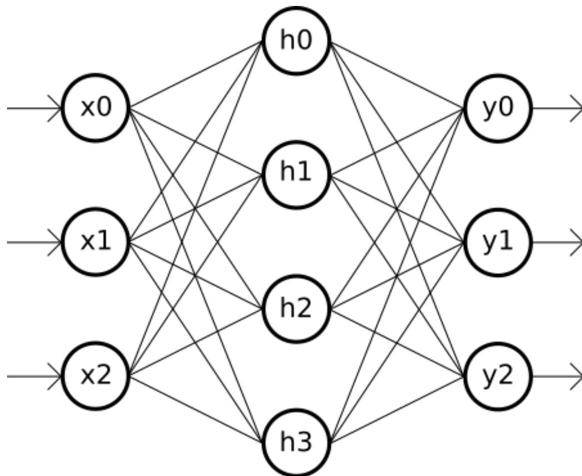


Figure 1: Dense (DNN)

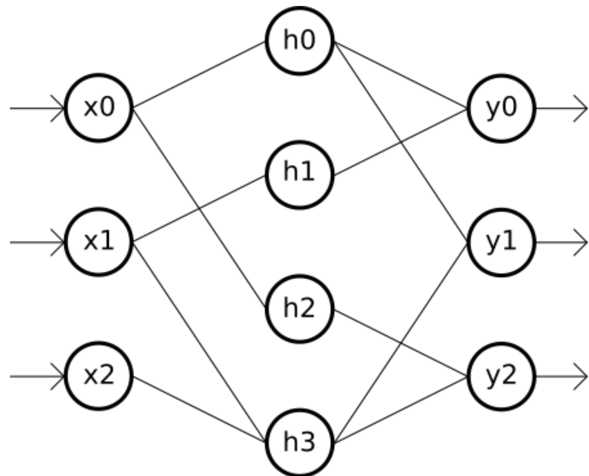


Figure 2: Sparse (SNN)

The introduction1 provided a historical overview of Artificial Neural Networks (ANN) and a concise introduction into Sparse Neural Networks (SNN). All of the ANNs described, and most of the ANNs used today, are dense. Within a dense neural network (DNN), all neurons in a given layer l^{-1} are connected to all neurons in layer l through weights, as illustrated in Figure 1. For more background information on DNNs, I'd like to point the reader to the book *Pattern Recognition and Machine Learning* by Christopher Bishop. With an SNN, not all of these weights are present, as illustrated in Figure 2. The fraction of weights not present in an SNN is called the *sparsity*. An SNN with a sparsity of 0.99 will only have $\frac{1}{100}$ of the weights of its dense counterpart.

Having an ANN where not all weights are present was first proposed by Mozer and Smolensky [30]. Researchers at the time were trying to understand why ANNs worked the way they do, as cited by Mozer and Smolensky. Techniques such as Principal Component Analysis [12] and weight decaying were used to determine which parts of an ANN had the most influence on performance. Mozer and Smolensky opted for another approach in which weights were completely removed from an ANN. The performance loss of the ANN was directly correlated to the importance of the removed weight. Thus, the first SNN was created out of a desire to better understand ANNs, not out of a need to increase performance and scalability.

2.2.1 Advantages of SNN over DNN

SNNs bring advantages over DNNs. They require fewer calculations, less memory, and are less prone to overfitting [23]. This is explained in the paragraphs below.

Calculation reduction. The calculations within a neural network consist primarily of the multiplication of weights and values. For example, the popular image recognition network AlexNet consists of around 650,000 neurons and 600 million weights [1]. The fully connected part of AlexNet consists of 9192 neurons (N_n) and around 60 million weights (N_w). Each weight adds a calculation, namely the multiplication of a value and that weight. Each neuron adds a few calculations through the summation of its inputs, and its activation function. A rough estimate of the number of calculations in the fully connected part of AlexNet then becomes $2N_w$ (one multiplication, one summation) + N_n (one activation function) for a total of roughly 120 million calculations. Within AlexNet, the weights are responsible for $\frac{2N_w}{2N_w+N_n} \approx 99.9\%$ of the calculations. It follows that reducing the number of weights roughly scales linearly with reducing the number of calculations. In theory, an SNN with a sparsity of 0.99 would be 100x as fast as its dense version. Reality doesn't exactly match the theory in this case. For one, extra calculations such as bias and possible batch normalization have been omitted from the example. More importantly, the current technology heavily favours dense matrices, especially hardware. GPUs are made for dense matrix calculations, and libraries are optimized for it.

Memory reduction. Just as removing connections reduces the number of calculations needed, it also reduces the memory footprint. A connection within a network is represented by its location within the network and its weight. Within a DNN, the location is implicit, relying on its location in memory. In an SNN, these locations have to be stored explicitly. Next to that, an SNN also needs to explicitly store the neurons that are connected. In dense networks, this is once again implicit, since a neuron is connected to all neurons in the next layer. Therefore, SNNs bring some overhead that is not present in DNNs. This means that memory usage is only reduced when the sparsity level is below a certain threshold.

Multiple formats have been developed to store sparse matrices, each with their own benefits and drawbacks, and it is converted between different formats for different tasks. For example, when creating a sparse matrix, one could use the LIL-format [35]. This format has the advantage that changes to the matrix sparsity structure are efficient. After creating the sparse matrix, one could convert it to the CSR-format [34]. This format does not efficiently support changes to the structure, but it does support efficient arithmetic operations such as multiplication. More information on formats for sparse matrices can be found in section 3.1 of [19].

Note that not all implementations of SNNs will apply these formats. Often, researchers will use a dense connection matrix and apply to it a binary mask. The mask sets all inactive connections to zero, effectively disabling them. This approach does obviously not reduce memory usage, but applying it to existing DNN implementations is straightforward. The reason for not using a true SNN is that the hardware, software, and algorithmic support is insufficient. Yu et al. [43] showed that a 89% sparse AlexNet is 25% slower on a CPU compared to its fully dense counterpart on a GPU. Popular libraries such as BLAS, MKL, and cuSPARSE originate from the field of linear algebra and are optimized for a sparsity level of more than 99%, much sparser than most SNNs (50%-99%). Training pure SNNs brings extra challenges since popular formats for storing SNNs, such as CSC or COO, are not suited for back-propagation [16].

2.2.2 Training of SNNs

The following section describes the different approaches of obtaining a trained sparse neural network. Figure 3 from Mocanu et al 2021 paper [28] gives a clear overview of these approaches. These approaches can be differentiated through multiple factors, listed below.

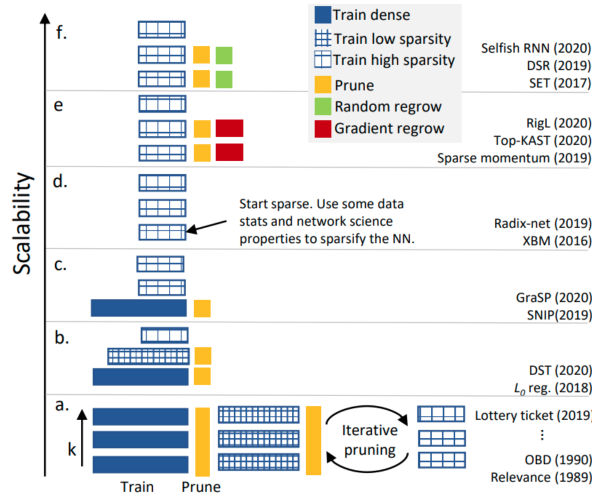


Figure 3: Schematic representation of various method types used to obtain sparse neural networks and a rough estimation of their scalability; a. Pruning, b. Simultaneously training and pruning, c. One-shot pruning, d. Sparse training (static), e. Sparse training (dynamic - gradient), f. Sparse training (dynamic - random) [28]

Difference between approaches

Initial topology. The initial topology can either be dense or sparse. If the topology is sparse, the connections can either be random, or predetermined as is the case with e.g. the Lottery Ticket Hypothesis.

Initial weights. The weights of the connections can either be random, or predetermined as is also the case with the Lottery Ticket Hypothesis.

Topology throughout training. The topology can either be fixed during training, or evolving. Topologies are evolving when connections are removed, which is the case when pruning dense networks, and when connections are removed and added, which is the case with certain approaches such as NEAT [39] and SET[25].

Approaches

The three factors given above can be combined in different approaches of obtaining a trained sparse neural network. Not all of the approaches are feasible. Different combinations are listed in Table 1. Extra information is given for *Dense-to-Sparse training*, *Ordinary SNN training*, and *Dynamic Sparse-to-Sparse training.*

Ordinary SNN training Ordinary SNN training consists of creating an SNN with random weights and connections, and not modifying the topology during training. Results in section 'Training performance' on page 27 will show that this method, at high sparsity

Initial topology	Initial weights	Topology	Approaches
Dense	random	fixed	Ordinary DNN training
Dense	random	evolving	Dense-to-Sparse training, Lottery Ticket Hypothesis [13]
Dense	predetermined	fixed	Transfer learning
Dense	predetermined	evolving	-
Sparse	random	fixed	Ordinary SNN training
Sparse	random	evolving	Dynamic Sparse-to-Sparse training (e.g. SET [25], DSR [29])
Sparse	predetermined	fixed	Lottery Ticket Hypothesis [13]
Sparse	predetermined	evolving	-

Table 1: Different approaches of obtaining a trained SNN

levels, is inferior to methods that modify the topology during training. The same results have been found by Mocanu et al. with their SET algorithm [25]. Their results show that at high sparsity levels, SNNs with fixed topology have inferior performance. Exactly this performance issue, combined with network rewiring inspiration originating in biology, motivated Mocanu et al. to design a training procedure that modifies the network topology during training. The Lottery Ticket Hypothesis implies that for a given SNN topology, there is a possibility that specific initial weights exist to properly train the SNN. However, the probability that random initial weights are good enough to properly train an SNN from scratch is nil.

Dense-to-Sparse training A common approach to creating an SNN is to train and prune a DNN. The pruning can be done once, immediately reducing the DNN to the desired sparsity level. Looking at Figure 3, the algorithms SNIP [18] and l0-regularization [21] take this approach. Another approach is to prune a DNN in steps. First, the DNN is trained, either fully or partially, and then slightly pruned. This is repeated until the desired sparsity level is reached, and is referred to as the *train-prune-retrain cycle*. This approach starts with a dense topology, and weights are randomly initialized. Looking at Figure 3. Many of these algorithms exist and date way back to 1989, when Mozer and Smolensky first applied this technique. Dense-to-Sparse training is also applied in the Lottery Ticket Hypothesis paper to create the SNN that will be retrained.

Dynamic Sparse-to-Sparse training In the Sparse-to-sparse training approach, the network starts and remains sparse during the entire training. As mentioned before, training an SNN from scratch with a random topology and random weights does most probably not produce good results. However, the Lottery Ticket Hypothesis paper has shown that combinations of topologies and weights exist that do manage to produce good results when trained. Their hypothesis is that a DNN contains many SNNs within it and that training a DNN optimizes one of these SNNs. Basically, training a DNN is akin to attempting to train numerous amounts of SNNs. Whereas training a DNN trains all these SNNs within itself simultaneously, Dynamic Sparse-to-Sparse training tries out multiple topologies over time, one after another. Throughout the training, the topology of the SNN is modified by deactivating some weights and activating others. The first algorithm that dynamically changes the network during training is SET [25]. Since SET, multiple Dynamic Sparse-to-Sparse training algorithms have been developed, such as DeepR, DSR, and RigL. There are multiple metrics to choose which connections to deactivate

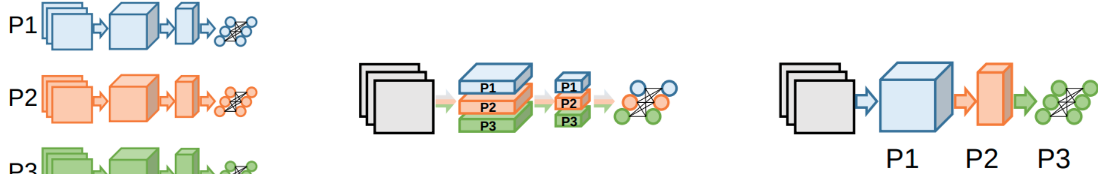


Figure 4: Data parallelism, model parallelism, and layer parallelism. [4]

and activate, as can be seen in Figure 3. SET deactivates connections based on their magnitude and deactivates a number of connections that have their weights closest to zero. DSR extends this by having multiple different thresholds that are trainable. RigL [11] activates connections based on their gradient. Note that this requires tracking the gradients of all connections, including the deactivated ones. Therefore, a truly sparse implementation is not possible for this algorithm.

2.3 Parallelization

Parallelization for the training and inference phases is a well-researched topic. A single machine can only provide so much computational power, and has a limited amount of memory available. Parallelization relieves these constraints, allowing for basically limitless computation power and memory. The extra available resources can be used to improve training and inference speed, increase the size of models, increase data throughput, and more. However, as with almost anything, overhead by i.e. communication between machines is responsible for diminishing returns.

2.3.1 Inference parallelism

The following section describes different approaches to how multiple machines can accommodate ANN inference. Each approach makes a different tradeoff between inference speed, memory requirements, and communication overhead. The approaches are illustrated in Figure 4.

Data parallelism Data parallelism is the most straightforward approach. Multiple machines have an instance of the ANN, and data is distributed over the machines. Each machine can run its ANN independent of the other machines. Communication consists only of transmitting data and inference results. This approach, however, requires a machine to load the entire ANN, which can strain the memory. Also, inference speed is always limited by the speed of the machine since there is no possibility for multiple machines to work together on a single data sample.

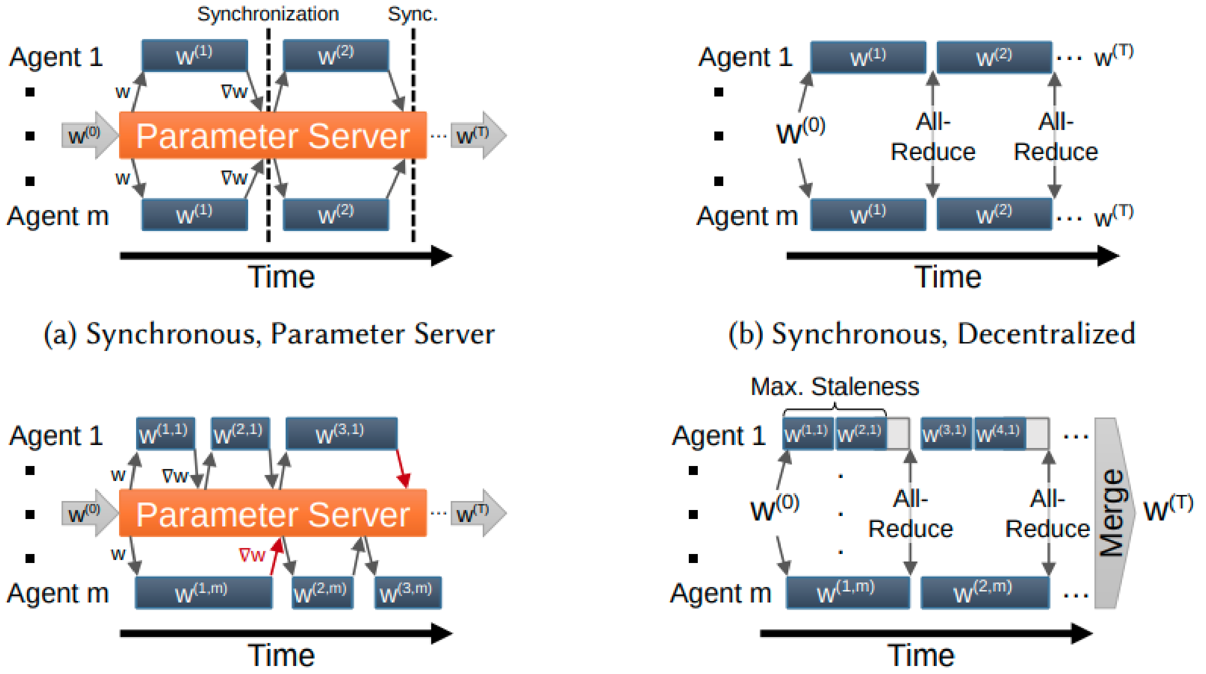
Model parallelism With this approach, the data is split over a single dimension, i.e. red, green, and blue of a colour image, and different machines use this to calculate different parts of the network. This approach reduces memory requirements since only a part of the ANN has to be stored at each machine. It also reduces inference time since multiple machines can work together, each calculating its own part. However, layers have interdependencies, and results have to be communicated between machines, adding extra overhead.

Layer parallelism With this approach, each machine is responsible for calculating a single layer of the ANN. It receives data from one other machine, puts it through its layer, and passes the results on to a single other machine. The communication overhead is equal to or less than the model parallelism approach, because interdependence is limited between machines. The memory footprint is reduced since a machine only needs to load a single layer of the ANN, instead of the entire model. However, inference speed is limited by the speed of the machine since only a single machine can work on a data sample at a time.

2.3.2 Training parallelism

Just as with inference, parallelism can aid the training of ANNs, allowing for faster training and larger models. There are however some difficulties unique to parallel training. With inference, there exists only a single version of the ANN in question, and each instance of the ANN is the same. When training on different machines however, the ANN instances are not guaranteed to stay the same. After some training, there exist multiple versions of the ANN which have to be merged into a single ANN. There is a multitude of approaches for parallel training, and the survey of Tal Ben-Nun and Torsten Hoefler [4] has categorized these into the following three categories : Model Consistency, Parameter Distribution and Communication, and Training Distribution.

Model Consistency Model consistency considers how weight updates of different machines are combined into a single network. Tal Ben-Nun and Torsten Hoefler listed four different levels of model consistency. Figure 5 gives an overview of these four levels. Regardless of the level, the machines never share actual weights between themselves. Only the weight gradients are shared, since these can simply be summed into a single gradient per weight. These gradients can then be used to update the weights. The first level is *synchronous with central server*. With this level, all machines train the same network for a single fixed duration or a fixed number of epochs on a small part of the dataset. After training, each machine sends its gradients to a central server. The central server sums the gradients, updates the network, and broadcasts the updated network to all machines. This is illustrated in Figure 5(a). The second level is *decentralized synchronous*. The difference between this level and the first level is that instead of machines sending gradients to a central server, the gradients are summed using an all-reduce operation [14]. Each machine can then update its own network. This is illustrated in Figure 5(b). The levels illustrated in Figure 5(c) and 5(d) relax the restrictions on synchronization. The third level *asynchronous with central server* allows each machine to send gradients to a central server whenever it is ready to do so. A good example of this is the HOGWILD algorithm [31]. HOGWILD is a threaded SGD [37] algorithm, where multiple threads update the same model without having any kind of thread safety. By removing the overhead of thread safety it gains a large performance boost, with the authors claiming that it sped up their performance by a factor of 100 [36]. Note that this speedup depends on a number of factors such as programming language, computer hardware, and the number of threads. With this level, gradients might have to be discarded if the model has been updated with other gradients in the meantime. HOGWILD solves this problem by applying updates to just a small part of the model, largely preventing collisions. The fourth level *decentralized asynchronous*, illustrated in 5(d), tries to find a balance between performance and synchronicity. There are predetermined points in time where the model is



(a) Synchronous, Parameter Server

(b) Synchronous, Decentralized

Figure 5: The four levels of model consistency [4].

globally updated, and between updates, machines can train as much as they want. This allows for flexibility in selecting machines. It makes it easy to add and remove machines, where not all machines have to have the same amount of performance.

Parameter Distribution and Communication Both the parameter server and the all-reduce algorithm have their benefits and drawbacks. The all-reduce algorithm can be implemented efficiently and spreads out the addition of gradients over multiple machines, whereas the parameter server has to do all additions by itself. The parameter server, in turn, allows for asynchronous training and requires less communication from each machine.

If a parameter server consists of a single machine, it forms the risk of becoming a point of congestion. It could be overloaded by the sheer amount of machines and significantly slow down the training process. Therefore, there are schemes in which a parameter server actually consists of multiple servers, or *shards*. Each shard can be responsible for a subsection of the model weights, and each machine only communicates with a single shard. This concept works well combined with *model parallelism* described in 2.3.1. Shards can be arranged in a tree-like fashion to aggregate gradients, further managing the load on the parameter server.

This structure of having multiple machines responsible for training and multiple shards responsible for updating the model can be leveraged to increase fault tolerance. Multiple machines can be assigned to train on the same data subset, and multiple shards can be made responsible for communicating with the same set of machines. The loss of a shard or machine would not have an impact on the integrity of the process. To increase fault tolerance even more, consensus [44] can be applied to detect any errors in e.g. communication or hardware.

A parameter server consisting of multiple shards also supports heterogeneity in the per-

formance of machines and the network. Performance issues in either a machine or part of the network can result in delayed results. This is turn can make the training process unstable, and prevent it from converging. Two algorithms have been designed to deal with these performance issues. The first algorithm introduces different learning rates for different machines, dealing with instability. The second algorithm only allows shards to communicate when the gradient is large enough to be deemed significant, thereby retaining convergence.

If the costs of communication are significant, there is the option to use an *inconsistent decentralized parameter update* using a gossip algorithm [17]. A gossip algorithm uses statistic probability to synchronize weights between all machines using as little communication as possible. Each machine sends its gradient to a specific number of random other machines. There is a possibility that not all machines receive the same update, but it is low enough to allow for convergence of the network regardless.

Training Distribution Previous methods all assume relatively frequent synchronization between machines. In the case that frequent updates are not possible, there exist techniques to merge networks after training has been completed. With *ensemble learning* [7], the outputs of multiple trained networks can be merged to generate a single output. The combination of multiple networks through averaging is called an *ensemble*. An ensemble can drastically increase the hardware requirements, since all networks in the ensemble have to be used. *Knowledge distillation* attempts to train a single network that mimics the behaviour of the ensemble. Essentially, the ensemble is once again brought down to the size of a single network.

2.4 NNSTD

Evolution modifies the structure of a network. One metric to quantify how different two network structures are, is the Neural Network Sparse Topology Distance (NNSTD) [20]. NNSTD measures the cost of transforming one sparse neural network structure into another sparse neural network structure. NNSTD gives a score between 0 and 1, where 0 indicates that two structures are the same (only overlapping weights), and 1 indicates that two structures are completely different (no overlapping weights). NNSTD does not look at the magnitudes of the weight, but purely if a weight exists or not. NNSTD also calculates this score by iterating over each pair of layers. It does not assume weights can traverse from one layer to another in accordance with the constraint made in the SET algorithm.

3 Expected NNSTD between two SNNs

The expected NNSTD distance between two arbitrary networks with the same sparsity can be calculated. NNSTD calculates distance based on the number of weights that do and don't overlap, and ignores weights that are deactivated in both networks. The probability that a weight is active in both networks $P_{overlap}$ equals the probability that a weight is active in one network P_1 multiplied by the probability that a weight is active in the other network P_2. This holds under the assumption that P_1 and P_2 are independent. The probability that a weight is active is inversely proportional to the sparsity S, given a uniform probability distribution for weight activation.

$$P_{overlap} = P_1 \cdot P_2 = (1-S) \cdot (1-S) = (1-S)^2$$

Given a neuron with N weights, the number of weights that are active for that neuron in both networks $N_{overlap}$ equals

$$N_{overlap} = N \cdot P_{overlap} = N \cdot (1-S)^2$$

The number of active weights that are active in either one or the other network N_{unique}, but not in both networks, equals twice the total number of active weights N_{active} minus the number of overlapping weights $N_{overlap}$.

$$N_{unique} = 2(N(1-S) - N_{overlap})$$

The expected NNSTD D for two arbitrary networks with the same sparsity is then given by the following equation.

$$\begin{aligned} D &= \frac{N_{unique}}{N_{unique} + N_{overlap}} \\ &= \frac{2(N(1-S) - N_{overlap})}{2(N(1-S) - N_{overlap}) + N_{overlap}} \\ &= \frac{2(N(1-S) - N_{overlap})}{2N(1-S) - N_{overlap}} \\ &= \frac{2(N(1-S) - N(1-S)^2)}{2N(1-S) - N(1-S)^2} \\ &= \frac{2N(1-S)S}{2N(1-S) - N(1-S)^2} \\ &= \frac{2N(1-S)S}{N(1-S)(1+S)} \end{aligned}$$

$$= \frac{2(1-S)S}{(1-S)(1+S)}$$

$$= \frac{2S}{(1+S)}$$

NNSTD distribution The NNSTD distribution follows the Central Limit Theorem. Given enough random trials, the NNSTD histogram closely follows a Gaussian distribution. The standard deviation (width) of the Gaussian is related to the size of the arrays used in the randomized trials. Larger arrays give a smaller, higher Gaussian. The values in Table 2 have been confirmed empirically. The value for sparsity level 0.00 has been omitted because NNSTD is always 0.

Sparsity level	0.10	0.20	0.30	0.40	0.50	0.60	0.70	0.80	0.90	0.95	0.99
expected NNSTD	0.18	0.33	0.46	0.57	0.67	0.75	0.82	0.89	0.95	0.97	0.99

Table 2: Expected NNSTD values for two arbitrary networks with the same sparsity

4 Proposed Merging Methodology

The techniques described in the previous section 'Parallelization' on page 14 rely on calculating gradients from a subset of the data, and merging these gradients either at a parameter server or locally at each machine through the all-reduce algorithm. This implies that all topologies stay the same during the training. These techniques can therefore not directly be applied to non-identical topologies. Training approaches which modify the structure of the ANN can choose to either apply the existing parallelization techniques and modify the topology on a central server, or find a way to merge different topologies together, thereby trying to retain the best of all networks.

The following section will discuss the proposed methods to merge multiple networks together. The merging of multiple SNNs with different topologies can be split up into two steps.

Step 1. merging weights In the case that multiple networks have a weight at a certain location i, j, these weights will have to be merged. Biases are dense, and will always have to be merged.

Step 2. pruning the newly created network Given a high enough sparsity, it is possible that two (or more) networks will have no overlapping weights. Therefore, when e.g. 3 networks are merged, a new network is created with 3 times the amount of weights. Pruning is necessary if retaining sparsity is a requirement.

4.1 Merging Methods

For step 1, five different weight merging approaches are suggested below. In each approach, three networks A, B, C with different topologies will be merged into a new network N. A position in the network i, j is given, where i indicates the layer and j indicates the weight of that layer. Given that there are 5 merging methods for both the weights and the biases, this results in 25 different ways of merging networks together. If a network does not have a weight at position i, j, the value 0 will be used.

Magnitude merging When multiple networks have a weight in the same position, the weight with the strongest magnitude will be preserved. This closely follows the SET principle, where weights with the lowest magnitude are dropped in favour of the strongest weights.

$$N_{i,j} = max_magnitude(A_{i,j}, B_{i,j}, C_{i,j})$$

Average merging With average merging, at each position in the network, the weights at that position are summed, and then divided by the number of networks.

$$N_{i,j} = (A_{i,j} + B_{i,j} + C_{i,j})/3$$

Addition merging With addition merging, at each position in the network, the weights at that position are summed. This merging method is the same as average merging, but without the division.

$$N_{i,j} = (A_{i,j} + B_{i,j} + C_{i,j})$$

Random merging This method picks a random weight from all the weights at position i, j. The networks that do not have a weight at i, j will be excluded however. This is done to prevent the value 0 from being chosen, which would further sparsity the network, and possibly disable it all together. The formula below assumes that all 3 networks A,B,C have a weight at position i, j

$$N_{i,j} = random(A_{i,j}, B_{i,j}, C_{i,j})$$

No merging This method simply discards all weights, and returns the value 0.

$$N_{i,j} = 0$$

Bias merging In contrast to the weights, the biases are not sparse. Therefore, when merging networks, biases will overlap at every position i, j. The merging methods mentioned in 4.1 work just as well for merging biases.

4.2 Merging method similarity

It is important to note that at high sparsity levels, the merging methods *magnitude*, *addition*, and *random* have a high probability of giving the same merging result. For example, take three networks A, B, C, and merge weights at the position i, j. If only network A has a weight w at position i, j, then all three merging methods mentioned above will select this weight w.

Magnitude

$$N_{i,j} = max(A_{i,j}, B_{i,j}, C_{i,j}) = max(w, 0, 0) = w$$

Addition

$$N_{i,j} = A_{i,j} + B_{i,j} + C_{i,j} = w + 0 + 0 = w$$

Random

$$N_{i,j} = random(A_{i,j}) = random(w) = w$$

4.3 Possible merging issues

Hardware Merging techniques that require sorting weights (such as with *magnitude*) might take a toll on hardware. It should not be too difficult for a few small networks. However, sorting many weights might become a problem when the number of networks increases.

Average magnitude differences For reasons relating to, e.g. different sets of training data, one network might have much stronger weights than the other. If a magnitude-based weight selection is used, this will strongly favour that one network. A possible solution could be to normalize the weights of both networks before merging them. However, these will have to be converted back to their non-normalized afterwards since nonlinear activation functions do not scale linearly with the normalized weights. Normalizing weights would affect the performance because of these nonlinear activation functions.

Equal representation of multiple networks in a memory-constrained, distributed environment Sparse Neural Networks can provide a solution to environments where memory is in short supply. As stated before, a reduction in weights almost proportionally reduces memory usage. However, when merging multiple networks, these networks will all need to be loaded in memory. This might be impossible if there is not enough memory available. Three options are then available, depending on the merging method used.

1. Merge SNNs sequentially. Given that at least two networks fit into memory, two networks can be merged into one repeatedly until all networks are merged. Indicating the merging method with the symbol $\oplus$, this can be expressed as the following formula

$$(A \oplus B) \oplus C$$

This does give issues with the merging methods *average* and *random*. Networks A and B will first be merged into AB, where both have 50% representation. Next, AB and C will be merged into ABC, resulting in a network where C will have an unfair 50% representation compared to A and B, which are both 25% represented. This goes for both *average* and *random*, since these do not have the commutative property. *Magnitude* and *addition* do have this property, and can be merged sequentially.

2. Merge SNNs in parallel. Merging SNNs in parallel will solve the commutative problem stated above. This can be expressed as the following formula

$$A \oplus B \oplus C$$

Merging multiple networks in this manner will work for all merging methods. However, this might not be possible if not all networks fit into memory. One possibility might be to load in and merge all networks weight by weight, thus reducing memory requirements. Memory is not the only possible caveat with merging SNNs in parallel. In an extremely decentralized setting, there is only ad hoc communication between nodes, where one can never be sure when and how many networks will be submitted for merging. Parallel merging requires all networks to be present. Networks coming in at a later time can only be merged in sequentially.

3. Merge SNNs sequentially with equal representation. The formula in step 1 can be modified to allow for sequential merging using *average* and *random*, while still ensuring equal representation. This is achieved by tracking the number of networks already merged and modifying the representation fraction of the next network. Given a merged network X, consisting of k merged networks, and a new network A, the formula becomes

$$(1 - \frac{1}{k})X \oplus \frac{1}{k}A$$

Where the multiplication is applied before $\oplus$. For *average*, $\frac{1}{k}$ indicates an actual number to multiply the weights with. For *random*, $\frac{1}{k}$ indicates the probability of selecting a weight from network A. Note that when resparsification is applied using either *magnitude* or *random* as the selection criteria, networks might still not have equal representation in the merged network.

5 Data and network architecture

there are two phases which basically both use the same dataset and network structure, but they do differ slightly. The similarities are explained here.

5.1 Data

Both phases make use of a (sub)set of the Fashion-MNIST dataset [42]. The Fashion-MNIST dataset consists of 70.000 grayscale images of 28x28 pixels. There are 10 classes, each class consisting of 6000 images for training and 1000 images for testing. Figure 6 displays nine samples from the dataset. The following items are present in the dataset : T-shirt/top, Trousers, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot.

The Fashion-MNIST has been created in response to the improved performance on the popular MNIST dataset. The Fashion-MNIST authors state that *"MNIST is too easy. Convolutional nets can achieve 99.7% on MNIST. Classic machine learning algorithms can also achieve 97% easily."*. A benchmark [5] on the Fashion-MNIST dataset has been done using a good deal of popular classifiers, such as Support Vector Machines, K-Means clustering, and Multi-Layered Perceptrons. Results show that most techniques achieve accuracies between 75% and 87%.

5.1.1 Justification

The Fashion-MNIST dataset has been chosen because of the performance achieved in the above-mentioned benchmark, as well as its widespread use. A quick search for "Fashion MNIST" on Google Scholar returns approximately 7000 results. It's good to have a large source of literature with which the results of the experiments can be compared, if needed.

Regarding the benchmark results; The networks that are to be trained should not reach an accuracy close to 100%. This could've been the case with the MNIST dataset. Such a high accuracy implies that the network used is too complex for the dataset, and that it has redundant parts. The most interesting merging cases are where there is no redundancy in these networks. This is because a merging method is forced to merge or choose critical parts from these networks, which must influence the performance. On the other hand, if it is tasked with merging highly redundant networks, it might happen that merging results in a network that is built out of only redundant parts, which should be useless. It might still be the case that there are large redundant parts in a network trained on Fashion-MNIST, and that there is an upper bound on the accuracy of this dataset. However, the fraction of redundancy of a network should be lower when trained on Fashion-MNIST than when trained on MNIST.

On the other hand, the dataset should not be too complex for the small networks used in the experiments. If the networks do not reach reasonable performance, the results of a merged network might be difficult to interpret. A drop in performance of the merged network might not be meaningful if the parents' performance is already low. Performance drops resulting from two well-performing networks will give more meaningful results and conclusions.

Fashion-MNIST fits the requirements for this experiment. It is widely used, allows networks to be trained to reasonable performance, and has the added benefit that it does not require a large amount of disk space. It might be interesting to compare the results

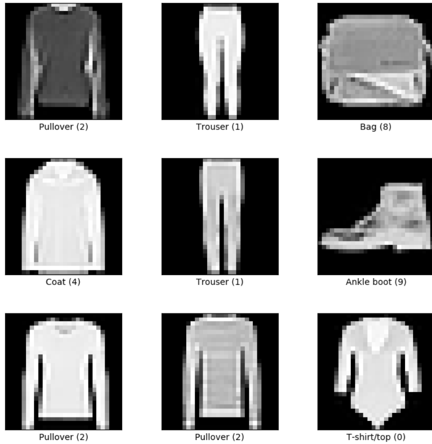


Figure 6: A sample from the Fashion-MNIST dataset.

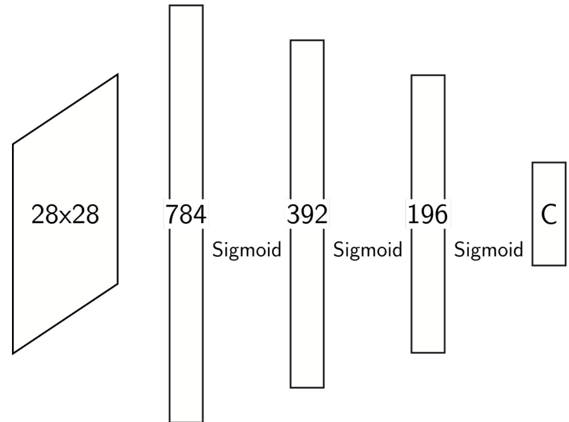


Figure 7: Network architecture used in the experiments. C indicates the number of classes, which differs per experiment.

of the experiments between Fashion-MNIST and another more complex dataset such as CIFAR-10. However, that is outside the scope of this work.

5.1.2 Dataset usage

Both phases do not use the entire dataset but use different subsets instead. Specifics will be given in the phases' respective *Data* subsections.

5.2 Network architecture

A network consists of three layers. The first layer has 784 neurons since an image of the Fashion-MNIST dataset has $28 \cdot 28 = 784$ pixels. The second layer has half the number of neurons of the first layer, 392. The third layer has half the number of neurons of the second layer, 196. The fourth and last layer has neurons equal to the number of classes. This will be specified in the respective phases. All layers are connected through the Sigmoid activation function. The network architecture is visualized in Figure 7.

6 Phase 1

Preliminary experiments have been done to show that the idea of merging sparse networks has substantiation. It also demonstrates the improvements that SET brings to the training of Sparse Neural Networks. This section will first describe the data and hyperparameters for this phase. Then, it will describe the performance of trained SNNs, as well as the performances of merged SNNs. The results will show if merging methods are able to capture the 'essences' of two SNNs and merge these into one.

6.1 Methodology

12 SNNs have been trained with different techniques, hyperparameters, and three different datasets. Datasets 1 and 2 will be mutually exclusive, and dataset 3 will be the aggregate of 1 and 2. Specific trained SNN pairs will be merged together, with one network being trained on dataset 1, and the other on dataset 2. The resulting merged network will be evaluated on all three datasets, and the results will be used to analyze the capability of the merging methods. The results should show how well the important parts of both parents have been captured and if the merged network has a strong preference for a dataset of either parent.

Important The merging method *nothing* has not been used in phase 1, since that method was conceived in phase 2. This results in a total of 16 merging methods instead of 25.

6.2 Data

Multiple subsets of the Fashion-MNIST data have been used. The first set contains classes 0 to 4. The second set contains classes 5 to 9. The third set contains all classes, 0 to 9. All subsets contain per class 5000 training samples and 1000 test samples.

6.3 Network architecture

The 12 trained networks have a density parameter epsilon of either $\epsilon = 0.1$ or $\epsilon = 0.5$. The number of weights in layer l is equal to $\epsilon \cdot (l_{width} + l_{height})$. For example, the weight matrix of the first layer has a width of 784 and a height of 392. This, given $\epsilon = 0.1$, gives $0.1 \cdot (784 + 392) = 118$ active weights. With a total of $784 \cdot 392 = 307328$ possible weights, this gives a sparsity of $1 - \frac{118}{307328} = 0.9991$. The second layer has a sparsity of 0.9992. The third layer has a sparsity of 0.9895. Sparsity has not been implemented for biases. These are all dense.

6.4 Hyperparameters

The first 6 networks have been trained with $\epsilon = 0.1$, the other 6 with $\epsilon = 0.5$. For half the networks, the SET evolution step has been disabled. This has been done to give another good indication of the potential of the SET algorithm. Each subset of data has 4 networks trained on it, 2 with SET enabled. Each network has been trained for 1500 epochs if $\epsilon = 0.1$, and 150 epochs if $\epsilon = 0.5$, all with an exponentially decaying learning rate starting at 30 and ending at 1. For the networks with SET enabled, evolution happened

id	ϵ	epochs	dataset	topology
1	0.1	1500	1	SET
2	0.1	1500	1	static
3	0.1	1500	2	SET
4	0.1	1500	2	static
5	0.1	1500	3	SET
6	0.1	1500	3	static
7	0.5	150	1	SET
8	0.5	150	1	static
9	0.5	150	2	SET
10	0.5	150	2	static
11	0.5	150	3	SET
12	0.5	150	3	static

Table 3: Enumeration of all networks trained

every 25 learning steps, with an exponentially decaying evolution rate starting at 50% and ending at 10%. For clarification, all trained networks are listed and numbered in Table 3.

6.5 Training performance

For all 12 networks, both the accuracy and loss have been plotted against the training epochs. The figures and tables can be found in the Appendix on page 66.

6.5.1 Performance with $\epsilon = 0.1$

The graphs in Figures 31, 32, and 33 clearly show that the networks that have SET disabled are barely or not able to increase in accuracy, even though their losses decrease. The 3 networks with SET enabled are all able to increase their accuracies. The loss plots have clear bumps in them, indicating where evolution occurred. The networks with SET enabled and trained on 5 classes reach 74% and 43% accuracy. The network trained on all 10 classes reaches 42% accuracy. Surprisingly, the network trained on 10 classes performs just as well as the network trained on classes 5-9. One would expect that training on double the number of classes will give worse performance. Final performances of all networks on all datasets is listed in Table 7.

6.5.2 Performance with $\epsilon = 0.5$

The graphs in Figure 34 to 39 show that all the networks are probably train on the datasets, with and without SET. The networks with SET enabled, however, do reach slightly higher accuracies. networks trained on 5 classes reach accuracies above 70%. Interestingly, the network trained on classes 5-9 performs around as well as the network trained on classes 0-4. This was not the case with $\epsilon = 0.1$. This might just be bad luck, given that SET is a random process. Final performances of all networks on all datasets are listed in Table 8.

6.6 Performance after merging

An interesting case is to merge two networks trained on different classes and see how the resulting merged network performs on all relevant classes. How does the merged network perform compared to the two original networks? To test this case, two merged networks have been created. The first merged network originates from networks 1 and 3, which have $\epsilon = 0.1$. The second merged network originates from networks 7 and 9, which have $\epsilon = 0.5$. In both cases, two networks that were trained on separate classes (datasets 1 & 2) are merged, and the resulting network is tested on all classes (dataset 3). Both the accuracies and losses of all relevant networks are compared. Results of merged networks are listed in Table 9 for $\epsilon = 0.1$ and Table 10 for $\epsilon = 0.5$. It is important to note that the merged networks have **not** been pruned after merging, thus having around twice the density of the original networks. Pruning has been omitted on purpose since its effects on a merged network is unknown. It is important to first establish if merging networks holds any merit before taking further steps.

The 4 merging methods (merging method *nothing* is not used) listed in 4.1 have been applied to both the weights and the biases of networks. A total of 16 combinations have been tested. Discussed below are the results for both values of ϵ.

6.6.1 Performance after merging with $\epsilon = 0.1$

Performance of different merging methods The performances of the merged network on dataset 3 is displayed in Table 9. The best accuracies and losses are obtained by averaging the biases. This method performs significantly better than the other bias merging methods. Averaging the biases gives around 20% better accuracies over random merging, and around 10% better accuracies over magnitude and addition merging. The weight merging method does not seem to have a large influence. The accuracies and losses of magnitude, addition, and random merging is basically the same given a bias merging method. This was foreseen in the section 'Merging method similarity' on page 21. The average weight merging method seems to perform the worst, with an exception when the biases are chosen randomly. Interestingly, the combination with the lowest loss (average - average) does not have the highest accuracy.

Performance compared to original networks The best accuracy obtained on dataset 3 by the merged network with $\epsilon = 0.1$ is approximately 30% with a loss around 2.52. This is not bad compared to the results of the original networks 1 and 3, with respectively 37% and 22% accuracy. The merged network seems to have the average performance of these two networks. It is however not as good as network 5 with an accuracy of 43%, which was trained directly on dataset 3. Ideally, it would combine the classification performance of both networks 1 and 3, resulting in an average accuracy of $(0.74 + 0.43)/2 = 59\%$, but unfortunately, this is not the case. Looking at the loss of around 2.52, it is much lower than networks 1 and 3, with a respective loss of 6.84 and 6.34. Again, network 5 performs better with a loss of 1.48.

6.6.2 Performance after merging with $\epsilon = 0.5$

Performance of different merging methods The performances of the merged network on dataset 3 is displayed in Table 10. The best accuracies and losses are obtained by averaging or adding the biases. These two methods give around 10% higher accuracies

than the two other bias merging methods. Just as with $\epsilon = 0.1$, the weight merging method does not seem to have a large influence. In this case, the combination with the lowest loss (average - average) also has the highest accuracy.

Performance compared to original networks The best accuracy obtained on dataset 3 by the merged network with $\epsilon = 0.5$ is approximately 35% with a loss around 1.92. This is worse than the two original networks 7 and 9, with a respective accuracy of 40% and 45%. It is also much worse than network 11, which was trained directly on dataset 3, with an accuracy of around 78%. Looking at the loss of 1.92, the merged network performs significantly better on dataset 3 compared to networks 7 and 9, with a respective loss of 6.15 and 6.26. The merged network has more than three times the loss of network 11 with a loss of 0.60.

6.7 Summary

The results show that merging two SNNs trained on two different datasets can lead to a merged SNN that is able to predict from the aggregate of both datasets. This means that the used merging techniques are able to capture the discriminative essence of both SNNs and place these into a single SNN. Performance of the merged SNN is however strongly dependent on the merging technique used, indicating that some techniques better capture the essence of both SNNs than others. Results also show that the technique chosen to merge biases has a more substantial impact on performance than the technique chosen to merge weights. This can be explained by the fact that three out of the four merging techniques are identical in their function if both SNNs do not have weights at the same places, which is highly likely at high sparsities. Comparing the merged SNN to the SNN trained directly on dataset 1 and dataset 2, the merged SNN performs considerably worse. This leads to the conclusion that even though the merging techniques do manage to capture some of the essences of both SNNs, they are still lacking.

7 Phase 2

7.1 Phase 1 to phase 2

Results from phase 1 have shown that it is possible to merge two SNNs in such a way that the essence responsible for their performance is preserved in the merged SNN. The question remains what this essence entrails. What makes a neural network perform well? This question sparked the development of SNNs in the first place, in 1989. By process of elimination, researchers tried to figure out what made an SNN work. Research such as the Lottery Ticket Hypothesis and SET indicates that it is a combination of topology and weights that play well together. I expect that research into the merging of SNNs will correspond closely with this question about the essence of neural networks, about what makes neural networks work. If we want to merge SNNs successfully, we must first understand this essence before manipulating it.

To get closer to the answer of this question, it might be a good idea to see how well the merging methods perform while trying to take out as many other factors as possible. This includes the three different datasets that were used in phase 1. Merging results from phase 1 showed that the merging method *magnitude* gives a relatively good performance. However, does this mean that stronger weights are important in a neural network? Or is this because of some unforeseen side-effect from merging two SNNs trained on different datasets? It might be possible that one dataset leads to much stronger weights in a network, thus influencing the outcome of the merging method *magnitude*.

Before asking ourselves if merging two SNNs into one can perform well on two datasets (as tested in phase 1), we should ask ourselves a more rudimentary question. Can two SNNs, that differ only in topology, be merged together, without losing performance? Merging two SNNs that differ only in topology is the most basic scenario. It takes all other influences out of the experiment. This thought process led to the setup of phase 2. Phase 2 attempts to merge pairs of SNNs that differ only in their topology. All other factors such as dataset and hyperparameters are always equal between two SNNs about to be merged. A total of 63.000 merged neural networks are analyzed to get a better picture of the performance of the merging methods.

Changes between phase 1 and phase 2 Some changes were made to the methodology before phase 2 was started. First, the merging method *nothing* was added as a baseline. This increases the total number of merging method combinations from 16 to 25. Second, the NNSTD metric was added. This metric will be used to better analyze the results from phase 2. Unfortunately, NNSTD could not be retroactively added to phase 1.

7.2 Methodology

To explore the capability of merged SNNs to retain the parents' performance, numerous SNNs have been trained, evaluated, and merged in pairs. Subsets of the Fashion-MNIST dataset have been used for training and evaluation. Sparsity level is the only variable hyperparameter in this experiment. 12 sparsity levels have been selected, ranging from 0.00 (all weights present) to 0.99 (missing 99% of all weights). 20 networks have been trained for each sparsity level 0.00, 0.10, 0.20. 0.30, 0.40, 0.50, 0.60, 0.70, 0.80. 0.90,

0.95, 0.99, resulting in a total of 240 trained networks. Note that sparsity is applied per network layer. If the sparsity is applied on the entire network, it can happen that one layer is stripped of all weights, thus crippling the network. For each sparsity level, these 20 networks have been paired (including with itself), resulting in 210 network pairs per sparsity level. Each pair has been merged using one of the 5x5 proposed merging methods (page 20), giving 5250 merged SNNs per sparsity level, resulting in a total of 63000 merged SNNs.

12 sparsity levels x 210 network pairs x 25 merging methods = 63000 merged SNNs

The performance (accuracy and loss) of the merged SNNs are evaluated on the evaluation subset of Fashion-MNIST. Next to these metrics, the distance between the merged SNNs and their original SNN pairs has been measured using the NNSTD metric (page 17). These results are then used to analyse the effectiveness of each proposed merging method.

7.3 Data

Five classes were selected from the Fashion-MNIST dataset : T-shirt/top, Trouser, Pullover, Dress, and Coat. For each class, 1000 train samples and 100 test samples were taken. In contrast to phase 1, phase 2 trains all networks on the same dataset.

7.4 Network architecture

Trained networks are exactly as described in section 'Network architecture' on page 25, with 5 output neurons corresponding to the 5 selected Fashion-MNIST classes.

7.5 Hyperparameters

All networks have been trained on the same dataset and with the same hyperparameters (except for sparsity level).

Training epochs Each network has been trained for at most 1500 epochs. Suppose at any point during training, a network passes the threshold of 85% accuracy on the test set. In that case, the training will terminate after 200 extra epochs or if it reaches 1500 epochs, whichever arrives first. This has reduced training time by around 60%.

Learning rate and evolution rate The learning rate is 1 at epoch 0, exponentially decaying to 0 at epoch 1500. The evolution rate, which indicates the fraction of the weights to replace during network evolution, is 0.5 at epoch 0, and exponentially decays to 0 at epoch 1500. Both rates can be seen in Figure 8.

Evolution Evolution of the network happens every 50 epochs, according to the SET algorithm by Decebal et al [26]. At evolution, a certain amount of weights are deactivated, starting with the weights that have the lowest magnitude. After this step, the exact same amount of inactive weights are randomly chosen and activated. Note that a weight that is being activated might very well be a weight that was just deactivated. Evolution happens on a per-layer basis, meaning that weights can not move from one

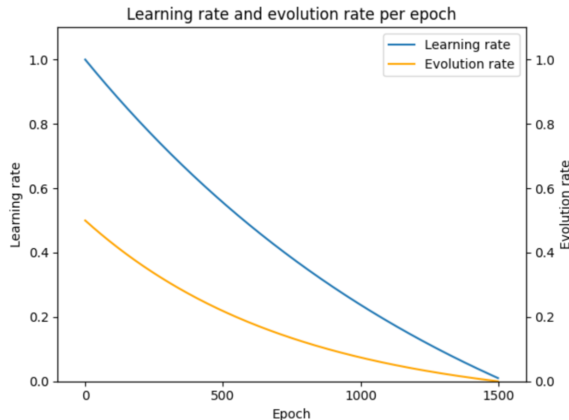


Figure 8: Learning rate and evolution rate per epoch.

layer to another. A layer will always have the exact same amount of active weights before and after evolution. Given a sparsity level s, evolution rate e, and the number of weights in a layer n, the exact number of weights affected during evolution equals $n(1-s)e$. This means that denser networks will have more weights affected during evolution. Weights are not reset on deactivation but keep their current value. This leads to evolution having no effect on completely dense networks, as will be further explained in 7.6.3. Note that this is not possible in a truly sparse setting, since deactivated weights will have to be removed from memory. Biases are fully dense, and not subjected to evolution.

7.6 Training Results

7.6.1 Early stopping

As can be seen in Figure 11(c), all sparsity levels except for 0.99 reach an average accuracy of 85% before 250 epochs, and settle around 87% accuracy. Networks might have reached a slightly better accuracy if trained for the full 1500 epoch. However, since reaching the best performance on trained networks is not part of this experiment, slightly better accuracy is irrelevant for this research.

7.6.2 Accuracy and loss over time

Figure 12 shows the final accuracy for all trained networks. Figure 10 contains four plots showing the accuracy (left) and loss (right) of all trained networks over all epochs. Figure 11 contains six plots which show the average accuracy and learning rate, as well as a moving average of the standard deviation, with a window size of 10. The standard deviation has been plotted around a moving average of the loss and accuracy, also with a window size of 10. This has been done to make the standard deviation plot smoother and give a clearer picture.

Epoch cutoff on Figure 11. The plots in Figure 11 are plotted up to the lowest final training epoch per sparsity level. For example, if 1 network at sparsity level 0.70 required 250 epochs to finish training, and all other 19 networks at sparsity level 0.70

required 350 epochs, then the average will still be plotted to only 250 epochs, thereby discarding the last 100 epochs of all the other networks. This has been done not to give a wrong depiction of the average accuracy and loss in the later epochs. As can be seen in Figure 10(c), networks tend to become unstable and fluctuate strongly in their loss and accuracy. When the number of networks that are being averaged start dropping, this instability starts showing itself in the average. This average then suggests that all networks experience the same peaks and drops simultaneously, which is not the case.

Final accuracy Figure 11(c) shows that all but the sparsest networks reach the 85% accuracy threshold somewhere between 50 and 250 epochs on average. This is quick compared to the 1500 epochs training limit. Only the networks with sparsity level 0.99 take quite some time to reach this accuracy, somewhere between 600 and 1500 epochs. Figure 12 shows that even though networks with a higher sparsity level take longer to reach a certain accuracy, all networks stabilize around 87%, with just 3 outliers out of 240 total networks. This indicates that 87% accuracy is the limitation of the network on this dataset, given all the hyperparameters. With even networks with sparsity level 0.99 reaching this accuracy limit, it's certainly possible that there are even higher sparsity levels that would be able to reach the accuracy limit.

Accuracy vs loss While basically all networks can reach an accuracy of 87%, they do not all reach the same amount of loss. Figure 11(b) shows that on average, denser networks are able reach a lower loss faster before terminating training. Comparing the accuracy of Figure 11(c) against the loss of Figure 11(d) shows that after around epoch 100, the accuracy for most sparsity levels start plateauing, even though the loss is still decreasing. This implies overfitting of the networks, and again suggests that sparser networks should be able to train on this dataset as well. Figures 11(a) and 11(b) show that only for the lowest sparsity level .99, the accuracy and loss start plateauing around the same time, suggesting that this level of sparsity might be close the the maximum sparsity the network can handle before not being able to reach the suggested limit of 87% accuracy.

7.6.3 Influence of evolution on loss and accuracy

As stated in above in the Hyperparameters section 7.5, evolution of the network occurs every 50 epochs. Since trained, activated weights possibly get swapped out for untrained, deactivated weights, one would expect a jump in loss and a drop in accuracy around every 50 epochs. Figure 11(e) and 11(f) show that this is indeed the case, especially in early epochs where the evolution rate is relatively high. Figure 11(f) shows a spike in loss when evolution happens. This spike is stronger for more sparse networks. The reason for this may be that the probability that an activated weight gets replaced with a deactivated weight is higher for sparser networks, as will be explained later on. The figures do not show a drop in accuracy, but merely the plateauing for a handful of epochs for the networks with higher sparsity levels. A possible explanation might be that, combined with the spike in the loss, the weights deactivated during evolution had some influence on the class prediction but were not decisive. Deactivating these weights was not enough to flip class predictions but enough to change the loss landscape in such a way that the network has to train some more to get back at where it was before evolution. Once it again reaches that point, it can continue increasing its accuracy.

One exception to the general trend of spikes in loss and plateauing accuracies can be seen at sparsity level 0.99. There is a very slight bump in the loss, nowhere near as extreme as at sparsity level 0.95. A plateauing in accuracy cannot be determined since the accuracy was already plateaued before evolution occurred. This might explain the missing spike in loss since no improvement in both accuracy and loss hints at a relatively flat loss landscape, thus minimizing the effect of moving around on that landscape because of evolution.

Weight deactivation probability The probability that an activated weight is selected for evolution equals the evolution rate e. The fraction of all deactivated weights equals the sparsity level s. The fraction of all weights, including deactivated weights, that are selected for evolution equals $f = (1-s)e$. The probability that an active weight, selected for evolution, is not again selected for activation equals $1 - \frac{f}{s+f}$. The probability that an active weight gets selected for evolution and is consequently deactivated equals

$$e \cdot (1 - \frac{f}{s+f}) = e \cdot (1 - \frac{(1-s)e}{s+(1-s)e}$$

Figure 9 shows a heatmap of weight deactivation probabilities for different evolution rates and sparsity levels. It shows that at higher evolution rates, weights in a sparser network have a higher probability of being deactivated. Note that for the completely dense network, the deactivation probability equals 0 regardless of the evolution rate. This makes sense, since there are no deactivated weight available to possibly replace the currently activated weights. Since networks with a higher sparsity level have a higher probability of weight deactivation, it follows that evolution will have a stronger impact on both their accuracy and loss.

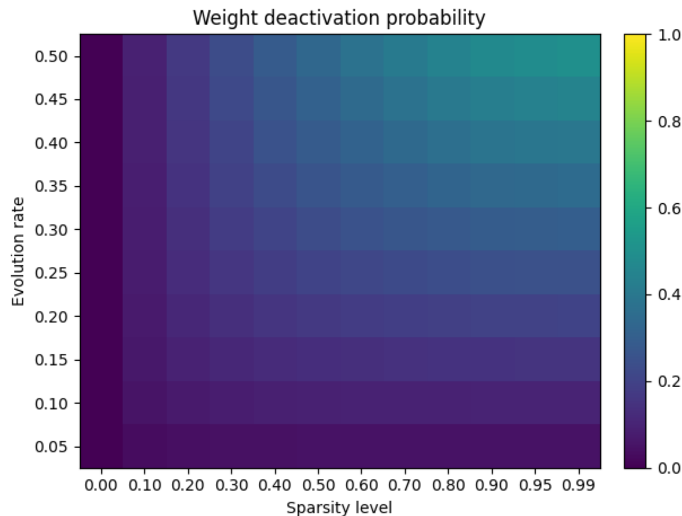
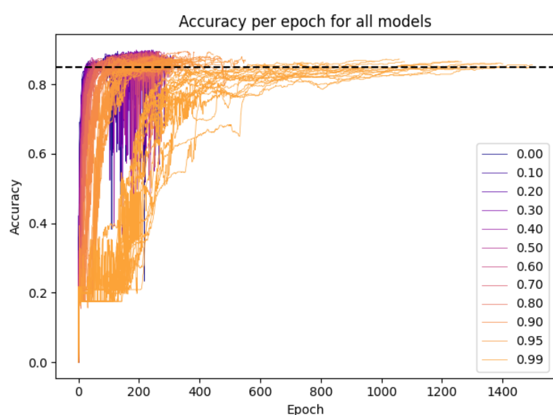


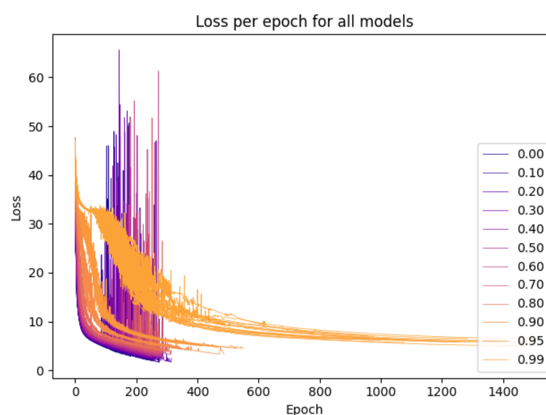
Figure 9: Weight deactivation probability for different evolution rates and sparsity levels.

network instability As can be seen in Figures 10(c) and 10(d), the more dense networks start becoming more unstable after reaching the accuracy threshold of 87%. Both their accuracy and loss start spiking heavily. The spikes in accuracy might be attributed to overfitting, but this does not confirm with the spikes in loss, which should still be going down. The spikes in loss are extreme, sometimes even higher than the original loss before

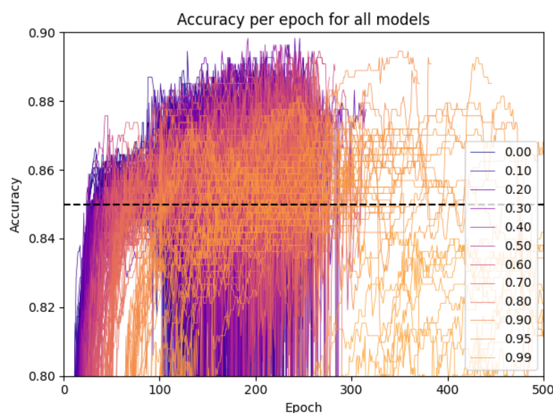
training. This can be clearly seen in Figure 10(b). These spikes in accuracies and losses are also short-lived. Most spikes are gone by the next training epoch. Figures 10(c) and 11(d) clearly show the the higher standard deviation on both the loss and accuracy for higher density networks. The reason for these spikes is still unknown. Evolution can not be the cause, since that only happens every 50 epochs, and the spikes are spread out over all epochs. Even though the denser networks are more unstable, Figure 12 shows that all but three networks reach around 87% accuracy. The most likely explanation is that the learning rate was still too high at those epochs.



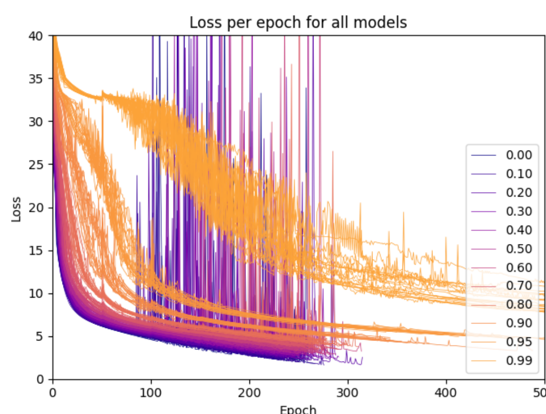
(a) Accuracy over all networks and epochs



(b) Loss over all networks and epochs

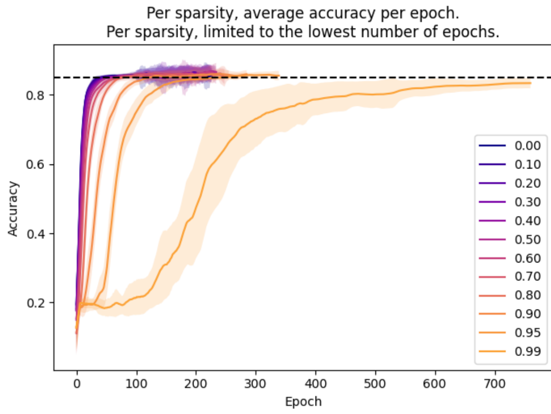


(c) Accuracy over all networks for epochs 0 to 500

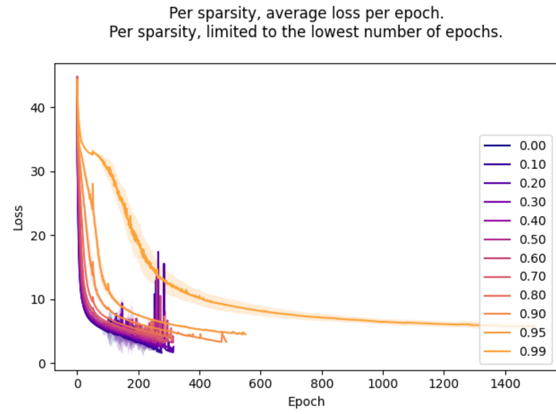


(d) Loss over all networks for epochs 0 to 500

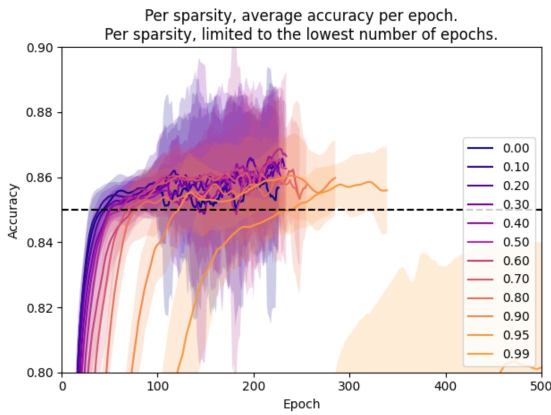
Figure 10: Per sparsity, accuracy (left column) and loss (right column) per epoch.



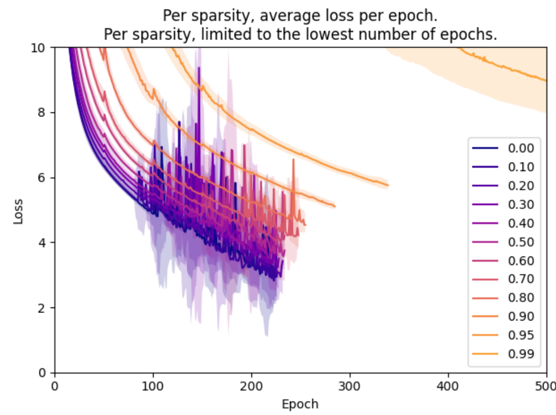
(a) Average accuracy over all epochs



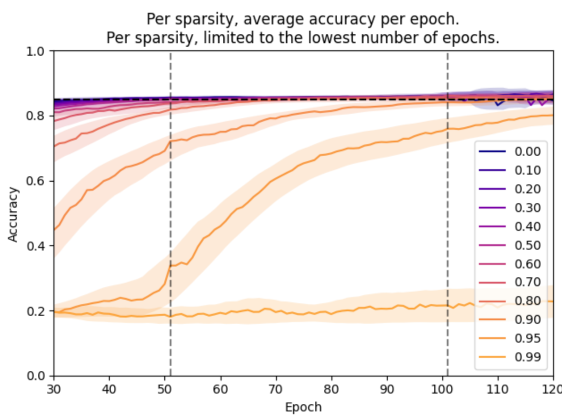
(b) Average loss over all epochs



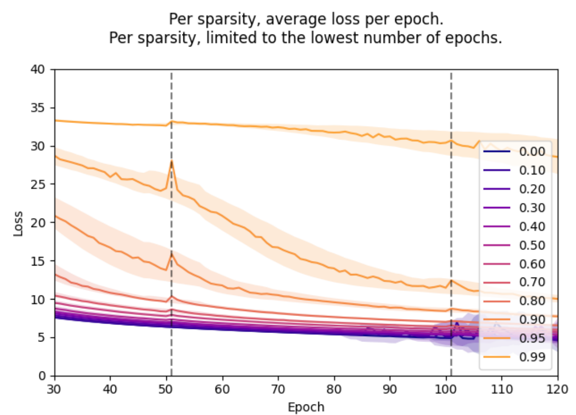
(c) Average accuracy over epochs 0 to 500



(d) Average loss over epochs 0 to 500



(e) Average accuracy over epochs 30 to 120



(f) Average loss over epochs 30 to 120

Figure 11: Per sparsity, average accuracy (left column) and loss (right column) per epoch. The shaded areas indicate a moving average, size 10, of 1 standard deviation. Per sparsity, the number of displayed epochs is limited to the network with the lowest number of training epochs.

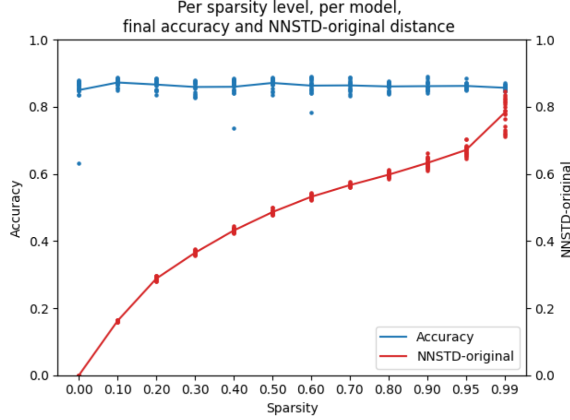


Figure 12: Final accuracy and NNSTD-Original distance for every network.

7.7 Merging Results

To keep the following sections readable, the following nomenclature is used.

magnitude_average : References all merged SNNs where the *magnitude* merging method was applied to the weights, and the *average* merging method was applied to the biases.

*addition_** : References all merged SNNs where the *addition* merging method was applied to the weights.

**_nothing* : References all merged SNNs where the *nothing* merging method was applied to the biases.

_ : References all merged SNNs.

7.7.1 Best overall performance

Figure 13 shows two heat maps of both the accuracy and loss for all 25 merging methods. For each merging method, the average accuracy and loss is calculated over 12 sparsity levels x 210 merged networks = 2520 values.

Best layer merging method Looking at the accuracy heat map in Figure 13, on average, the best layer merging method is *magnitude*, closely followed by *addition*. All *magnitude_** SNNs performs equal or just 1% better compared to *addition_**. The third best is *random_**, which shows an accuracy around 10% lower over all bias merging methods, compared to *magnitude_** and *addition_**. After this follows *average_**, which shows an accuracy around between 5% and 10% lower over all bias merging methods compared to *random_**. The same order of performance goes for loss as well; *magnitude_**, *addition_**, *random_**, *average_**. Interestingly, although the accuracy between *magnitude_** and *addition_** is the same, the loss for *magnitude_** is in all cases around 10% lower compared to *addition_**.

Best bias merging method The accuracy difference between the bias merging methods **_magnitude*, **_average*, and **_addition*, is almost negligible, except at *average_**. There is also almost no difference in accuracy between **_random* and **_nothing*. Figure

Weights		Biases	
Accuracy	Loss	Accuracy	Loss
Magnitude	Magnitude	Average	Average
Addition	Addition	Magnitude	Magnitude
Random	Random	Addition	Addition
Average	Average	Random	Nothing
Nothing	Nothing	Nothing	Random

Table 4: Ranking of all merging methods according to Figure 13. The methods are sorted in descending order, with the best performing method at the top.

13 shows that over all layer merging methods, **_average* gives the best accuracy, followed by **_magnitude* which only underperforms at *average_** by 5%. The third best bias merging method is **_addition* which, again, underperforms at *average_**, by 6%. The fourth bias merging method is **_random*, which performs around 3% worse than **_addition*. At the bottom is the bias merging method **_nothing*, which performs around 1% worse than **_random*.

Looking at the loss in Figure 13, the trend is almost the same. The only difference is that **_nothing* now slightly outperforms **_random*. Looking at **_addition* and **_average*, their accuracies are almost equal but the loss from **_addition* is higher. Ignoring the layer merging method *nothing_**, the heat map shows no other interesting observations on the bias merging method regarding the loss.

Best merging method The previous two paragraphs, together with Figure 13, determined the merging method ranking in Table 4. The best methods are at the top and the worst methods are at the bottom. The overall best merging method, looking at accuracy, is either *magnitude_average* or *magnitude_magnitude*. The best overall merging method, looking at loss, is *magnitude_average*. Thus, it can be concluded that overall, the best merging method is *magnitude_average*.

Layer : nothing The results show very poor performance for any *nothing_** SNN. For all of these, the accuracy is 0.20. With only 5 classes that are equally represented, this means that the merged networks are dead set on always choosing a single specific class, regardless of the input given. These merged networks are completely useless. *nothing_** SNNs always giving the same output is simply explained by the fact that these networks have no weights, meaning that the input is completely disregarded. The output is then dependent on only the biases in the last layer of a network. Since biases are constant, the output will also be constant.

The corresponding loss is also very high compared to other merging methods. Interestingly, even though the accuracy is the same across all five bias merging methods, the losses are not. Using *nothing_addition*, the loss is highest with 5.40. *Nothing_nothing* gives the lowest loss at 2.30. The worst layer merging method is *nothing_**, and given that, the best bias merging method is then **_nothing*. Figure 14(b) gives a more nuanced illustration about the loss for *nothing_**. It shows that networks with a sparsity between .60 and .80 have the highest loss. Networks with the lowest sparsity levels also have the lowest losses.

Bias : nothing **_nothing* SNNs are not as heavily affected as *nothing_** (except for the merging method *nothing_nothing*). Whereas using *nothing_** completely renders networks useless, using **_nothing* merely slightly affects network performance. For each layer merging method in the accuracy heat plot of Figure 13, the **_nothing* SNNs have the lowest accuracy out of the five bias merging methods. The loss does not show the same trend. The same networks do not necessarily have the highest loss out of the five bias merging methods. It shows that the bias is not critical in these networks, but only gives a relatively small accuracy increase. Of course, this does not imply that bias will always play a small role in every merged network. Figure 14(b) does not give interesting observations on this matter.

7.7.2 Impact of layer merging method and bias merging method

Figure 14(a) gives an indication of the impact that the layer-, and bias merging methods have, on average. The maximum impact and average spread of all methods on accuracy have been listed in table 5. The table shows that switching bias merging methods will have more impact on the accuracy than switching layer merging methods. Except for *average*, all methods have larger maximum differences and spread when switching the bias merging method, than when switching the layer merging method. Figure 14(a) shows that the bias has more impact at higher sparsity levels. This makes sense since the bias has more representation. Still, the non-bias weights have the majority, and it would be reasonable to suggest that these should make the larger impact.

At low sparsity levels, the bias only makes up a tiny part of all the weights, and the layer merging methods do not act the same, as they do at higher sparsity levels. Therefore, at lower sparsity levels, switching the layer merging method should have the most impact. At high sparsity levels, the bias makes up a larger part of all the weights and the layer merging methods start acting the same (4.2). Therefore, at higher sparsity levels, switching the bias merging method should have the most impact. This is supported by Figure 15 supports this. At low sparsity levels (0.00, 0.10), changing the bias merging method barely makes an impact. Changing the layer merging method from *magnitude_** to *random_** results in a 20% performance drop. At high sparsity levels (0.95, 0.99), changing the layer merging method barely makes an impact. Changing the bias merging method from **_addition* to **_random* results in a 15% performance drop.

In summary, at lower sparsity levels, the layer merging method has the most impact. At higher sparsity levels, the bias merging method has the most impact. It is unclear where exactly the border lies between these two.

7.7.3 Support for SET

Table 4 shows that, looking at accuracy, *average_** performs worse than *random_**, and that *magnitude_** performs better than *random_**. *average_** will sometimes result in stronger weights than *random_**, sometimes weaker. *magnitude_** will always result in weights stronger or just as strong as *random_**. It follows that stronger weights leads to better performance. This conclusion is also what powers the SET algorithm. The results found in the SET-paper [26] and these experiments seem to agree with each others.

layer merging method	addition_*	average_*	magnitude_*	random_*	
maximum difference	0.05	0.13	0.05	0.07	
spread (1 std)	0.02	0.05	0.02	0.03	

bias merging method	*_addition	*_average	*_magnitude	*_random	*_nothing
maximum difference	0.18	0.09	0.14	0.17	0.17
spread (1 std)	0.07	0.04	0.06	0.07	0.07

Table 5: Impact of different merging methods on the accuracy, according to Figure 13. *Maximum difference* gives the difference between the highest and lowest accuracy for that method. *Spread* gives the spread (1 std) over all values for that method. *Nothing_** has not been included since it is meaningless (see 7.7.1 layer:nothing) and skews the results for the bias merging methods. This table shows that switch the bias merging method will have a larger effect on performance than switching the layer merging method.

7.7.4 Merging methods at sparsity levels

Support for SET As stated in section 4.2, at higher sparsity levels, the merging methods *magnitude*, *addition*, and *random* start giving the same merging results. This is not immediately reflected by the accuracy heat map in Figure 13, which shows that on average, *random_** underperforms to *magnitude_** and *addition_**. However, looking at the more nuanced parallel coordinate plot of the accuracy in Figure 14(a), this statement is indeed correct. At higher sparsity levels, the accuracy for these three layer merging methods are nearly identical. Only at lower sparsity levels does the performance of *random_** drop drastically, down to around 55% for the fully dense networks. The difference between accuracy at the highest sparsity level and lowest sparsity level for *random_** is over 25%. For *magnitude_** and *addition_** this is 15%, at most. The specific data for this observation has been extracted from Figure 14(a) and highlighted in Figure 15. It clearly shows the performance drop at the *random* layer merging method at lower sparsity levels.

Why does *random_** work worse at denser merged SNNs? The lottery hypothesis states that a fully dense network can be seen as containing a large number of sparse networks. Whereas *addition_** and *average_** modify the weights, *magnitude_** and *random_** only pick weights. At higher sparsity levels, random acts more like *magnitude*, as explained in 4.2. At lower sparsity levels, *random_** has to more often choose from multiple weights instead of picking the single available weight. This could be seen as selecting one of the sparse subnetworks that are present in a more dense network. The data would then suggest that *magnitude_** is better than *random_** at selecting these sparse subnetworks, and that therefore, selecting weights based on magnitude is the best method. This once again supports the results found in the SET paper.

Impact of bias between sparsity levels Figure 14(a) shows that some methods work better at higher sparsity levels, while others work better at lower sparsity levels. This can be clearly seen when comparing *average_addition* and *average_average*. *average_addition* works better for low sparsity levels, while *average_average* works better for high sparsity levels. On average these two methods differ by 11%, as seen in Figure 13. Figure 14(a) however shows that the difference between these two methods is over 15% at the highest

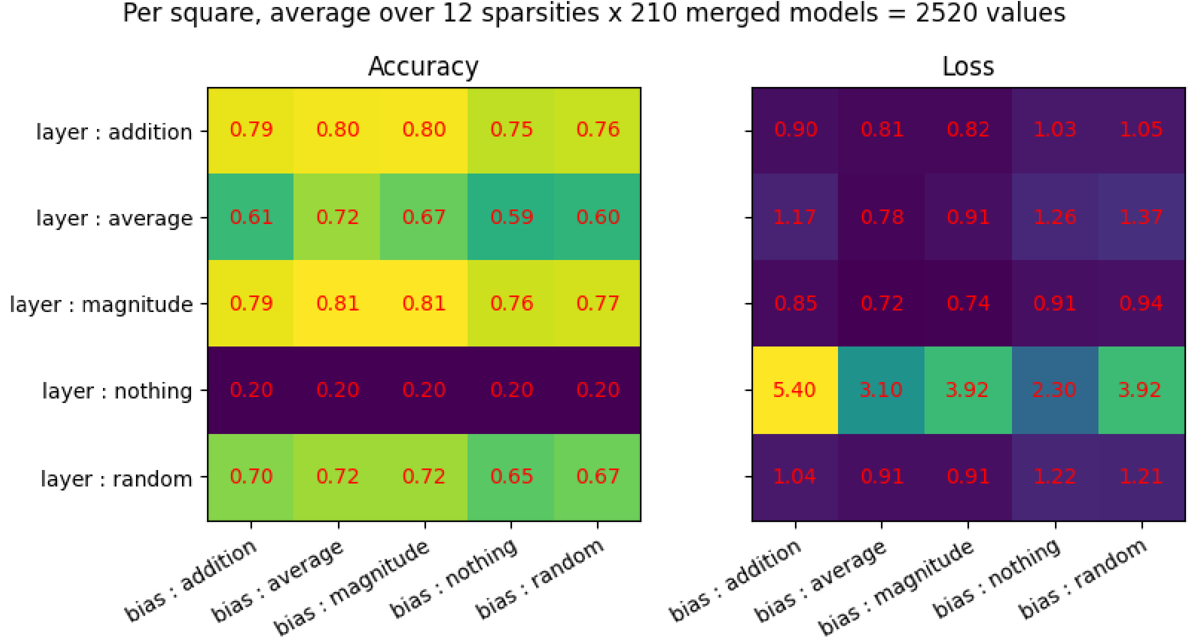


Figure 13: Each square represents a single merging method. For each merging method, the average accuracy (left) or loss (right) over 12 x 210 = 2520 networks is given. "layer" indicates the layer merging method used, and "bias" indicates the bias merging method used.

sparsity level, and around just 1% at the lowest sparsity level. It makes sense that the bias merging method has more effect on the performance of sparser networks, since the bias will have more representation. It is not yet clear why different bias merging methods give different performance results depending on the sparsity.

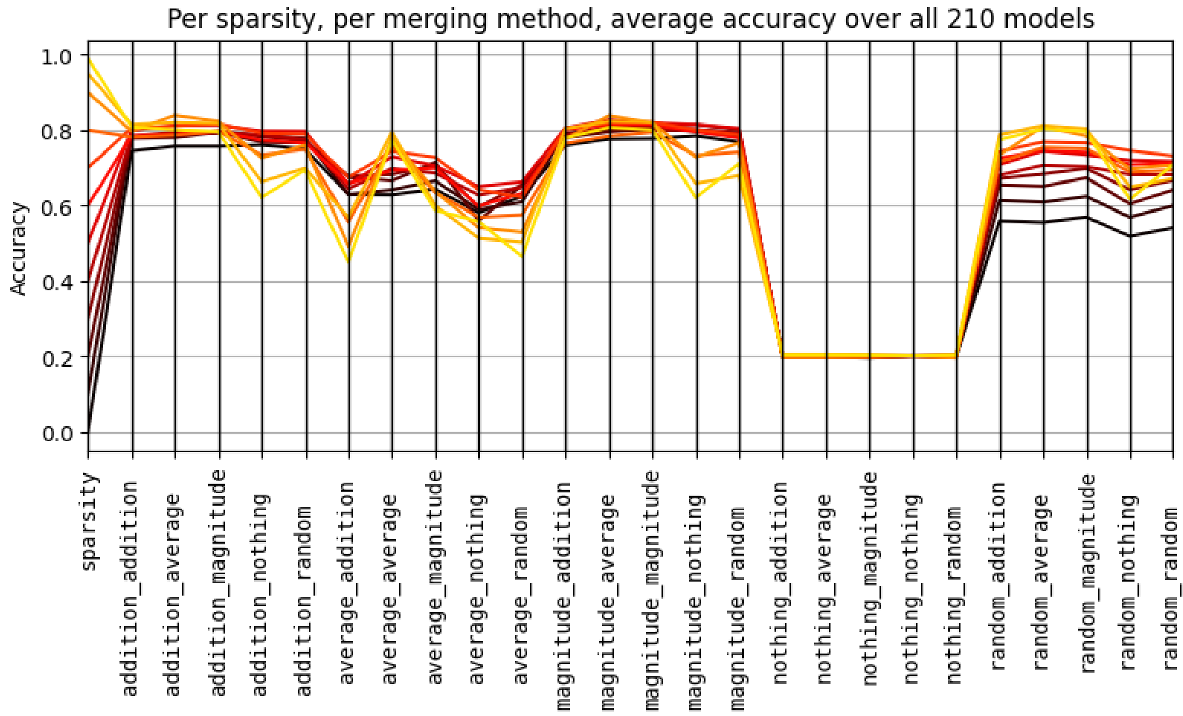
Why does *average_magnitude* perform worse at higher sparsity levels? Since the bias has more impact at higher sparsity levels, and *average_magnitude* selects the strongest biases, this might indicate that selecting the strongest biases is not a good idea. However, the opposite can be seen at *addition_magnitude* and *magnitude_magnitude*, so maybe it is a good idea.

7.7.5 Inconsistent result at *average_addition*

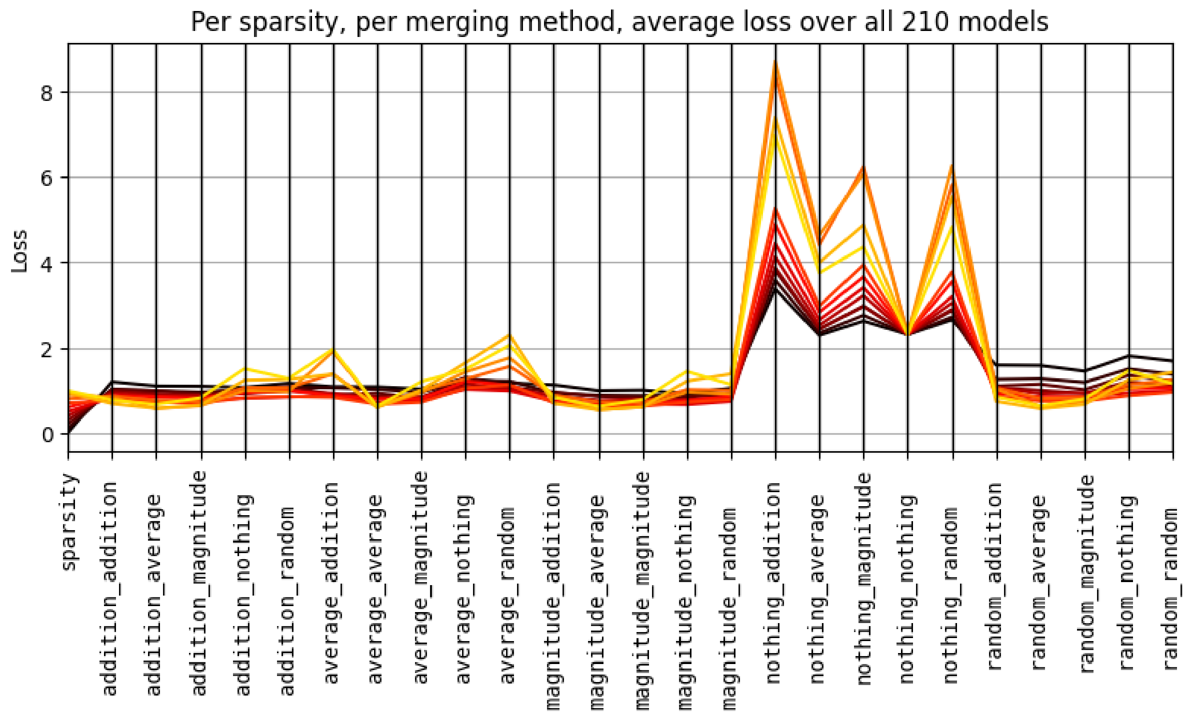
Average_addition shows an interesting result. At *addition_addition*, *magnitude_addition*, and *random_addition* the performance increases as sparsity levels rise. *Average_addition* shows the exact opposite. As sparsity levels rise, performance drops. The only difference between *average_** and the other three methods is that at high sparsities, *average_** weakens the strongest weights, while the other three methods all act like *magnitude_** and pick the strongest weights. The conclusion is then that the bias merging method **_addition* only works well when the strongest weights are selected. Why this happens is not clear to me.

7.8 Performance and NNSTD-original

The relation between merged network performance and its NNSTD-original to its parents can be measured. Merged networks come with two NNSTD-original values, indicating



(a)



(b)

Figure 14: For all sparsity levels and merging methods, average accuracy (top) and loss (bottom) over 210 merged networks

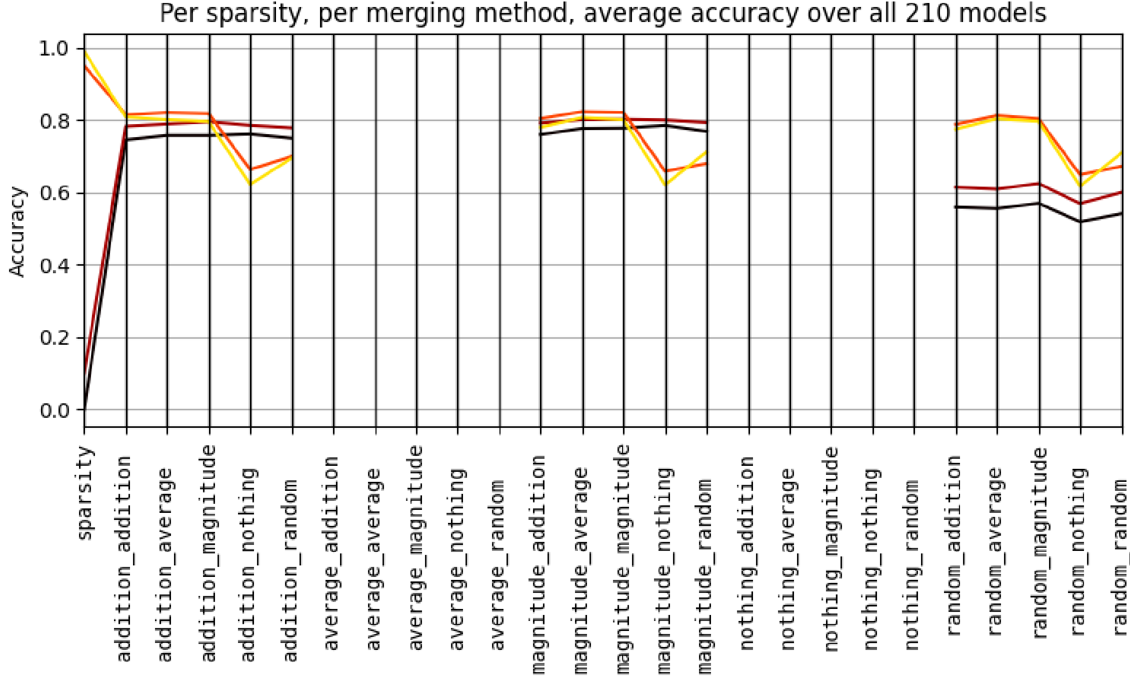


Figure 15

Figure 16: Sparsity levels 0.00, 0.10, 0.95, and 0.99. Layer merging methods *magnitude*, *addition*, and *random*. Average accuracy over 210 merged networks. This parallel coordinate plot shows how at higher sparsity levels these three layer merging methods perform the same, and how at lower sparsity levels *random* drops in performance significantly. Section 4.2 gives a possible explanation for this phenomenon.

the structure similarity to both of its parents. If high performance is correlated with low NNSTD-original to one or both parents, it might give an estimate of how merging might impact performance. If a merged network has a small NNSTD-original to one parent and a large NNSTD-original to another parent, it can show how merging methods favor one network over another.

Excluded results Two sets of merged networks have been taken out of the results in this section. The first set consists of the dense networks. These networks have an NNSTD-orginal of 0 to both parents, and therefore, looking at this is pointless. The second set consists of all networks applying the layer merging method *nothing*. These networks have an NNSTD-original of 1 to both parents. Looking at these numbers is also useless.

A note before the Figures 19(a) and 19(b) are evaluated. It's important to notice that the circles tend to overshoot, and that the top of a circle does not indicate the highest accuracy.

7.8.1 Accuracy and NNSTD-original

Relationship between NNSTD-original variance and sparsity levels As the sparsity levels increase, the spread of NNSTD-original increases as well. Looking at

Figure 19(a), the horizontal diameters of ellipses increase as sparsity level rises. The horizontal diameters change most drastically between the sparsity levels .95 and .99. This can also be seen in Table 6, which gives the average horizontal diameter per sparsity level. The average horizontal diameter slowly increases, even stabilizes halfway, and then sharply increases at the end.

Inconsistency between actual NNSTD and expected NNSTD The reason that the NNSTD-original values go up when sparsity levels go up is simply that the probability that two highly sparse networks have overlaying weights is lower. For two networks below a sparsity level of 0.50, it isn't even possible to not have overlapping weights. NNSTD increasing when sparsity increases has also been shown with the formula in section 'Expected NNSTD values for two arbitrary networks with the same sparsity' on page 19. There is however one inconsistency. While both the formula and the results show an increase in NNSTD, they don't show the same increase. For example, at a sparsity level 0.95, the formula expects an NNSTD of approximately 0.97. The results in Figure 20 show an NNSTD of approximately 0.67. Similarly, Figure 19(a) show lower NNSTD values for all sparsity levels than what would be expected, looking at Table 2.

The data implies that in basically all cases, the weights of the two parent networks are not completely random and thus not independent. This would explain the difference between the results and the formula since the formula assumes that two networks are independent. The two networks are not independent because they train on the same dataset. An explanation can be found in their *Sparse Connectivity Pattern*. Mocanu et al. have shown [27] that when an SNN is trained using SET, the first layer connects the most weights to the most important features. SET implicitly develops a feature selector when training SNNs. Figure 17 by Mocanu et al. shows an example of this phenomenon. The two figures on the left show the Sparse Connectivity Pattern at SNN initialization. The brightness indicates the number of connected weights to that specific pixel. The middle two pictures show this pattern after 150 epochs, and the two rightmost pictures show the pattern after 5000 epochs. At the rightmost pictures, the pixels at the border barely have any weights connected to them. All weights seem to be focused in the middle. This makes sense, since this is where the numbers are drawn. The pixels at the border barely contain any information.

This Sparse Connectivity Pattern can explain the dependence between two SNNs trained on the same dataset using SET. Both networks will focus on the same important pixels in the middle, while both also ignoring the pixels at the border. This in turn leads to more overlap when merging, leading to lower NNSTD results. This assumption can be confirmed by plotting the Sparse Connectivity Pattern for all trained networks, but this can unfortunately not be done due to time constraints. Instead, the Sparse Connectivity Pattern has been plotted for three networks, at three different moment, trained with different sparsity levels. The results can be see in Figure 18. The figure confirms that the networks prefer to connect to the pixels in the centre of the images. The pixels on the left and right side of the images are largely ignored. Looking at a few of the Fashion-MNIST samples in Figure 6, it can indeed be seen that there is often no information on these sides.

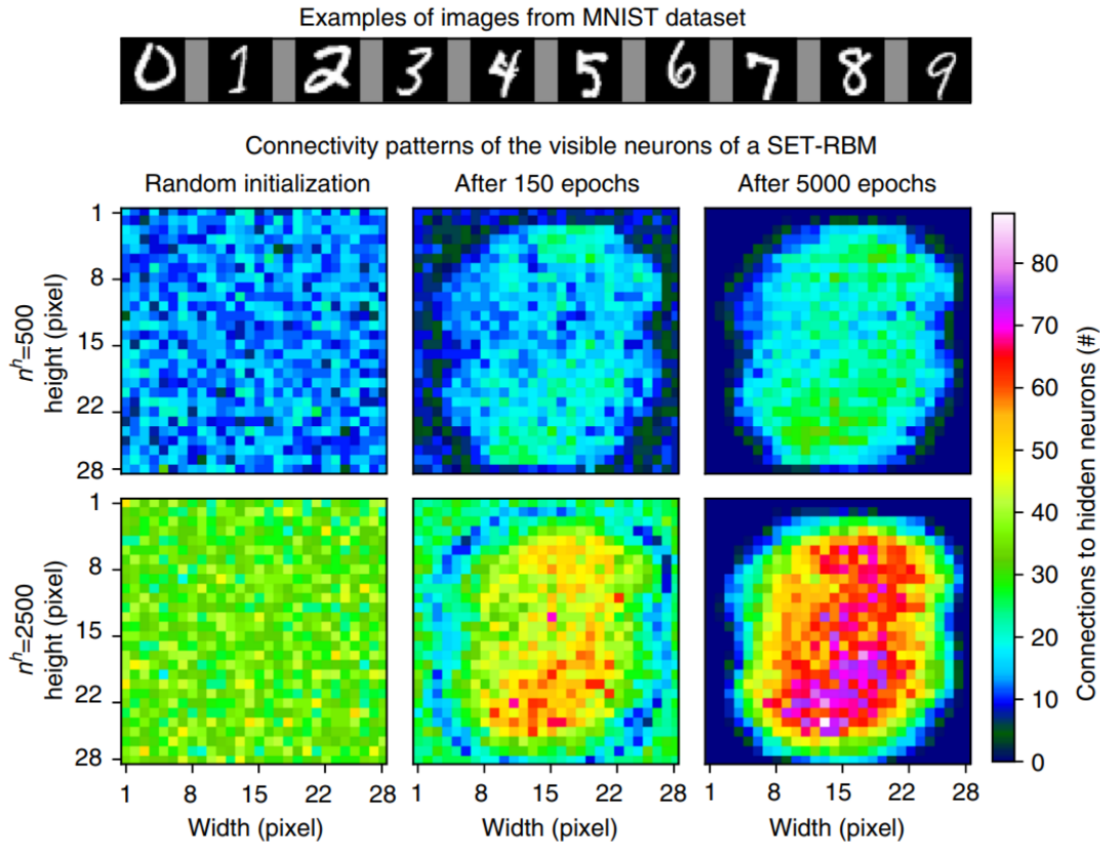


Figure 17: Sparse Connectivity Pattern for the first layer of an SNN, trained on the MNIST Dataset. Mocanu et al. have shown [27] that SNNs act as feature selectors, by connecting weights to the most important features.

Sparsity	.10	.20	.30	.40	.50	.60	.70	.80	.90	.95	.99
Diameter	.006	.011	.017	.018	.018	.015	.017	.027	.035	.057	.132

Table 6: For each sparsity, average diameter over all ellipses drawn in Figure 19(a). At sparsity level .99, strong jump can be seen from .057 to .132

7.8.2 Grouping ellipses by layer merging method

Figure 19(b) shows the same data as in Figure 19(a). However, it is now grouped by both sparsity level and layer merging method. The four different layer merging methods have all received different colors. This means that the corresponding sparsity level in Figure 19(b) can now not be seen anymore, but should be induced from Figure 19(a). More importantly than sparsity levels, the layer merging methods show a clear pattern over all sparsity levels.

Difference in accuracy consistency Looking at Figure 19(b), a quick observation can be made for each layer merging method regarding accuracy between sparsity levels.

*Magnitude_** shows consistent accuracy up to sparsity level 0.70. The green circles range from around 0.7 to around 0.85. After that, the accuracies start significantly spreading out.

*Addition_** shows consistent accuracy up to sparsity level 0.70. The red circles range

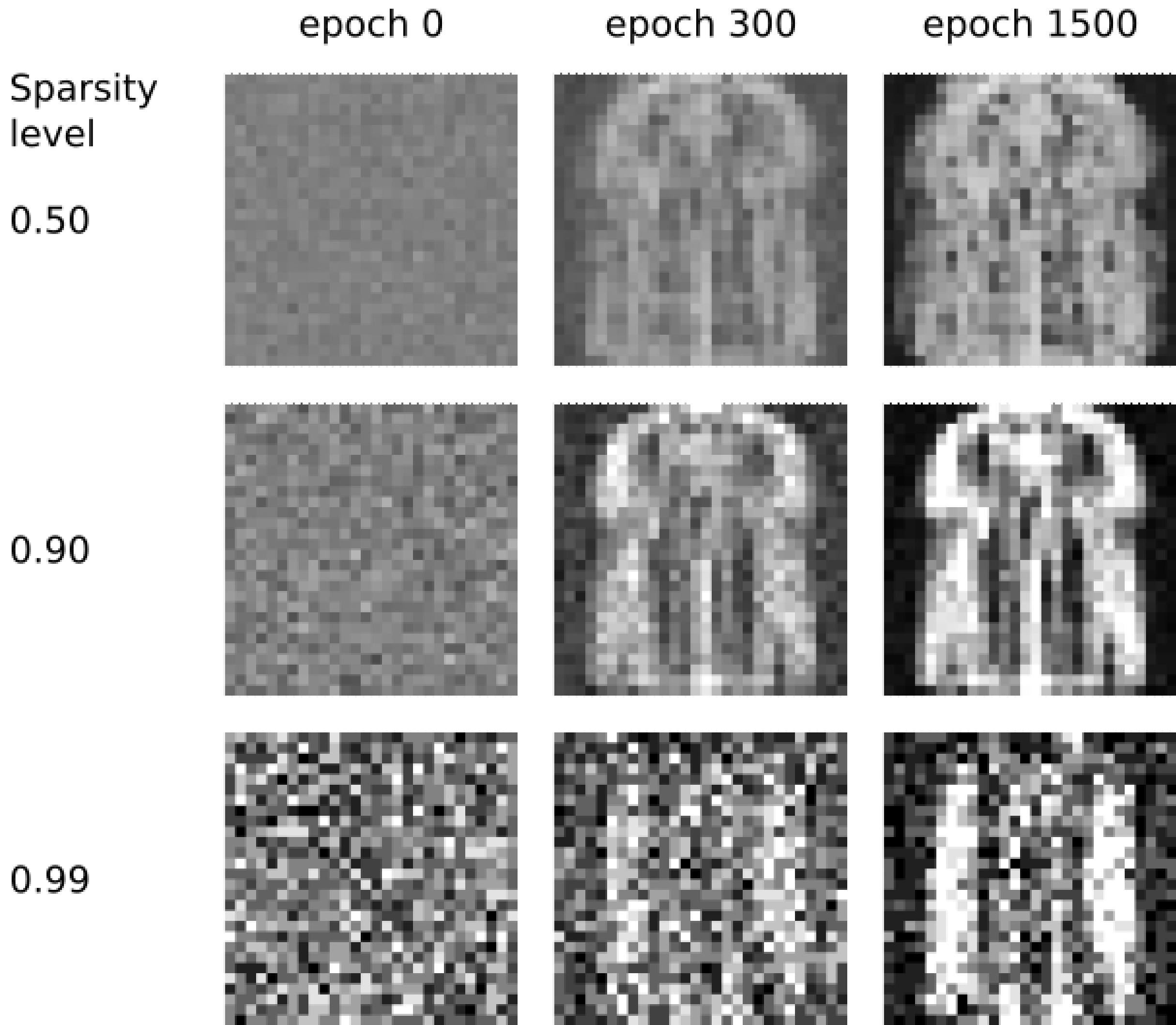


Figure 18: Sparse Connectivity Pattern for the first layer of three networks, trained on the Fashion-MNIST Dataset. The networks have been trained on sparsity levels 0.50, 0.90, 0.99. Snapshots have been made at epoch 0 (initialization), epoch 300, and epoch 1500 (end of training).

from around 0.7 to around 0.85. After that, the accuracies start significantly spreading out.

*Average_** shows almost consistent, slightly upwards accuracy up to sparsity level 0.70. The performance of Average_* increases slightly as the sparsity level rises, which can be seen when looking at the blue circles. These blue circles move slightly upwards.

*Random_** shows an upward accuracy when the sparsity level increases. This continues up to around a sparsity level of 0.70. After that, the accuracies start significantly spreading out.

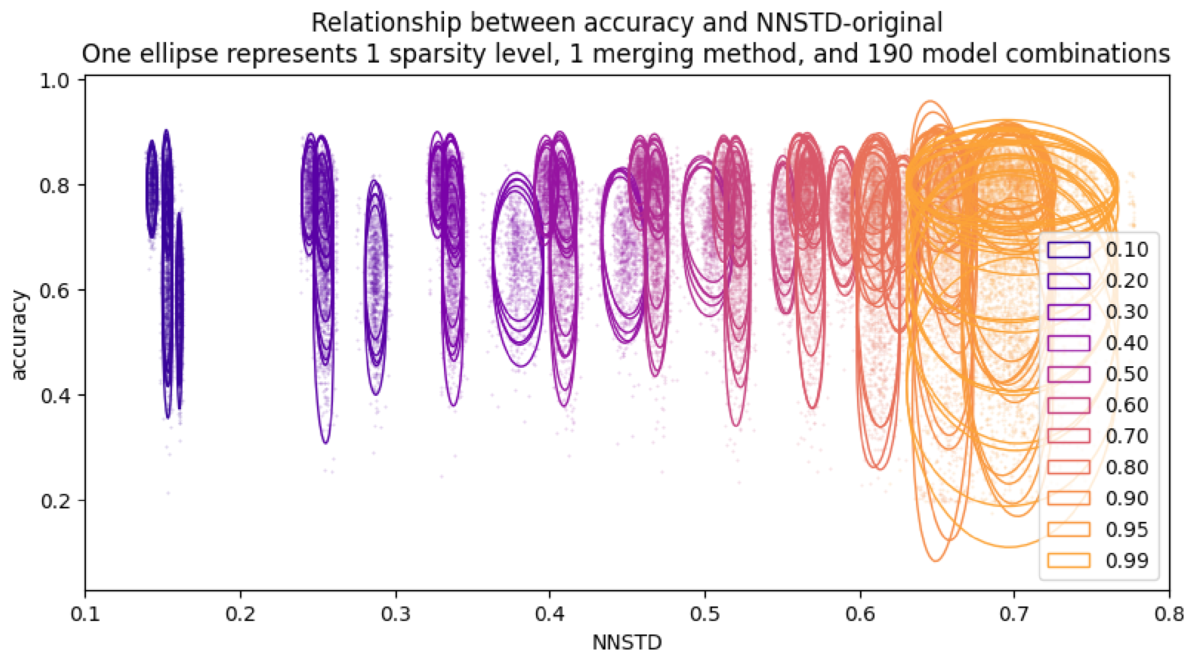
NNSTD-original At every sparsity level, the layer merging method *magnitude* can be seen in the top-left, with *addition* and *average* at a slightly higher NNSTD-original value. Right next to that, at again a slightly higher NNSTD-original level, *random* can be seen.

Accuracy *Magnitude* and *addition* have the highest accuracies and lowest spread in accuracy. *Average* has the largest spread in accuracy, and data points both at the top and bottom of accuracy values. *Random* has a slightly less accuracy spread compared to *average*, and its accuracy values are more centred around the mean.

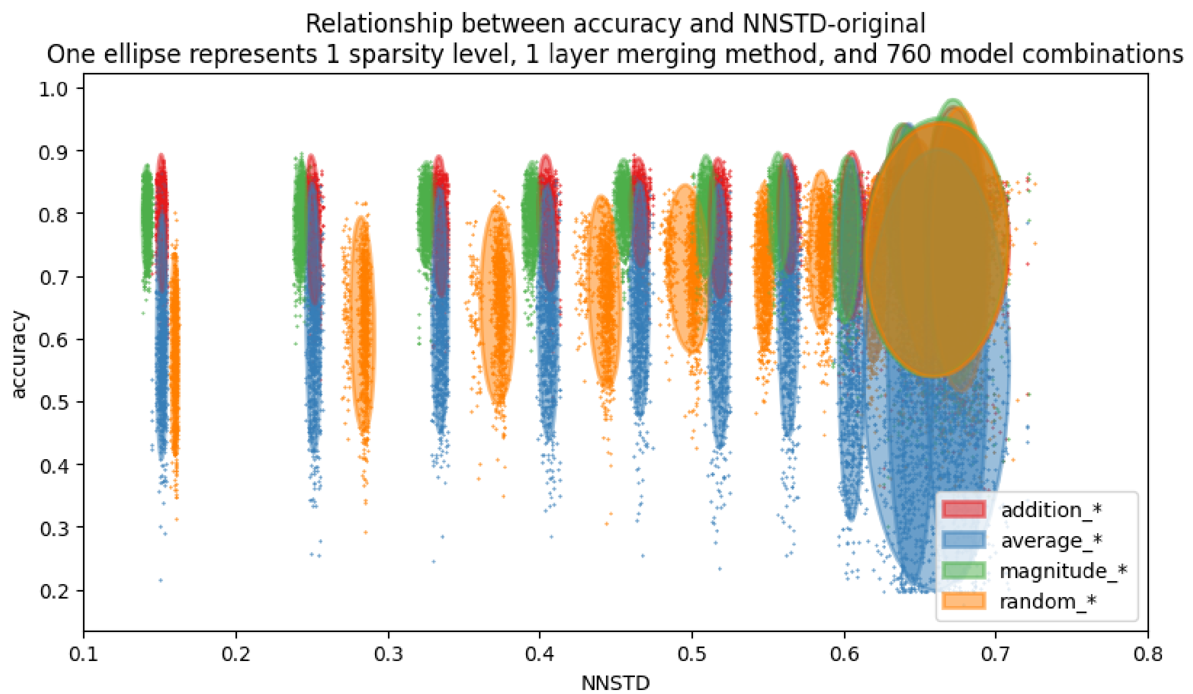
At the higher sparsity levels, the ellipses start overlapping and the results are difficult to see visualize. Therefore, the results for sparsity level 0.90, 0.95, and 0.99 are plotted more clearly in Figure 20. At sparsity levels 0.90 and 0.95, the pattern that can be seen at lower sparsity levels starts to disappear. The ellipses for *magnitude*, *addition*, and *random* almost overlap. For sparsity level 0.99, these three ellipses overlap completely. This is in line with the explanation in section 4.2. Sparsity level 0.99 has results spread out over the same range as sparsity levels 0.90 and 0.95 combined, both over accuracy and NNSTD-original. As stated before, the overlap is explained in 'Merging method similarity' on page 21.

NNSTD overlap between sparsity levels 0.90 and 0.99 Figure 20 zooms in on the accuracies and NNSTD values at the highest sparsity levels. The figure shows that for both sparsity levels 0.90 and 0.99 the NNSTD ranges between 0.60 and 0.70. Sparsity level 0.99 has a larger spread than 0.90. This contradicts the expected NNSTD values predicted in section 'Expected NNSTD between two SNNs' on page 18. NNSTD values not conforming to expectation have already been discussed before, but overlapping NNSTD values have not. How can it be that for some networks at sparsity level 0.99, they have a lower NNSTD value than some networks at sparsity level 0.90.

One explanation for this can be found again in the feature selection phenomenon as discussed above, in combination with the number of training epochs. Figure 10(a) shows that the networks at sparsity level 0.99 train at least twice as long as the network at sparsity level 0.95. This means that these networks also go through twice the evolution steps. This in turn means that these networks have more opportunities to connect weights to the most important features, and thus lower their NNSTD.



(a) Relationship between accuracy and NNSTD-original. One ellipse represents 1 sparsity level, 1 merging method, and 190 network combinations. An ellipse covers a confidence interval of 2 standard deviations. It's important to notice that the circles tend to overshoot, and that the top of a circle does not indicate the highest accuracy.



(b) Relationship between accuracy and NNSTD-original. One ellipse represents 1 sparsity level, 4 merging methods with the same layer merging method, and 760 network combinations. An ellipse covers a confidence interval of 2 standard deviations. It's important to notice that the circles tend to overshoot, and that the top of a circle does not indicate the highest accuracy.

Figure 19

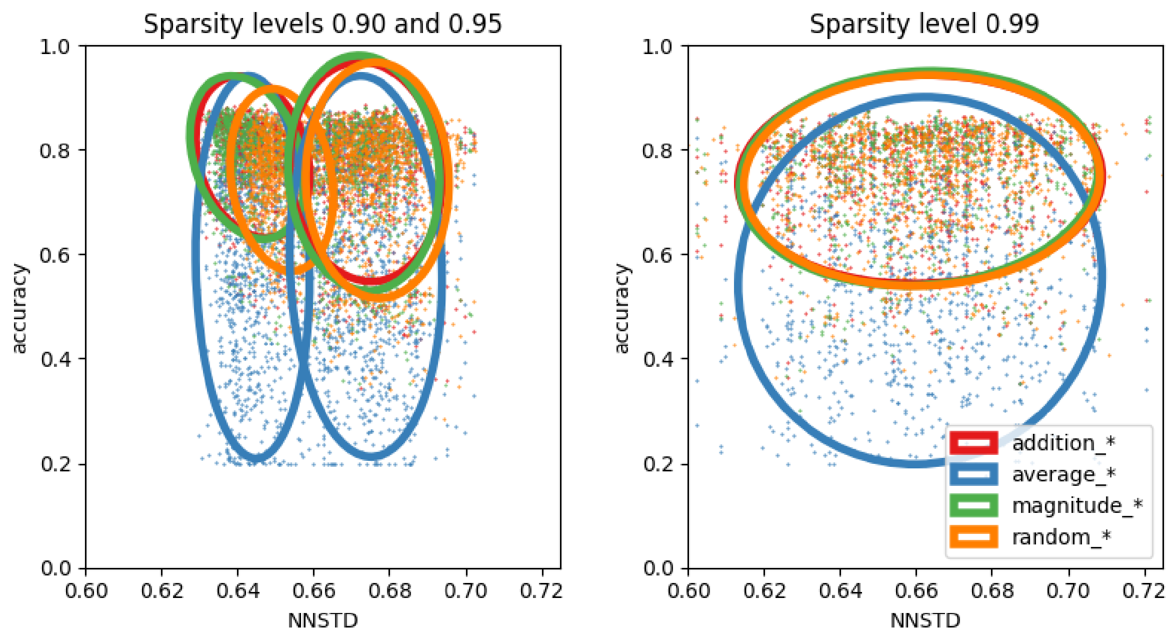
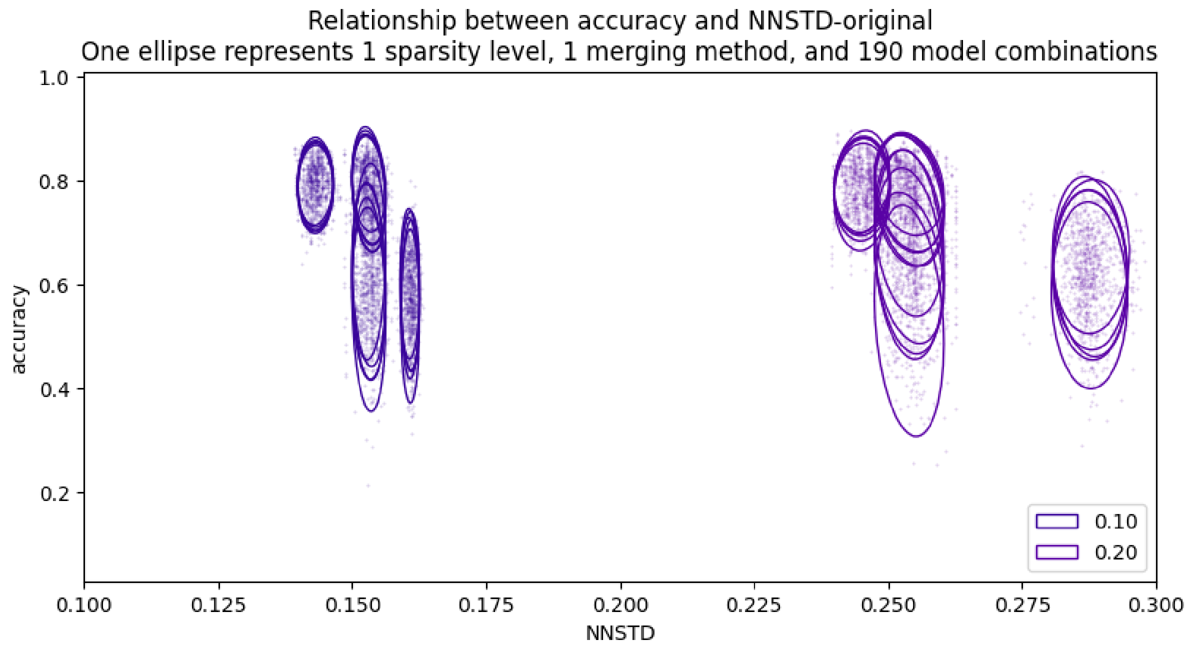
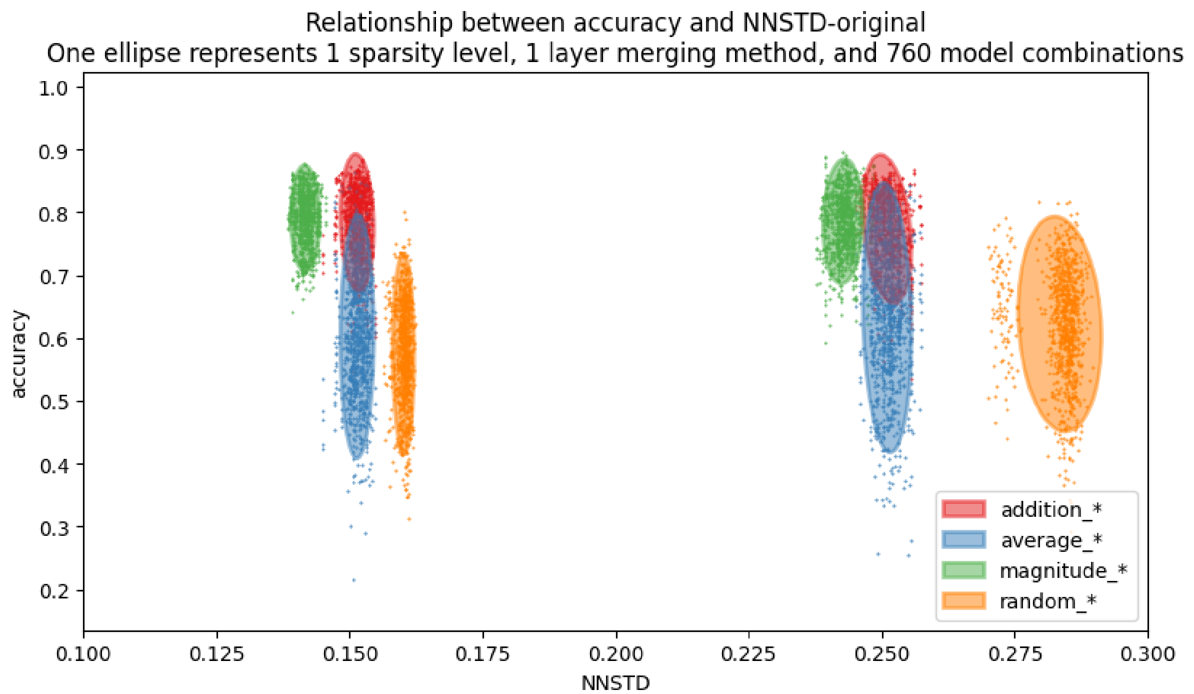


Figure 20: Relationship between accuracy and NNSTD-original. One ellipse represents 1 sparsity level, 4 merging methods with the same layer merging method, and 760 network combinations. An ellipse covers a confidence interval of 2 standard deviations.



(a) For all sparsity levels and merging methods, average accuracy over 210 networks



(b) For all sparsity levels and merging methods, average accuracy over 210 networks

8 Phase 2 - Extended research

Phase 2 has shown that the best layer merging method is *magnitude_**. There is a possibility that the merged network resembles one parent much more than the other. The high resemblance to one of the parents, which has a good performance, could explain the good performance of the merged network. A reason for the merged network having a high resemblance to one of the parents could be that one parent has weights with a relatively strong magnitude compared to the other parent. If the magnitude merging technique is chosen, the weights from that one parent would then be preferred over the weights of that other parent. To combat this, one could apply normalization to the weights of all networks before merging. The normalization of weights might give a more fairly distribution of selected weights over both parents.

Bias-variance tradeoff When training a neural network, it is important that a good balance between underfitting (bias) and overfitting (variance) is achieved. Underfitting results in networks that do not adequately capture the relation between input and output. Overfitting results in networks that model the given data, but not the underlying patterns. It remembers the mapping from input to output instead of learning the relationships between input and output.

Bias variance decomposition shows that the loss of a network can be decomposed into three components: bias, variance, and noise. For the mathematics behind this, I would like to point the reader to the book Pattern Recognition and Machine Learning 2006, section 3.2 The Bias-Variance Decomposition, formula 3.41 [6]. Loss from noise is inherent to the dataset, and can not be eliminated. That leaves the loss from bias and variance. A tradeoff, or good balance, has to be made, which is called the bias-variance tradeoff. Since neural networks are prone to overfitting, one will often work with regularization methods. These can be applied to control overfitting in models. Examples are the L1 norm and L2 norm that apply penalties to the loss function based on the value of the weights, and stopping training when the loss on the test set increases. Two other regularization methods, that both touch the same subject as merging neural networks, are *Ensemble* and *Dropout*.

Ensemble and Dropout Ensemble [22] is a regularization method that reduces overfitting. Multiple networks are trained, after which all networks are used for inference. The output of all networks is then averaged to get the final output. This specific form of Ensemble is also called *Bagging*. This method only works if the outputs are not highly correlated. This can be accomplished by either picking different subsets of data for training, or by training networks with different topologies. Either the training data, the network topology, or both, need to differ between networks. The averaging leads to less variance and thus to better generalization. Even if all networks would be able to perfectly model the (specific subset of) data (extreme overfitting), the averaging of these models would still converge towards the underlying patterns in the data.

Dropout [38] is a regularization method that works by randomly disabling a subset of weights during training with probability p. In essence, every time a different subset of weights is disabled, a network with a different topology is trained. During inference, all weights in the network are enabled and scaled to simulate the averaging of all these networks with different topologies. The averaging closely resembles the Ensemble method

explained above. The Ensemble method averages the output of many networks, whereas Dropout averages the networks themselves. The original paper shows a performance increase with Dropout, and the method is well-known and widely used.

This extended research is meant to look into how normalization affects the merging of networks, and if merging more than two networks gives observations similar to those found in Dropout.

8.1 Methodology

As explained in the previous paragraph, Dropout is able to increase performance by averaging multiple models into one. This is comparable to phase 2, where two models are merged into one. An important difference is that with Dropout, all networks are represented equally. This is not guaranteed with the merging methods from phase 2. As explained, all merging methods except for *average_** can result in unequal representation. For both *magnitude_** and *addition_**, this is caused by the weights of one network being much stronger than the other network. To combat this, new networks have been trained while continuously being normalized.

Normalization Since various definitions of normalization exist, the exact formula will be given:

$$\text{norm}(x) = \frac{x}{\sqrt{\sum(x_1^2, ..., x_n^2)}}$$

where x represents a vector of n weights. In words, the euclidean distance of the vector x is set to 1. A hidden layer is a matrix of weights, and the normalization can be applied either column-wise or row-wise. This also depends on the way the weights are stored in memory, and is therefore application-specific. In this experiment, weight vectors have been normalized such that all incoming weights of a neuron have euclidean distance 1. The other option would have been to normalize all outgoing weights of a neuron. There was no reason to choose one method over the other. To prevent underfitting, the biases have not been normalized.

The same sparsity levels as in phase 1 and 2 have been used. For each sparsity level, 5 networks have been trained. The data, network architecture, and hyperparameters are the same as in phase 2 (see 'Methodology' on page 30). The only difference is that early stopping has been disabled and that after each training iteration, the model was normalized. After merging, the resulting network is pruned back to the target sparsity, but not again normalized.

Initially, the plan was to reuse the networks trained in phase 2 by normalizing these. However, normalization had a strong impact of the performance of these models, which lead to the decision to train new models. The impact of normalization can be seen in Figure 22.

8.2 Training results

Figure 23 shows the training results for the 12x5 networks over all 1500 epochs.

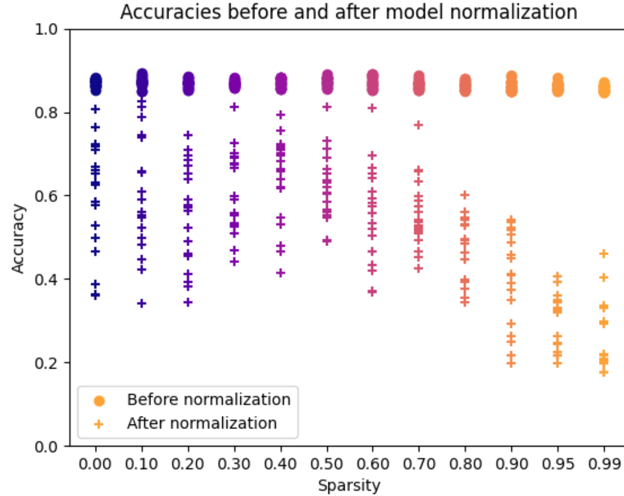


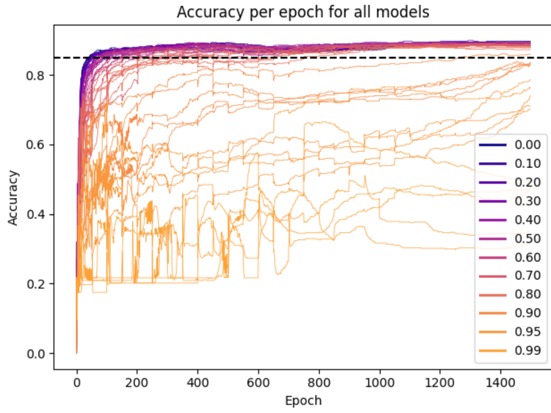
Figure 22: Accuracy of the 12x20 networks of phase 2 before and after applying normalization. The graph shows an average drop in accuracy of at least 30%, with some of the sparser networks showing performance no better than guessing (20%).

Comparison between non-normalized and normalized networks The first observation is that at lower sparsities, the normalized networks do not suffer from the instability that the non-normalized networks suffer from. A reason for this might be that the normalization prevents the weights from exploding. Interestingly, both the normalized and non-normalized networks at lower sparsity seem to reach 85% accuracy around epoch 50. It seems that under these circumstances, the normalization does not interfere with the capability of the networks to learn, and prevents overfitting as soon as the networks reach the assumed accuracy limit.

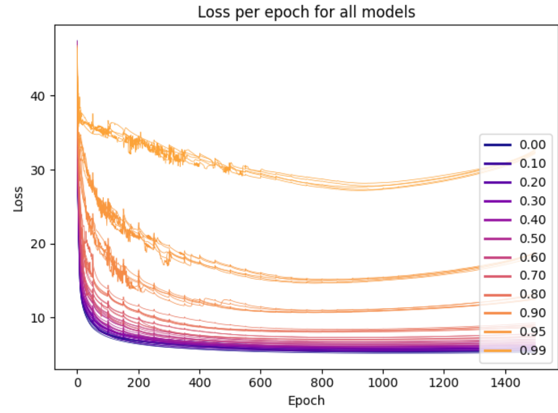
The second observation is that the normalized networks at low sparsities can reach a higher final accuracy compared to the non-normalized networks. Whereas the non-normalized networks have a hard time going over 85% accuracy, the normalized networks can reach 90% accuracy. The possibility that the non-normalized networks might have reached 90% accuracy as well can not be excluded, were it not for stopping the training early.

The third observation is that the loss between the normalized and non-normalized networks is comparable. Evolution seems to have a stronger effect on the normalized models, which clear peaks being observed in Figure 23(d).

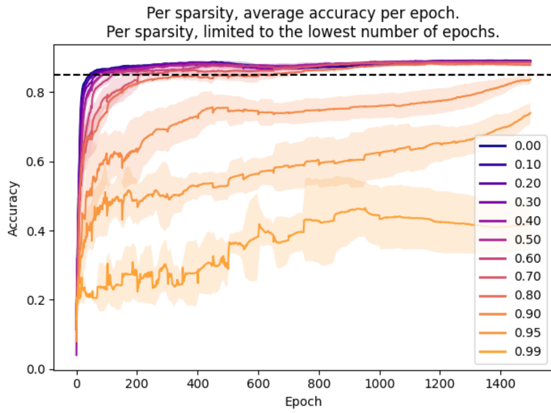
Performance loss at the end of training Starting somewhere between epochs 700 and 900, the losses of networks at the higher sparsity levels start increasing again. This can be seen in Figure 23(d). An explanation for this can be that normalization has a detrimental effect on the accuracy, as was already demonstrated in Figure 22. At the beginning of training, the learning rate is high enough to compensate for the normalization. The learning rate drops during training (see Figure 8), and at some point, the normalization overwhelms the capability of the learning rate to compensate. It is noteworthy that when looking at sparsity levels 0.90 and 0.95, the losses increase after around epoch 800, but the accuracies keep improving, at an increasing rate even. This can be seen by comparing Figure 23(c) and Figure 23(d).



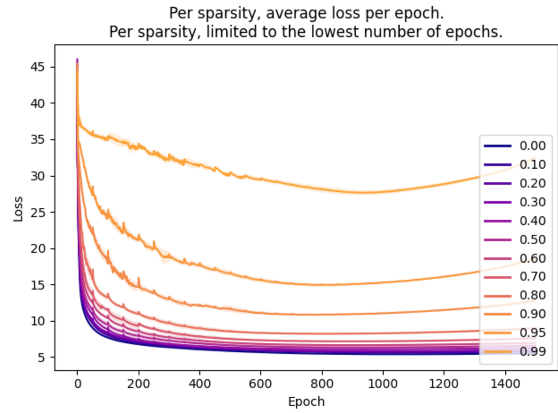
(a) Accuracy over all networks and epochs



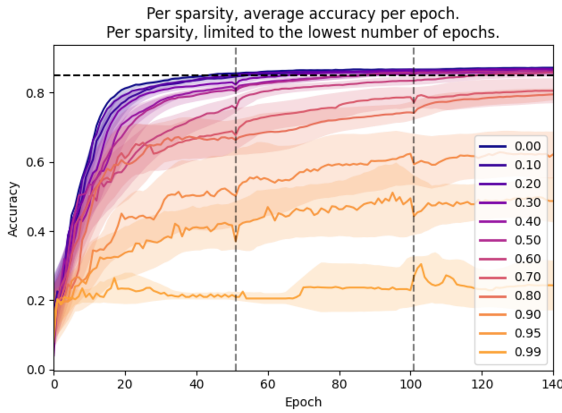
(b) Loss over all networks and epochs



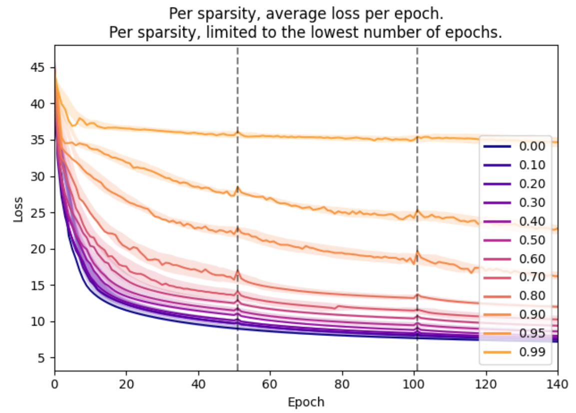
(c) Average accuracy over all epochs



(d) Average loss over all epochs



(e) Average accuracy over all epochs



(f) Average loss over all epochs

Figure 23: Per sparsity, accuracy (left column) and loss (right column) per epoch.

8.3 Merging results with two networks

8.3.1 Impact of layer merging method and bias merging method

Compared to the heatmap in phase 2 (Figure 13), the heatmap from this extended research in Figure 24 does not provide much insight. Accuracies for *average_**, *magnitude_**, and *random_** are 75%, with 77% for *addition_**. Looking at the loss, *addition_** performs best, but only slightly. *Addition_** seems to be the best merging method, but barely. Figure 26(a) gives more detailed results of the merging results. An important observation compared to phase 2 is that the bias merging method and layer merging method has basically no impact on either the accuracies or losses. At higher sparsities, the layer merging method has slightly more effect. Interestingly, looking at Figure 26(a), the merging method *average_** shows slightly superior performance at denser networks, but inferior performance at sparser networks.

8.3.2 Performance and NNSTD-Original

Again, we can compare the results of merging the normalized networks with the results of phase 2.

Accuracy spread at lower sparsity levels Looking at the NNSTD graphs in Figures 19(b) and 25(b), the first observation is that the spread of accuracies is much lower at the normalized networks for the more dense networks. This seems to apply for the methods *average_** and *random_**. One explanation might be that when averaging two vectors with the same norm, or picking random weights from two vectors with the same norm, will result in a new vector that also has roughly the same norm. This was not the case when non-normalized vectors were merged in phase 2. Less difference between the norm of the parents and the child might help keep merged networks stable.

Merging methods converging In phase 2, the merging methods *addition_**, *average_**, and *magnitude_** would converge to the same results as sparsity levels increased. The same effect can be observed in the results of this extended research for the NNSTD values. At the highest sparsity level, the ellipses seem once again to overlap.

Addition_* outperforming *Magnitude_* An interesting observation is again how well the *addition_** merging method performs. In phase 2, Figure 19(b), *magnitude_** and *addition_** were competing for the best performance. At all sparsity levels, their ellipses covered the same accuracy interval. These results in Figure 25(b) however, show that *addition_** outperforms *magnitude_** up to around sparsity level 0.80, even though it also shows a higher NNSTD. The networks merged with *addition_** have a topology that is further away from its parents compared to networks merged with *magnitude_**, and still these networks outperform.

8.4 Merging results with five networks

All experimentation has been focused on merging two neural networks to keep the analysis as simple as possible. To get better insight into possible future work, and to test the ensemble method that dropout employs, more than two networks have been merged. For each sparsity level, all five networks have been merged together using all 5x5 merging

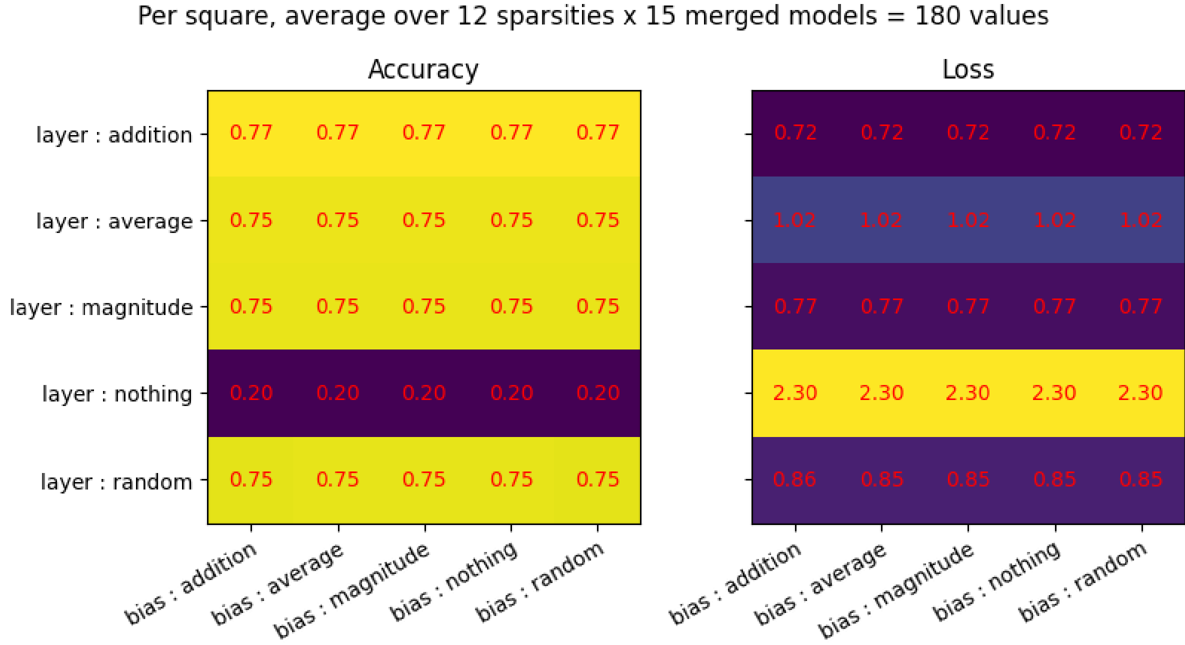
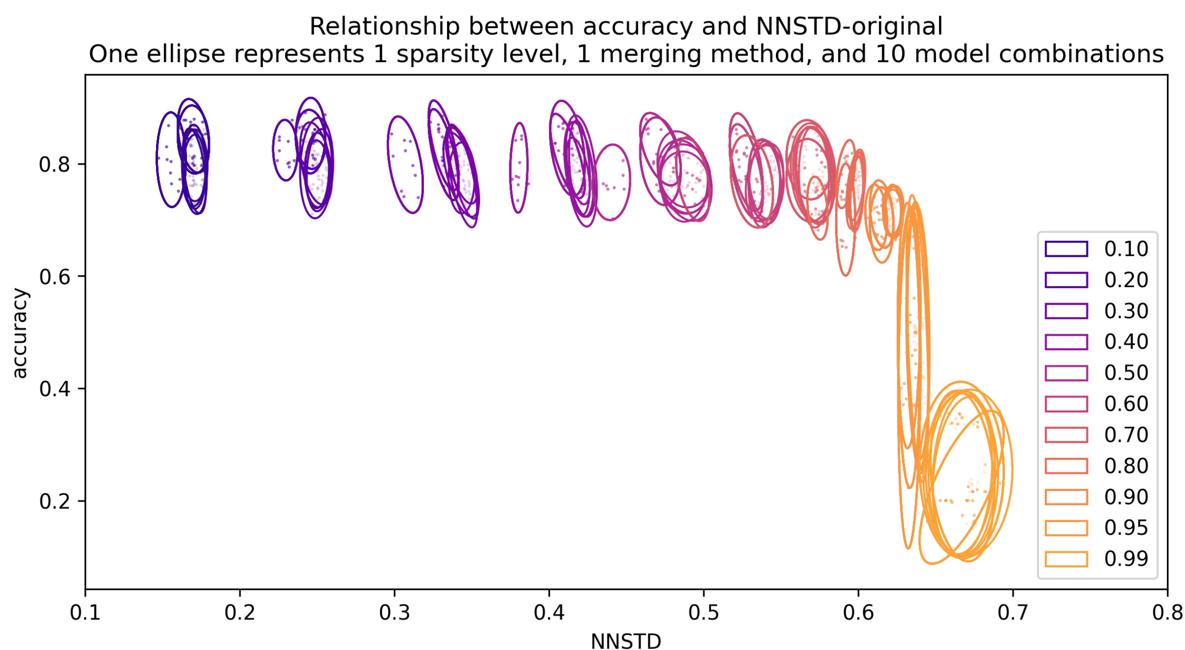


Figure 24: Each square represents a single merging method. For each merging method, the average accuracy (left) or loss (right) over 12 x 15 = 180 networks is given. ”layer” indicates the layer merging method used, and ”bias” indicates the bias merging method used.

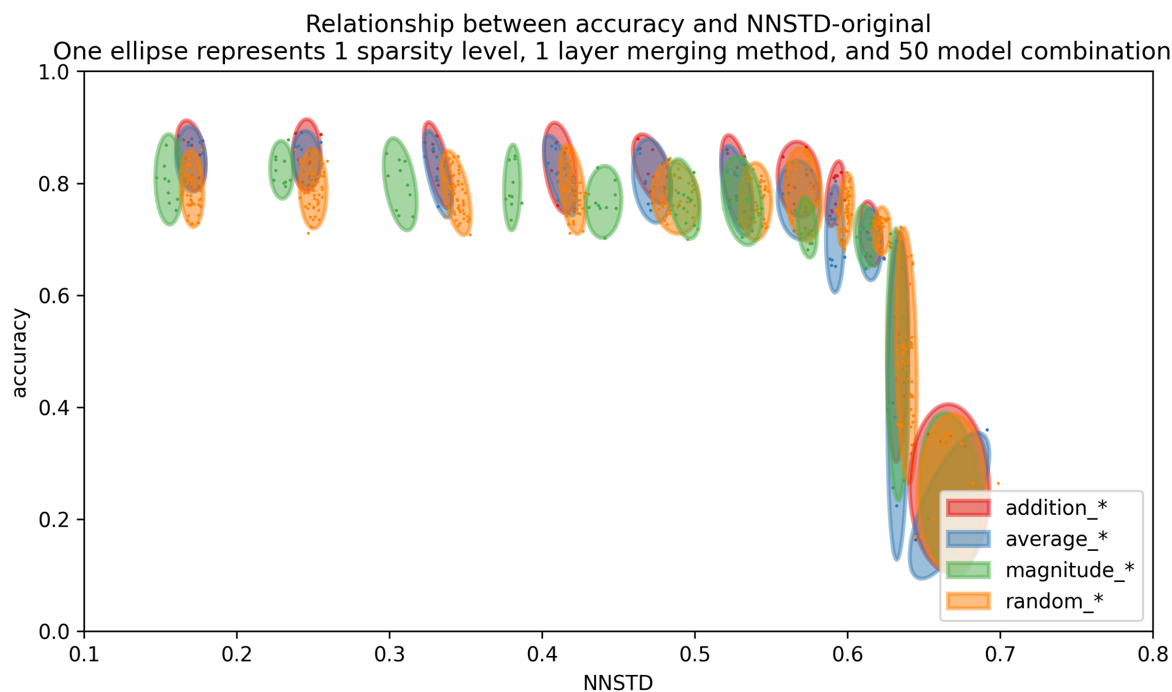
methods, resulting in 25 merged networks per sparsity level, for a total of 300 merged networks. After this, there are two choices to be made. The resulting merged networks can again be normalized and again be pruned to the target sparsity. All four distinct combinations have been applied to the 300 networks, resulting in a total of 1200 merged networks. The results are shown in the four parallel coordinate plots in Figure 27.

Difference between merging two networks and five networks To compare the merging of two networks and five networks, we have to look at the plot where the merged models are pruned and not normalized, Figures 26(a) and 27(b). The results are quite clear. There is no instance where merging five networks outperforms merging two networks. Especially with the merging method *average_**, that performed the best when merging two networks, the performance has dropped significantly. At the merging method *random_**, the bias merging method seems to suddenly have a large impact again, with sometimes even a 20% difference in accuracy between bias merging methods. *Addition_** seems to be the least affected. Interestingly, at sparsity level 0.95 and merging method *addition_**, the network drops to around 15% accuracy, meaning that it performs worse than guessing. Unfortunately, the observations made in the methods Ensemble and Dropout, that combining networks could improve performance, can not be seen in these results. However, these methods improve performance by reducing overfitting. If no significant amount of overfitting occurs, then it follows that there will be no significant performance boost.

Differences between (not) applying normalization and pruning In Figure 27, the top two plots show the non-normalized networks, and the bottom two plots show the

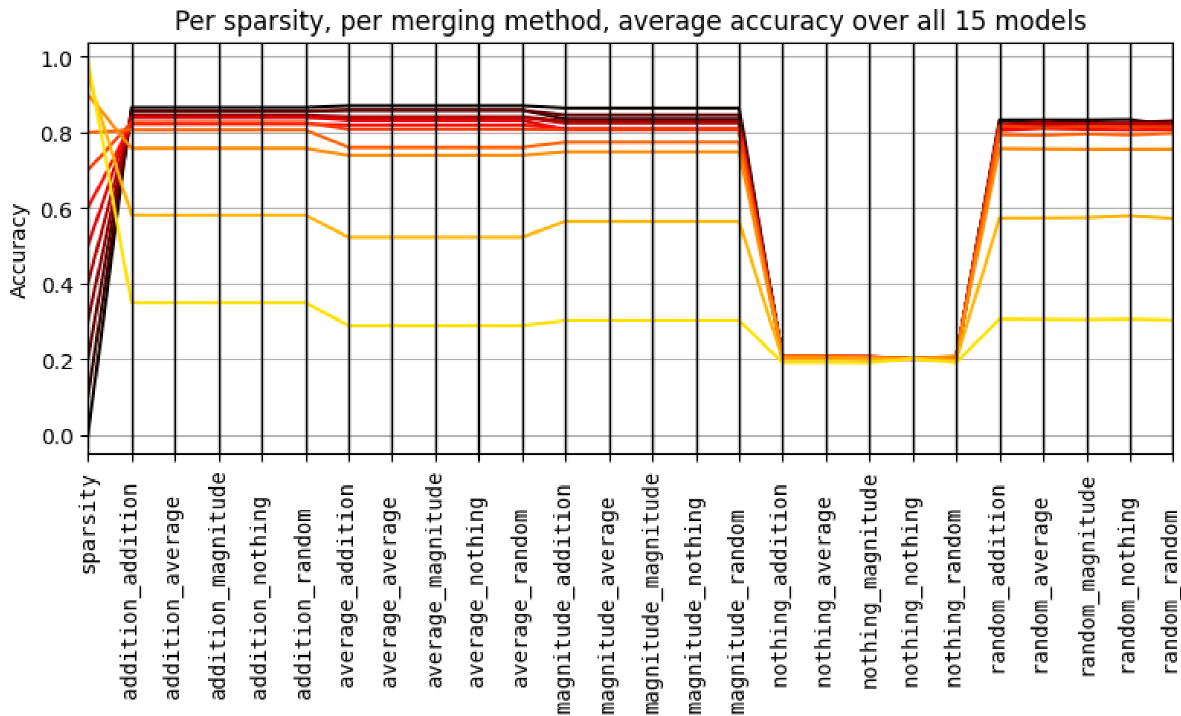


(a) Relationship between accuracy and NNSTD-original. One ellipse represents 1 sparsity level, 1 merging method, and 10 network combinations. An ellipse covers a confidence interval of 2 standard deviations. It's important to notice that the circles tend to overshoot, and that the top of a circle does not indicate the highest accuracy.

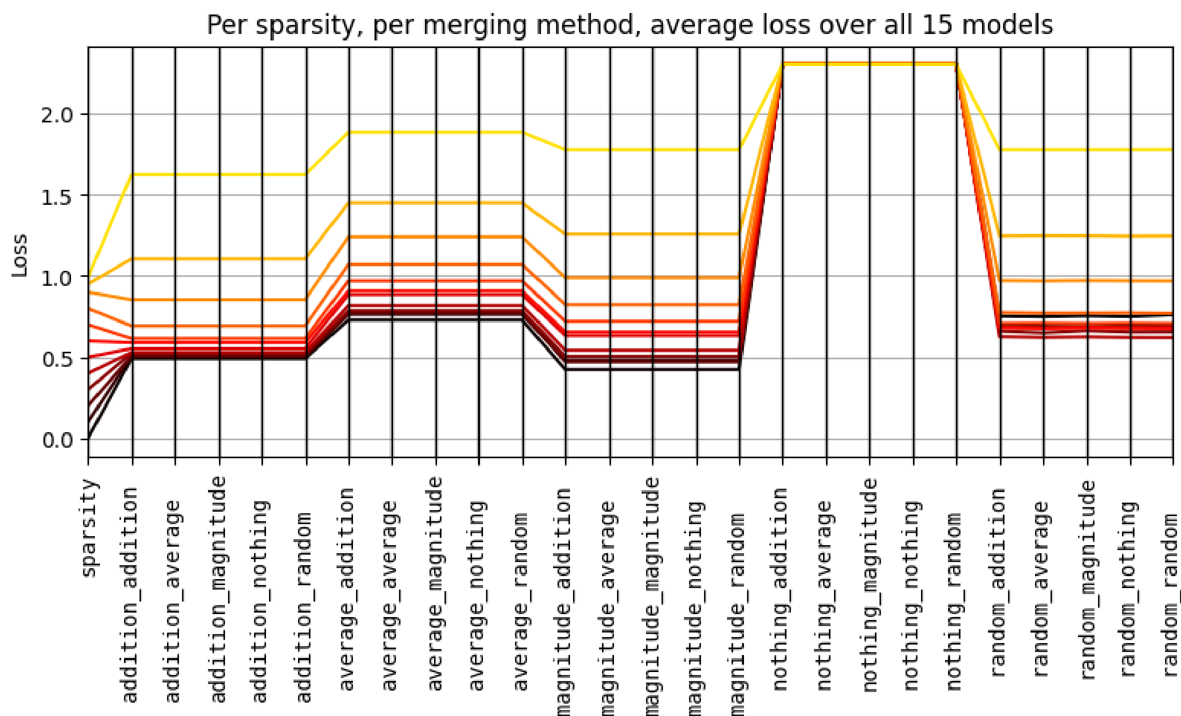


(b) Relationship between accuracy and NNSTD-original. One ellipse represents 1 sparsity level, 4 merging methods with the same layer merging method, and 50 network combinations. An ellipse covers a confidence interval of 2 standard deviations. It's important to notice that the circles tend to overshoot, and that the top of a circle does not indicate the highest accuracy.

Figure 25



(a)



(b)

Figure 26: For all sparsity levels and merging methods, average accuracy (top) and loss (bottom) over 15 merged networks

normalized networks. With *addition_**, there is no significant difference between the two. With *average_**, normalization shows a clear advantage over the non-normalized models. With *magnitude_** and *random_**, there is no clear pattern.

Looking at the difference between pruning and not pruning, applying pruning seems to lead to slight to strong performance losses. Especially the sparser models see a strong dip in performance. The strongest performance dips can be seen at *magnitude_**. *Addition_** is quite indifferent to pruning, incurring only slight performance losses.

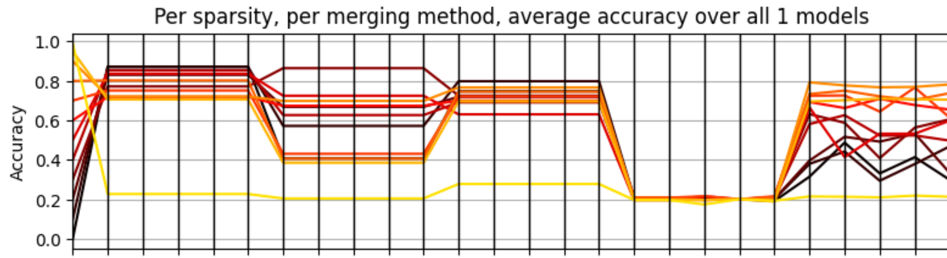
8.5 Conclusions extended research

Training The extended research has shown, as a first, that continuously applying normalization when training sparse neural networks can result in better performance. It is important that the learning rate is high enough to compensate for the performance hit taken by normalization. Normalization also helps denser models stay stable during training.

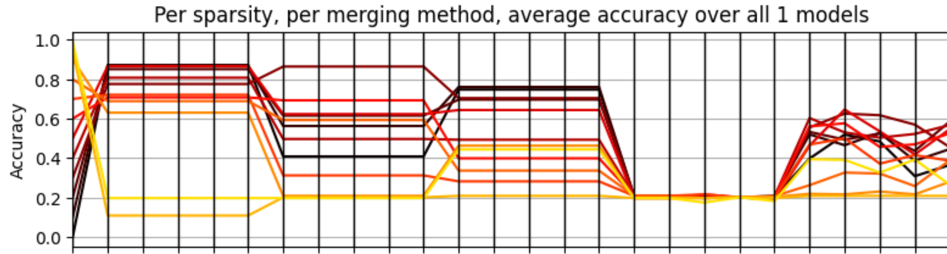
Merging Compared to the initial research of phase 2, *magnitude_** is not the best merging method anymore. When merging normalized networks, looking at merging both two and five networks, *addition_** is the overall best choice, regardless of resparsification and renormalization. Given that normalization produces such good results, one might expect that *addition_** might result in weight vectors with new norms that are just too strong, but this does not seem to be the case.

Bias-variance tradeoff Ensemble and Dropout have shown to be able to give a performance boost. Looking at all the merging results in this research, merged networks have not once given a performance boost over their parent networks. A reason for this might be that these methods specifically reduce overfitting. The reason that this performance boost has not been seen when merging networks, is that none of the networks ever suffered from overfitting. This can be confirmed by looking at the plots illustrating the loss on the test set during training. The only time loss ever goes up is when the highly sparse networks in the extended research suffer from performance loss due to normalization. This is exacerbated by the fact that sparse neural networks are already less prone to overfitting. An interesting research question can follow from this, which will be detailed in 'Future Work' on page 62.

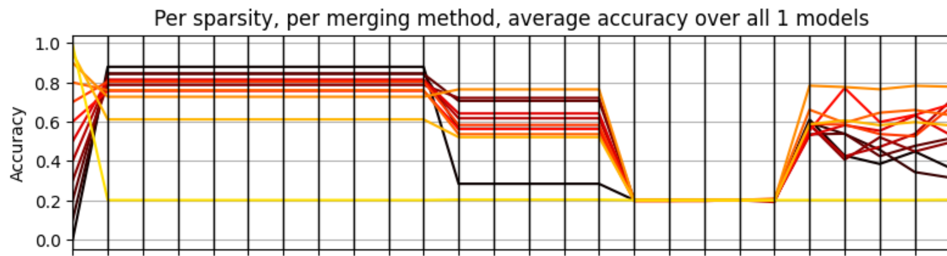
Note that these conclusions do not automatically transfer to any other environment, where different models, hyperparameters, and datasets can be in play.



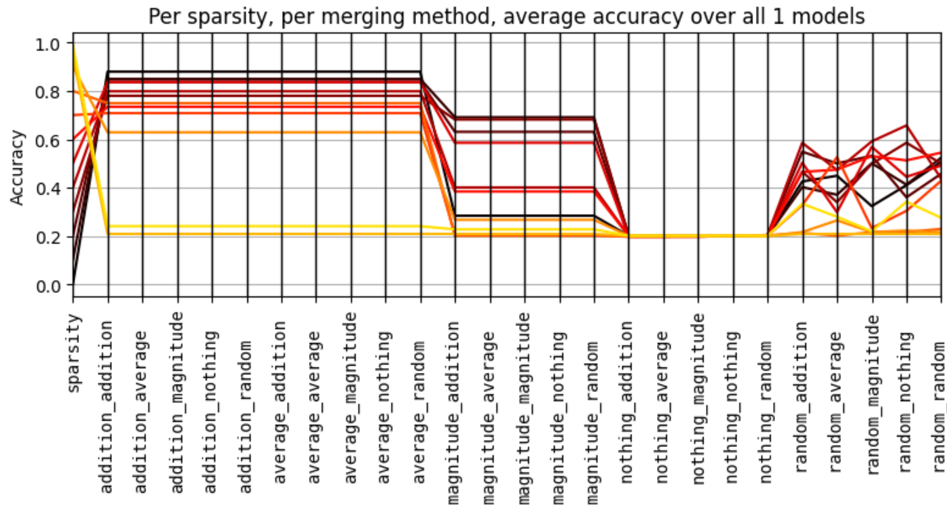
(a) Not normalized, not pruned



(b) Not normalized, pruned



(c) Normalized, not pruned



(d) Normalized, pruned

Figure 27: The four parallel coordinate plots that show the results of merging five sparse neural networks together. Each plot represents one of the four distinct combinations between applying normalization and pruning The color represents the sparsity level. A darker colour indicates a denser model.

9 Discussion and thoughts

9.1 Importance of bias merging method

Phase 1 showed (6.7) that the bias merging method had a larger impact on the performance than the layer merging method. Phase 2 showed that this is true at higher sparsity levels (7.7.2). However, phase 2 also showed that at lower sparsity levels, the choice of layer merging has a larger impact. The extended research of phase 2 where networks were normalized, showed that the bias merging method barely made impact, if any.

9.2 Overall best merging method

Both phases 1 and 2 showed that overall there is no better merging method than *magnitude*. Does the layer merging method *magnitude* only work best because SET uses the weights' magnitude as selection criteria? Since SET uses this criterion, it follows that the weights with the largest magnitude are the most important. If the layer merging method *magnitude* is then used, it once again selects the weights that have already been deemed most important.

The phase 2 extended research showed that when normalizing beforehand, *addition_** is the best merging method. It gives, on average, the best merging results, and is robust to both possible resparsification and renormalization.

What if, instead of selecting the weights with the largest magnitudes, SET selected weights with the smallest magnitude. It might be that a merging method works best if it then also selects the weights with the smallest magnitude, instead of the largest.

9.3 Sparsity level 0.99

Phase 2 has shown that with the current dataset and network structure, networks were able to reach the evident accuracy limit of 87%. This implies that networks could possibly be sparsified even further. Merged SNNs have shown the same performance. If a resparsified merged SNN, with parents trained on two different datasets, has no performance loss, then that must mean that the parents can be sparsified further. The resparsified merged SNN only has parts of those parents, and still works just as well. The resparsified merged SNN could theoretically be separated back into the two parent SNNs. These parent SNNs will now also be more sparse, without performance loss.

9.4 Parallelizing an optimized version of Dropout

Working on the extended research of phase 2 has helped me realize that training and merging sparse neural networks can be seen as a parallel version of Dropout. When Dropout disables weights and goes through a training iteration, a single sparse topology is trained. Merging sparse neural networks allows for the training of many different sparse neural topologies in parallel, which can then be merged back into one.

The Lottery Ticket Hypothesis (LTH) has shown that a dense network can be seen as a conglomeration of many sparse networks. LTH also tells us that when these sparse networks are taken out of the dense network, not all of these sparse networks are able to be trained to a good performance. Simply not all sparse networks are viable, and it has

been shown by LTH that good performance is a combination of topology and random weight initialization that need to 'match'.

The SET algorithm overcomes this limitation by modifying the topology during training, thus searching through the topology space and slowly matching up the topology and weights. Dropout, however, does not modify its topology. Its training iterations might be spent on a topology that is not viable. It is not clear to me if these iterations can be seen as a waste, due to the intense weight sharing within Dropout. Regardless, Dropout will average, or 'merge', all possible sparse neural networks that are present within a dense neural network, many of which might not be viable or have not been trained at all. Merging multiple networks trained with SET, however, ensures that all networks are viable and trained.

One can assume that merging networks trained with SET can be seen as parallelizing an optimized version of Dropout, where inviable and untrained networks are discarded. Unfortunately, the results of this research have not yet laid the foundation for this claim.

10 Future Work

The results provided in phases 1 and 2 spark interesting questions which could be a starting point for further research. Research questions are listed below, with each question being elaborated on in its own subsection. Research questions relating to a specific phase will be prepended with that phase.

1. *Phase 1 : Classification preference and data distribution* How do merging techniques lead to better performance of a merged SNN on one dataset over the other?
2. *Training a merged SNN* Can a merged SNN be trained effectively to increase performance?
3. *Impact of bias on network performance* What is the influence of bias merging techniques on the performance of the merged SNN?
4. *Performance of different merging techniques* How can current research into training SNNs be applied to the merging of two SNNs?
5. *Apply topology transformation before merging* Can two or more SNNs be modified to look more alike before merging, to improve merging performance?
6. *Phase 2 : Improve resparsification by including Sparse Connectivity Pattern* Can resparsification be enhanced by combining information from the Sparse Connectivity Pattern with current selection criteria?
7. *Phase 2 extended research : Exploiting overfitting* Can sparse neural networks be trained differently, by combating overfitting with merging?

10.1 Phase 1 : Classification preference and data distribution

The merged networks have been tested on dataset 3, which contained the classes of both the original networks. What has not been tested is the performance on datasets 1 and 2. It could be possible that the merged network performs well on dataset 1 and poorly on dataset 2. This would result in an average performance on dataset 3. It would be

interesting to see why the merged network prefers one dataset over the other. It could be that one dataset had much simpler classes, fewer classes, and more training samples. Research into this question could give more insight into how to divide datasets over different networks and if two datasets are good candidates for merging.

10.2 Training a merged SNN

The performances of the merged SNNs were assessed immediately after merging. Another option is to first train the merged SNN on the collection of all classes on which the original networks were trained. There is a possibility that the merged SNN converges quickly. The SET algorithm has shown that the topology of an SNN plays a major role in its performance. The merging of two SNNs could result in a topology suited to training.

10.3 Impact of bias sparsification on network performance

Much discussion has revolved around the selection and merging of the non-bias weights. The merging of biases of two networks however, might have an even larger impact on the performance a merged network. As mentioned in 4.2, three out of the four merging techniques are equal when there is no overlap in the weights, which is expected in sparse neural networks. Biases however are not sparse, and there is a 100% overlap between the biases of two networks. The results of phase 2 (7.7.2) have shown that, at high sparsity levels, the way biases are merged can result in a significant performance difference of at least 20% accuracy. The extended research showed however, that the bias merging method barely makes an impact on performance, if any. Research into this question could give insight into the effect of biases on network merging. While bias sparsification might not significantly improve hardware requirements, it might balance out the impact of the layer merging method and bias merging method. Research into sparsifying bias could lead to better performing SNNs at high sparsity.

10.4 Performance of different merging techniques

The networks merged in the preliminary results are the results of combining two networks that are trained on different classes. What has not been tested is the merging of two networks which have been trained on the same network. It would be interesting to see the results of merging such networks. Another way to see this is to remove as many variables from the merging equation as possible. Merge two networks with the same hyperparameters, trained on the same dataset; the only variable left is the different merging techniques. Trying to optimize the performance of this merged network would mean optimizing the merging techniques. This, in turn, leads to an even deeper question: Which weights are most important for classification. This question loops around to research into pruning, where attempts are made to preserve only the most important weights. SET has shown that magnitude is a strong candidate when selecting weights. However, the preliminary results did not necessarily reflect this. Other papers, such as [18], suggest a more complex selection candidate. Saliency, for example, is a combination of the magnitude and gradient of a weight. Research into this question could provide insight into which weights are important for merging into a network. Since there is already a significant amount of research in this direction, one could look at current existing techniques and try to apply these to merging.

10.5 Apply topology transformation before merging

The topology of two SNNs might seem different at first sight but could be identical, otherwise known as isomorphic. Isomorphism does not care about values of vertices (weights in the case of an SNN). It only matters if a weight is present or not in a certain position. Suppose two graphs have a different topology but are isomorphic. In that case, both graphs can be modified to end up with the same topology without changing any of the graphs defining characteristics. In terms of an SNN, this would mean that weights and neurons can be swapped around without impacting the network's performance.

In this research, the SNN pairs have not been checked for isomorphism before merging. If this had been done, performance loss could possibly have been prevented. Even in the case where two SNNs are not exactly the same, they could be modified to resemble one another as much as possible. The first step would be to modify the network so that there is as much weight overlap as possible. This would reduce the effect described in 4.2, where merging methods start to give the same results. The second step would be to line up weights that approximately have the same value. Merging weights with, for example, opposite values v and $-v$ would either cancel out one or both. Aligning weights with the same value should result in a better performance.

10.6 Phase 2 : Improve resparsification by including Sparse Connectivity Pattern

Previous research has shown that SNNs in combination with SET act as a feature selector [27]. The most important inputs will end up with the largest number of weights connected to them. When two SNNs are merged, the resulting SNN has an equal or lower sparsity level. To retain sparsity, the merged SNN will have to be resparsified again. In this research, sparsification was done by dropping the weights with the lowest magnitude, in accordance with the SET assumption that the weights with the largest magnitude are the most important.

The sparse connectivity pattern of both parents could be combined with the magnitude of the weights to improve sparsification. Given an SNN that needs to be sparsified, it might have a weight that has a large magnitude, but is connected to an input that is deemed unimportant by the sparse connectivity pattern. The sparsification algorithm might then opt to instead keep a weight with a slightly smaller magnitude but larger importance. Selecting weights like this could improve performance.

10.7 Phase 2 extended research : Exploiting overfitting

Phase 2 extended research touched on applying the principles from the Ensemble and Dropout methods to merging neural networks. It has shown that merging networks does not necessarily lead to a drop in performance, but never leads to an improvement. According to the *bias-variance tradeoff* as well as looking around at existing work with *Ensemble* and *Dropout*, a performance boost is expected. Both these methods increase performance by reducing overfitting. This suggests that overfitting is not present in the models trained for this research. Thus no performance boost is to be gained. Are there situations where we can gain something by allowing overfitting, which can then be combated by merging?

To purposefully devise overfitting, one could decrease the training data size. The expectation is that a certain degree of overfitting can be compensated by merging the trained networks. The decrease in data can lead to even lower hardware requirements and faster training times.

11 Appendix

11.1 Reproducibility considerations and code bugs

200 epochs timeout In phase 2, training of networks was stopped 200 epochs after a certain accuracy threshold was reached. This significantly improved training time without (significantly) impacting performance. However, the denser networks were still quite unstable, and dipped below this threshold multiple times within these 200 epochs. At such a point, the 200 epoch interval should be reset, once again waiting 200 epochs. This was not done however. Once the threshold was passed, the network would stop training after 200 epochs regardless of the accuracy. This could have negative impact on performances. It should not impact the results of this research, since networks were trained to their limit. Regardless, it should be taken into account if the code is used for other research.

Proper weight deactivation In a sparse setting, weights that are deactivated and removed from memory. In this research, sparsity was simulated by keeping all weights in memory (including deactivated ones), and applying a binary mask to the weights after training. However, if at evolution time a weight is deactivated and immediately randomly activated again, the mask is not applied to that weight, and the weight keeps its magnitude. To properly simulate sparsity, the weight should have been reset to 0 or a random value depending on the weight initialization procedure. By not properly resetting weights in such cases, networks might receive an unfair performance advantage, because they receive a 'new' weight that has already been trained. Again, it should not impact the results of this research, since networks were trained to their limit. Regardless, it should be taken into account if the code is used for other research.

11.2 Phase 1

Table 7: Performance of original networks with $\epsilon = 0.1$

dataset	dataset 1		dataset 2		dataset 3	
network	accuracy	loss	accuracy	loss	accuracy	loss
1	0.74	0.70	0.00	12.98	0.37	6.84
3	0.00	11.52	0.43	1.16	0.22	6.34
5	0.28	1.71	0.58	1.25	0.43	1.48

Table 8: Performance of original networks with $\epsilon = 0.5$

dataset	dataset 1		dataset 2		dataset 3	
network	accuracy	loss	accuracy	loss	accuracy	loss
7	0.81	0.51	0.00	11.78	0.40	6.15
9	0.00	12.21	0.89	0.31	0.45	6.26
11	0.77	0.67	0.80	0.54	0.78	0.60

Table 9: Merging performance for $\epsilon = 0.1$ and SET on dataset 3

bias	magnitude		average		addition		random	
weights	accuracy	loss	accuracy	loss	accuracy	loss	accuracy	loss
magnitude	0.21	7.91	**0.30**	2.52	0.18	3.53	0.24	6.78
average	0.10	8.52	0.20	**2.39**	0.14	3.32	0.10	11.01
addition	0.21	7.97	0.28	2.52	0.18	3.59	0.10	17.14
random	0.21	7.91	**0.30**	2.52	0.18	3.73	0.10	13.33

Table 10: Merging performance for $\epsilon = 0.5$ and SET on dataset 3

bias	magnitude		average		addition		random	
weights	accuracy	loss	accuracy	loss	accuracy	loss	accuracy	loss
magnitude	0.26	3.79	0.31	2.18	**0.35**	2.59	0.27	4.25
average	0.19	6.03	**0.35**	**1.92**	0.16	4.61	0.18	3.73
addition	0.25	3.74	0.31	2.28	**0.35**	2.58	0.12	5.09
random	0.26	3.77	0.33	2.14	0.34	2.70	0.25	4.46

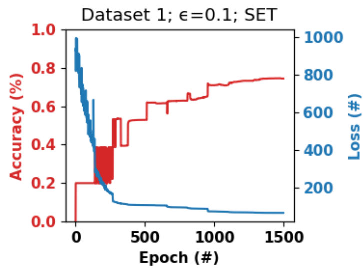


Figure 28: Network 1

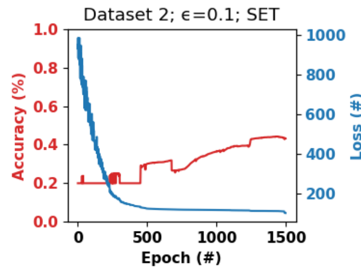


Figure 29: Network 3

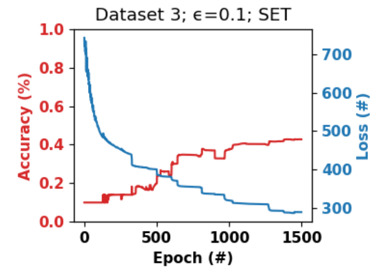


Figure 30: Network 5

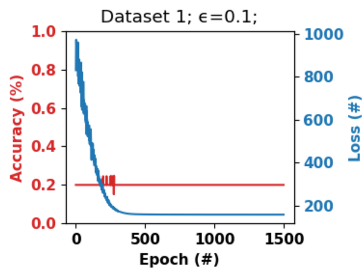


Figure 31: Network 2

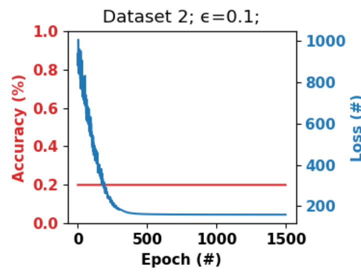


Figure 32: Network 4

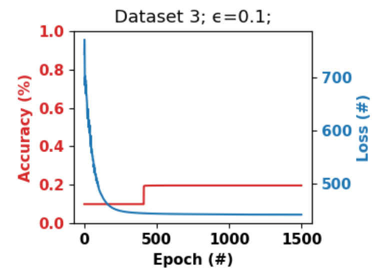


Figure 33: Network 6

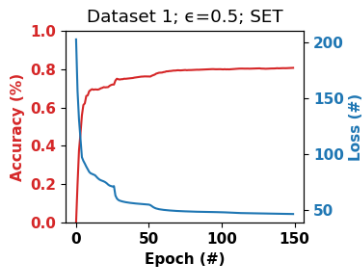


Figure 34: Network 7

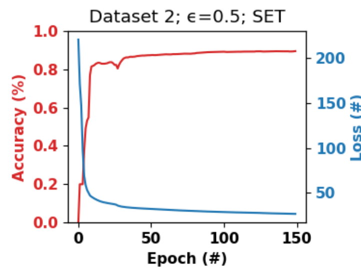


Figure 35: Network 9

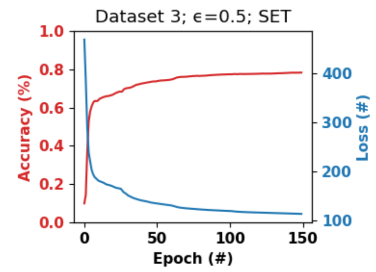


Figure 36: Network 11

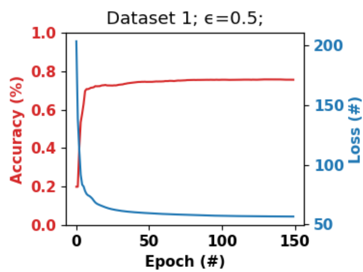


Figure 37: Network 8

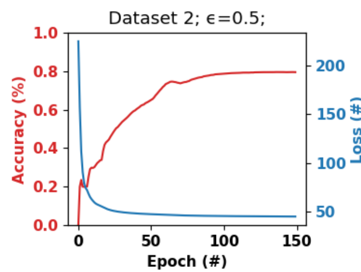


Figure 38: Network 10

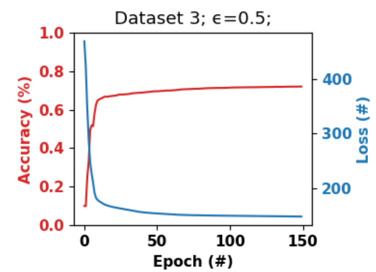


Figure 39: Network 12

References

[1] *A Closer Look at AlexNet*. https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/tutorials/tut6_slides.pdf. (Accessed on 11/13/2020).

[2] AISmartz. *CNN Architectures Timeline (1998-2019)*. https://www.aismartz.com/blog/cnn-architectures/. Oct. 2019.

[3] *Amazon SageMaker – Managed Distributed Training for Machine Learning – Amazon Web Services*. https://aws.amazon.com/sagemaker/distributed-training/.

[4] Tal Ben-Nun and Torsten Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *CoRR* abs/1802.09941 (2018). arXiv: 1802.09941. URL: http://arxiv.org/abs/1802.09941.

[5] *Benchmark dashboard*.

[6] CHRISTOPHER M. BISHOP. *PATTERN RECOGNITION AND MACHINE LEARNING*. SPRINGER-VERLAG NEW YORK, 2016.

[7] Jason Brownlee. *Ensemble Learning Methods for Deep Learning Neural Networks*. https://machinelearningmastery.com/ensemble-methods-for-deep-learning-neural-networks/. Dec. 2018.

[8] Laura Castañón. *deep-neural-networks-are-coming-to-your-phone-heres-how-that-could-change-your-life*. https://news.northeastern.edu/2020/01/28/deep-neural-networks-are-coming-to-your-phone-heres-how-that-could-change-your-life/. Jan. 2020.

[9] Danny Hernandez Dario Amodei. *AI and Compute*. https://openai.com/blog/ai-and-compute/.

[10] *Distributed training of deep learning models on Azure - Azure Architecture Center — Microsoft Docs*. https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/ai/training-deep-learning.

[11] Utku Evci et al. "Rigging the Lottery: Making All Tickets Winners". In: (2020). arXiv: 1911.11134 [cs.LG].

[12] Karl Pearson F.R.S. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 2.11 (1901), pp. 559–572. DOI: 10.1080/14786440109462720. eprint: https://doi.org/10.1080/14786440109462720. URL: https://doi.org/10.1080/14786440109462720.

[13] Jonathan Frankle and Michael Carbin. "The Lottery Ticket Hypothesis: Training Pruned Neural Networks". In: *CoRR* abs/1803.03635 (2018). arXiv: 1803.03635. URL: http://arxiv.org/abs/1803.03635.

[14] Keisuke Fukuda. *Technologies behind Distributed Deep Learning: AllReduce — Preferred Networks Research & Development*. https://tech.preferred.jp/en/blog/technologies-behind-distributed-deep-learning-allreduce/. (Accessed on 04/03/2021). July 2018.

[15] Geoffrey Hinton, Simon Osindero, and Yee-Whye Teh. "A Fast Learning Algorithm for Deep Belief Nets". In: *Neural computation* 18 (Aug. 2006), pp. 1527–54. DOI: 10.1162/neco.2006.18.7.1527.

[16] Torsten Hoefler et al. *Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks*. 2021. arXiv: 2102.00554 [cs.LG].

[17] Kondalalith1. *The Gossip Protocol in Cloud Computing - GeeksforGeeks*. https://www.geeksforgeeks.org/the-gossip-protocol-in-cloud-computing/. May 2020.

[18] Namhoon Lee, Thalaiyasingam Ajanthan, and Philip H. S. Torr. “SNIP: Single-shot Network Pruning based on Connection Sensitivity”. In: *CoRR* abs/1810.02340 (2018). arXiv: 1810.02340. URL: http://arxiv.org/abs/1810.02340.

[19] Shiwei Liu et al. *Sparse evolutionary Deep Learning with over one million artificial neurons on commodity hardware*. 2021. arXiv: 1901.09181 [cs.NE].

[20] Shiwei Liu et al. “Topological Insights in Sparse Neural Networks”. In: *CoRR* abs/2006.14085 (2020). arXiv: 2006.14085. URL: https://arxiv.org/abs/2006.14085.

[21] Christos Louizos, Max Welling, and Diederik P. Kingma. “Learning Sparse Neural Networks through L_0 Regularization”. In: (2018). URL: https://openreview.net/forum?id=H1Y8hhg0b.

[22] Evan Lutins. *Ensemble Methods in Machine Learning: What are They and Why Use Them? — by Evan Lutins — Towards Data Science*. https://towardsdatascience.com/ensemble-methods-in-machine-learning-what-are-they-and-why-use-them-68ec3f9fef5f. Aug. 2017.

[23] Anas Al-Masri. *What Are Overfitting and Underfitting in Machine Learning? — by Anas Al-Masri — Towards Data Science*. https://towardsdatascience.com/what-are-overfitting-and-underfitting-in-machine-learning-a96b30864690. June 2019.

[24] WARREN S. MCCULLOCH and WALTER PITTS. “A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY”. In: (1943). URL: https://link.springer.com/content/pdf/10.1007/BF02478259.pdf.

[25] Decebal Constantin Mocanu et al. “Evolutionary Training of Sparse Artificial Neural Networks: A Network Science Perspective”. In: *CoRR* abs/1707.04780 (2017). arXiv: 1707.04780. URL: http://arxiv.org/abs/1707.04780.

[26] Decebal Constantin Mocanu et al. “Evolutionary Training of Sparse Artificial Neural Networks: A Network Science Perspective”. In: *CoRR* abs/1707.04780 (2017). arXiv: 1707.04780. URL: http://arxiv.org/abs/1707.04780.

[27] Decebal Constantin Mocanu et al. “Evolutionary Training of Sparse Artificial Neural Networks: A Network Science Perspective”. In: *CoRR* abs/1707.04780 (2017). arXiv: 1707.04780. URL: http://arxiv.org/abs/1707.04780.

[28] Decebal Constantin Mocanu et al. “Sparse Training Theory for Scalable and Efficient Agents”. In: (2021). arXiv: 2103.01636 [cs.AI].

[29] Hesham Mostafa and Xin Wang. “Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization”. In: (2019). arXiv: 1902.05967 [cs.LG].

[30] Michael C. Mozer and Paul Smolensky. “MozerSmolensky1989.pdf”. In: (1989).

[31] Feng Niu et al. “HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent”. In: (2011). arXiv: 1106.5730 [math.OC].

[32] F. Rosenblatt. “The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain”. In: *Psychological Review* (1958), pp. 65–386.

[33] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0. URL: https://doi.org/10.1038/323533a0.

[34] *scipy.sparse.csr_matrix — SciPy v1.5.4 Reference Guide*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html.

[35] *scipy.sparse.lil_matrix — SciPy v1.5.4 Reference Guide*. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html.

[36] Srikrishna Sridhar. *Parallel Machine Learning with Hogwild! — by Srikrishna Sridhar — Medium*. https://medium.com/@krishna_srd/parallel-machine-learning-with-hogwild-f945ad7e48a4. May 2015.

[37] Aishwarya V Srinivasan. *Stochastic Gradient Descent — Clearly Explained !! — by Aishwarya V Srinivasan — Towards Data Science*. https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31. Sept. 2019.

[38] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (June 2014), pp. 1929–1958.

[39] Kenneth Stanley and Risto Miikkulainen. "Evolving Neural Networks through Augmenting Topologies". In: *Evolutionary computation* 10 (Feb. 2002), pp. 99–127. DOI: 10.1162/106365602320169811.

[40] *Using distributed training — AI Platform (Unified) — Google Cloud*. https://cloud.google.com/ai-platform-unified/docs/training/distributed-training.

[41] *What is GPT-3? Everything your business needs to know about OpenAI's breakthrough AI language program — ZDNet*. https://www.zdnet.com/article/what-is-gpt-3-everything-business-needs-to-know-about-openais-breakthrough-ai-language-program/. (Accessed on 11/12/2020).

[42] Han Xiao, Kashif Rasul, and Roland Vollgraf. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms*. Aug. 28, 2017. arXiv: cs.LG/1708.07747 [cs.LG].

[43] J. Yu et al. "Scalpel: Customizing DNN pruning to the underlying hardware parallelism". In: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 548–560. DOI: 10.1145/3079856.3080215.

[44] Ziliang. *Paxos consensus for beginners*. https://medium.com/distributed-knowledge/paxos-consensus-for-beginners-1b8519d3360f. May 2020.