Technische Universiteit
Eindhoven
University of Technology

Department of Electrical Engineering
Electronic Systems Group

Using NAS to improve accuracy of SNNs

Master Thesis

J. Nijenhuis, BSc

Version 1.1

Eindhoven, June 2021

Contents

Contents **ii**

List of Figures **iv**

List of Tables **v**

1 Abstract **1**

2 Introduction **2**
2.1 Introduction . . . 2
2.2 Problem Statement . . . 2
2.3 Research Question . . . 3
2.4 Contributions . . . 3
2.5 Chapter Overview . . . 3

3 Spiking Neural Networks (SNNs) **4**
3.1 Introduction . . . 4
3.2 Background . . . 4
3.3 SNN vs ANN . . . 5
3.4 Training . . . 6
3.5 Framework Comparison . . . 7
3.5.1 BindsNET . . . 7
3.5.2 DECOLLE . . . 7
3.5.3 SLAYER . . . 7
3.5.4 Framework Comparison Results . . . 9
3.6 N-MNIST . . . 10
3.7 SLAYER baseline accuracy . . . 11

4 Neural Architecture Search (NAS) **12**
4.1 Introduction . . . 12
4.2 Background . . . 12
4.2.1 Search Space . . . 13
4.2.2 Search Strategy . . . 13
4.2.3 Performance Estimation . . . 13
4.3 SGAS Flexibility . . . 13
4.3.1 Fixed Layout . . . 13
4.3.2 Cells . . . 14
4.3.3 Operations . . . 14
4.3.4 Pre- and post process (set of) operation(s) . . . 14
4.3.5 Performance Estimation . . . 15

5 Combining SNN and NAS **16**
5.1 Introduction . . . 16
5.2 Search Space: Operations . . . 16
5.3 Search Space: Cell . . . 17
5.4 Search Space: Stem and classification steps . . . 18
5.5 Search Space: Number of layers . . . 18
5.6 Performance estimation . . . 18
5.7 Comparison to SLAYER baseline . . . 19

6 Results **20**
6.1 Introduction . . . 20
6.2 Data Reduction . . . 20
6.3 SGAS + SLAYER Results . . . 22
6.3.1 Large cells, few layers . . . 22
6.3.2 Small cells, many layers . . . 23
6.3.3 Small, single-input cell, many layers . . . 25
6.3.4 Forcing the SLAYER baseline model . . . 27

7 Conclusion and Future Work **29**
7.1 Conclusion . . . 29
7.2 Future Work . . . 30
7.2.1 General . . . 30
7.2.2 Changing the loss function . . . 30
7.2.3 Widening the search space . . . 30
7.2.4 More complicated datasets . . . 31

Bibliography **32**

List of Figures

3.1 The ANN Neuron Model 5
3.2 The SNN Neuron Model 5
3.3 SLAYER error credit assignment over time. (Frame taken from [1]'s YouTube explanation) 8
3.4 Probability density function of spike state change. (Taken from [1]) 8
3.5 N-MNIST saccadic movements 10
3.6 SLAYER baseline network architecture 11

4.1 The three steps making up NAS 12
4.2 SGAS fixed architecture layout 14
4.3 Two example configurations of a 3-step SGAS cell 14

6.1 N-MNIST dataset reduction experiments visualised after two epochs of training with SLAYER 21
6.2 4 steps, 2 layers, reduction cell 23
6.3 4 steps, 2 layers, accuracy over time 24
6.4 2 step, 5 layers, normal and reduction cell 25
6.5 2 step, 5 layers, accuracy over time 26
6.6 Single-input, 1 step, 4 layers, normal and reduction cell 26
6.7 Single-input, 1 step, 4 layers, accuracy over time 27
6.8 SLAYER baseline cells 27
6.9 SLAYER baseline trained by SGAS, accuracy over time 28

List of Tables

3.1 SNN vs ANN (N-)MNIST comparison 6
3.2 Framework comparison 9
3.3 Comparing the different SNN frameworks on speed. All experiments were done on N-MNIST with a simple 34x34x2 (N-MNIST input) - dense 128 - dense 10 network. 10

6.1 N-MNIST dataset reduction experiments after two epochs of training with SLAYER 21

Chapter 1

Abstract

This report is based on research attempting to find a novel, more accurate Spiking Neural Network (SNN) architecture through the use of Neural Architecture Search (NAS). There is a huge difference between an Artificial Neural Network (ANN) neuron model and an SNN neuron model. Because of this difference, and the highly temporal nature of SNNs, it is not yet known if SNNs work best in similar network architectures as ANNs, or if they need (significantly) different architectures. In this work, NAS is exploited in an attempt to automatically find novel, more accurate SNN architectures, that have not been reported before. The idea is that using NAS to find architectures, rather than conventional research, might result in networks that are less constrained by previous knowledge and experience, as that previous knowledge and experience might not be applicable to SNNs. This work presents an implementation of SNN inside a NAS framework. The framework that was taken as a basis has been partly rewritten and updated to be able to work on SNNs rather than on ANNs. The results show an unfixed discrepancy between running the same model through SLAYER and through the newly implemented SNN & NAS framework. Due to this issue, it is impossible to definitively claim that NAS is able (or not) to find novel, more accurate SNN architectures. When comparing results from inside the presented framework, it shows that NAS is able to find an architecture that achieves better accuracy than the SLAYER baseline (when implemented and trained through the presented framework), achieving 99.03% accuracy with the searched architecture, compared to 98.18% accuracy with the SLAYER baseline. The non-NAS, SLAYER-trained baseline was able to achieve up to 99.44% on N-MNIST, so the searched architecture is still not able to achieve state of the art performance, but that is due to the performance discrepancy between the two implementations. Some ideas for future research are proposed that might solve this issue. For now, it seems that NAS is able to find novel, more accurate architectures, but it is not possible to make this claim definitive due to the performance discrepancy.

Chapter 2

Introduction

2.1 Introduction

Machine Learning (ML) is being introduced more and more into our daily lives. From text completion to self-driving cars to noise reduction, ML is everywhere. However, running elaborate ML models requires significant computation resources, and often training such a network requires significantly more resources. Because of this, much research is being spent on reducing the complexity of models, making inference (and potentially training) cheaper, faster, or more power efficient. One of these efforts is the field of Spiking Neural Networks (SNNs). SNNs are biologically-inspired 'brains', that work similarly to how mammalian brains operate. This field hopes to bring the extreme power efficiency (20W, [2]) and intelligence of our biological brains to electronics.

At the same time, another field in ML is starting to receive significant attention, which is AutoML (Automatic Machine Learning). AutoML attempts to automatically determine appropriate parameters for a certain ML task that it is given by a user, where that user is not required to have any knowledge of ML. So, AutoML actually attempts to build an entire network, including hyperparameter and architecture, based solely on the dataset that it receives. One of the building blocks that make up AutoML, which is responsible for automatically determining an appropriate network architecture is Neural Architecture Search (NAS).

2.2 Problem Statement

This Master's thesis attempts to combine NAS and SNN, tasking a NAS method with the goal of finding novel, more accurate SNN architectures. The rationale behind this combination is the fact that SNNs are a very new research field, and even though many aspects of SNNs resemble ANNs (Artificial Neural Networks, the currently 'conventional' neural networks), some core concepts are significantly different (see also Chapter 3). This difference *might* mean that our current knowledge of ANN architectures is not transferable to the SNN domain. Applying NAS to (potentially) come up with novel architectures allows researchers to skip a lot of trial and error, and outsource that work to computers. This thesis attempts to be a first step in finding such novel, more accurate architectures. It will do so by developing and testing a new, NAS-based, framework that is able to search for new SNN networks. The framework will give the user much flexibility in choosing the search space. Unfortunately, simulating SNNs is currently a slow and tedious process, but with the introduction of this framework, it is hoped that it is no longer necessary to manually simulate many different architectures, but to outsource that work to NAS, and still end up with competitive architectures.

2.3 Research Question

The Research Question for this Master Thesis is:

Can NAS be used to find novel, more accurate SNN architectures?

2.4 Contributions

The contributions of this report will be:

- A framework that combines NAS and SNN in order to (hopefully) find novel, more accurate SNN architectures
- An overview of various different SNN architectures, complete with their accuracies on N-MNIST and runtimes
- An insight in possible N-MNIST data resolution options to speed up NAS training

2.5 Chapter Overview

The thesis is organised by chapter. Chapter 3 discusses current SNN challenges and available frameworks. Chapter 4 does the same, but in the context of NAS. After that, Chapter 5 is about combining the two concepts, and the corresponding challenges that arose during that combination. Chapter 6 discusses the results of the various experiments that were conducted during the research. Chapter 7 concludes the research, and discusses potentially interesting directions for future work.

Chapter 3

Spiking Neural Networks (SNNs)

3.1 Introduction

As already mentioned in the previous chapter, SNNs are a novel way of looking at ML, based on the communication and interaction within mammalian brains. The main difference between ANNs and SNNs is that SNNs feature a different neuron model than conventional ANNs, and the fact that SNN input and output have a time dimension, which ANNs do not have. An overview of these differences can be found in Section 3.3. This chapter starts with some relevant background information on SNNs, and then continues with a more deep-dive into the relevant aspects for this research.

3.2 Background

The main difference between ANNs and SNNs is the neuron model. An ANNs neuron model looks like Figure 3.1, whereas an SNN neuron model looks like Figure 3.2. As can be seen, the main differences are the lack of bias and activation function in the SNN neuron model, and the changed shape of the neuron internally. ANNs operate mostly on timeless data, there are some architectures that introduce a dependency on previous runs, such as Recurrent Neural Networks (RNNs). SNNs, on the other hand, are inherently temporal. They receive input spikes, and such input spikes alter the neuron's membrane potential. If this membrane potential reaches a certain threshold, the neuron itself sends out a spike, and resets its membrane potential. This means that SNNs always work *over time*. So, also their datasets should be (either after conversion or inherently) temporal. Input spikes also arrive over time, not necessarily at the same time. Because of this, SNNs are parallel *to the neuron level*, meaning that individual neurons receive inputs individually, process them individually, and send out spikes individually. In ANNs, outputs are usually calculated layer by layer, but in SNNs, this should be neuron by neuron.

There is no 'correct' neuron model for SNNs. The most used model is the Leaky Integrate and Fire (LIF) neuron model [3], which reduces its membrane potential while it does not receive spikes, contains a refractory period (a period after a spike where the membrane potential cannot increase), and sends a spike if the membrane potential exceeds a certain threshold. This model is used most often, as it is relatively close to nature, while still being easy to calculate.

Another option is the Hodgkin Huxley neuron model [4], which is very complicated, but very close to the neuron model in mammalian brains.

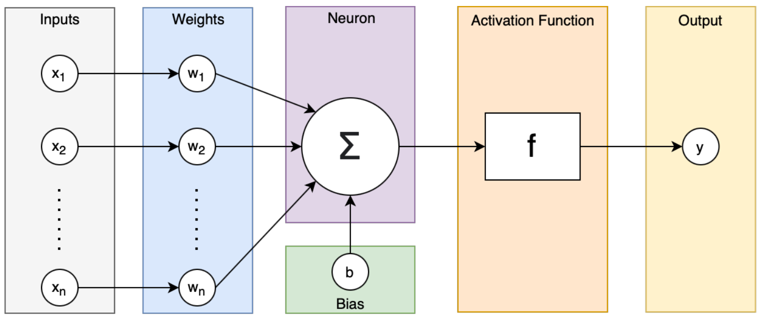


Figure 3.1: The ANN Neuron Model

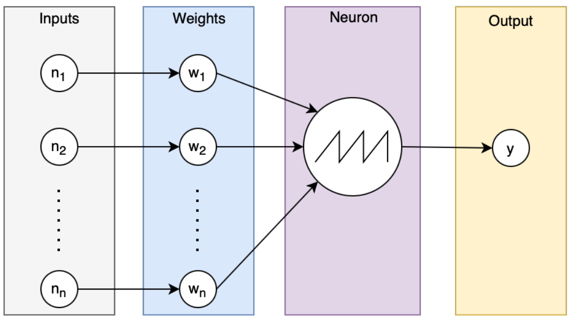


Figure 3.2: The SNN Neuron Model

3.3 SNN vs ANN

An overview of the differences between ANNs and SNNs is given below.

- SNNs are significantly more parallel than ANNs. SNNs are parallel *to the neuron level*, whereas in ANNs at the very least all neuron values on the input of a neuron have to be calculated before calculating the value of that neuron.
- SNNs are inherently temporal. Each spiking neuron generally needs to receive multiple spikes before it fires a spike of its own, creating temporal behaviour. ANNs don't have this behaviour.
- Spiking neurons are binary, either they fire or not. ANN Neurons usually have a wide range (and thus non-binary) of input and output values (e.g. 0 - 255 for 8-bit integers).
- The membrane potential in a spiking neuron is easier to calculate (just a single add per incoming spike) than the output of an ANN neuron (a multiply add operation for each input).

  *Note: This is **not** always true for simulations, as the simulations also need to take care of the membrane potential decay. Dedicated hardware automatically calculates this decay, and the decay in simulation only has to be calculated on receiving an input spike (there is no need for any intermediate value)*
- Due to the binary nature of spiking neurons, they are *not* differentiable. ANN neurons are differentiable. This means that the most used training method for ANNs, gradient descent, is not applicable to SNNs (differentiability can be approximated to use gradient descent, but that remains an approximation)
- For similar networks, SNNs (when run on dedicated hardware) are significantly more power efficient than ANNs

Note: TrueNorth simulates a million Spiking neurons in real-time while consuming 63mW. An equivalent network on a high-performance computing platform is $100 - 200\times$ *slower than real-time, while consuming* $100,000 - 300,000\times$ *more energy per synaptic event* [5]

3.4 Training

One of the currently most challenging aspects of SNNs is training them. Contrary to 'normal' Neural Networks (NNs), SNNs always have a time dimension. Where normal MNIST [6] contains 70k images (60k for training, 10k for testing) of 32x32 pixels, N-MNIST [7] contains 70k images of 34x34 pixels (slightly larger than the original set), *but also with an extra dimension of 300 time steps, and twice the number of channels (see Section 3.6).* This makes the N-MNIST dataset already over 675x bigger than the MNIST dataset it originates from. This significant increase in data results in the same significant increase in required training time. Whereas decent MNIST results (99+%) can be achieved in a matter of seconds to minutes with ANNs, N-MNIST results (e.g. through SLAYER, see below) take **hours**, if not **days** of training on SNNs.

Most frameworks (see section 3.5) focus on providing options for the training. Due to the binary nature of spikes, gradient descent cannot easily be applied, as the spike function is non-differentiable. To solve this problem, there are three go-to methods:

- Supervised Learning: Using a surrogate gradient
- Supervised Learning: Conversion from ANN to SNN
- Unsupervised Learning: Spike Time Dependent Plastiticy (STDP)

In this research, it was chosen to focus on surrogate-gradient methods. The other two options (conversion from ANN to SNN and STDP) were determined to be less suitable for combining with NAS. ANN to SNN conversion requires the ANN and SNN topologies to be the same, which defeats the purpose of searching novel SNN architectures through NAS. And the available STDP-supporting frameworks work significantly slower than surrogate-gradient frameworks (e.g. BindsNET, see below, needs 7 hours for one epoch of training with a small 200 neuron network).

A more detailed description on SLAYER (which is the chosen framework) and the way it performs the training follows below in section 3.5.4.

Table 3.1: SNN vs ANN (N-)MNIST comparison

Dataset	**# ch.**	**x, y dim.**	**# time steps**	**size diff. (relative to MNIST)**
MNIST	1	32x32	1	1x
N-MNIST	2	34x32	300	± 675x

3.5 Framework Comparison

For the selection of the SNN framework that would be used in the research, it was decided to test various frameworks on the following properties:

- Speed
- Flexibility
- Biological plausibility

The following subsections will discuss the capabilities of BindsNET [8], Decolle [9] and SLAYER [1], and Table 3.2 in Section 3.5.4 will show the overview of the properties per framework. That section will also discuss the chosen framework.

3.5.1 BindsNET

BindsNET (Biologically Inspired Neural & Dynamical Systems NETwork) is an SNN simulation and training library. It is focused on biological accuracy, which means choosing biological representability over speed. Whereas bigger frameworks such as PyTorch and Tensorflow come with predefined layers, including connections and interactions between these layers, BindsNET provides individual neuron types, layers (groups of neurons) and connections (synapses), and gives complete freedom on how these interact. This results in significant flexibility in creating new networks, which might come in handy while working with NAS, but at the same time comes at a significant cost in speed. Frameworks such as PyTorch (and SLAYER, see below) exploit a significantly advanced and mature code-base, which gives a lot of speed. This becomes clear when running BindsNET, as training a very simple 2-layer network with STDP with a total of only 200 non-input neurons requires almost 7 hours of training *per epoch*. Which is unfeasibly slow.

3.5.2 DECOLLE

DECOLLE (DEep COntinuous Local LEarning) is a framework that, similarly to BindsNET, focuses on biological accuracy. This biological accuracy does not necessarily use the Hodgkin Huxley neuron model (which is the most true-to-nature model), but it does implement local error functions, so that training takes place locally instead of the more go-to backpropagation through time (BPTT). BPTT requires knowledge of the entire network, and trains the network through all layers, and through all timesteps. While BPTT is an approach that has been shown to be able to achieve State of the art SNN results (see SLAYER below), it is not compatible with neuromorphic hardware or neurobiology. Neuromorphic hardware and neurobiology both work on the neuron level, rather than on a layer or timestep level. So, a biologically accurate way of training, rather than BPTT (which requires knowledge of the entire network), is the local learning that they perform. Although this local learning is more compatible to both neuromorphic hardware and neurobiology, it comes at a cost of speed (see Table 3.3 for runtimes). DECOLLE implements their network based on PyTorch, with LIF neurons (Section 3.2), with the LIF neurons, including their memory, implemented like an activation function within a 'normal' PyTorch network. This exploits most of the optimisations present in PyTorch, but DECOLLE is not set up as an SNN framework, but more like a proof of concept for their training method. Because of this, it only worked with their specific network layout, and did not contain the concept of different layers or operations. This made it so that it required a significant expansion to introduce the flexibility required by NAS.

3.5.3 SLAYER

SLAYER (Spike LAYer Error Reassignment) takes a different approach than the other two, in that it is a framework that is fully integrated into the existing PyTorch framework. Because of this, it is able to exploit many of the optimisations present in PyTorch, thereby achieving (compared to other

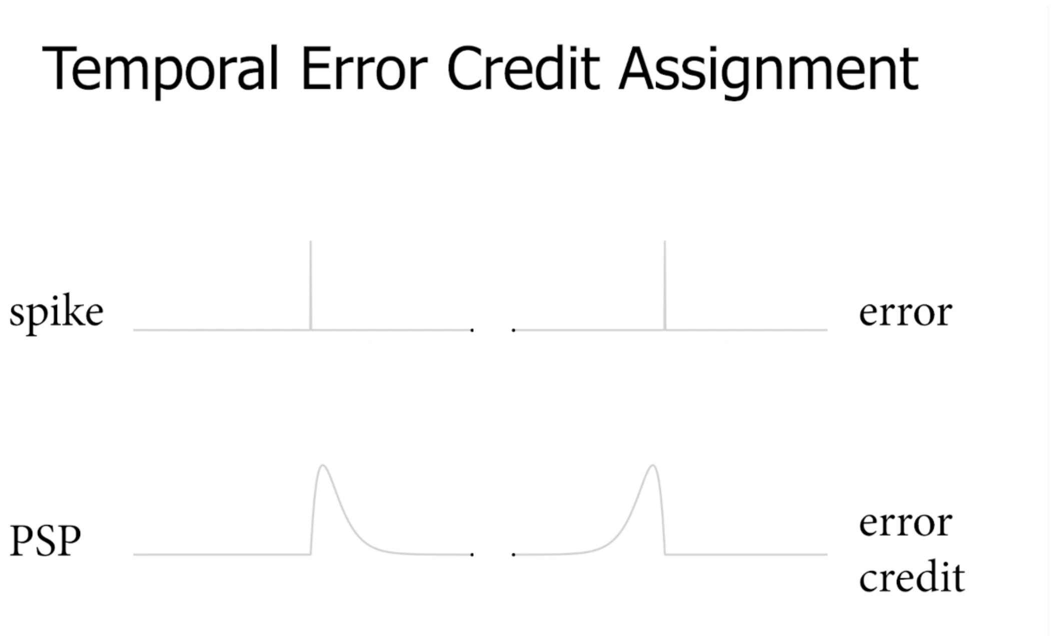


Figure 3.3: SLAYER error credit assignment over time. (Frame taken from [1]'s YouTube explanation)

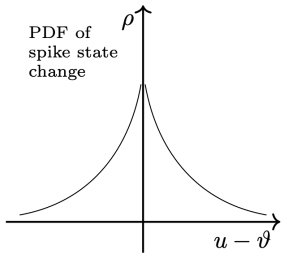


Figure 3.4: Probability density function of spike state change. (Taken from [1])

Table 3.2: Framework comparison

Framework	Speed	Flexibility	Biological Plausibility
BindsNET	-	++	++
DECOLLE	+	-	+
SLAYER	++	+	-

SNN simulators) decent speed, comparable to what can be expected for ANN simulation/training, taking into account the significantly increased size of the dataset due to the added time dimension. At the same time, this causes it to be unable to achieve full biological accuracy (see DECOLLE above), as it approximates the spike function rather than implements an actual spike function. SLAYER solves two main problems in training SNNs: (1) Temporal error credit assignment, and (2) the lack of a derivative of the spike function. As spikes are temporal, they depend not only on the state of previous neurons in the same timestep, but also on the states in previous timesteps. In order to perform BPTT, it is required to not only propagate the error back per timestep, but also across timesteps, so that the temporal influence of previous spikes is taken into account as well, and compensated for. SLAYER does this by modelling the output of a spiking neuron not as a pure spike. Instead of a pure deterministic spike, SLAYER uses the stochastic spiking neuron approximation for backpropagating errors. This relaxation can be seen in Figure 3.3, where on the top (at 'spike' and 'error') the deterministic spikes can be seen, and in the bottom (at 'PSP' (PostSynaptic Potential) and 'error credit'), the relaxed stochastic spiking neuron approximation that SLAYER uses is shown. Using this approximation, SLAYER can propagate its error not only backwards through a single timestep, but also backwards over time, which solves (1). (2) is solved by looking at a spiking neuron not as a fixed deterministic spiking/non-spiking entity, but as the probability of change in the spiking state. Depending on the current membrane potential, and the spike threshold, the probability that a neuron will fire a spike during the next timestep is determined. This probability is continuous, whereas the spike function is not. This means that the probability, contrary to the spike function, can be differentiated. A schematic example of the probability density function that SLAYER uses is shown in Figure 3.4, where u is the membrane potential, and ϑ the spiking threshold (the value of the membrane potential at which the neuron fires). SLAYER uses the spike escape rate function for this [10, 11]. It should be noted that research has shown that applying a surrogate gradient (which is exactly what this probability is), rather than an actual gradient, results in sub-optimal results [12]. For example, deciding on the weights of an ANN network through evolution has been shown to be competitive compared to gradient descent when using a surrogate gradient. The same has not been shown yet for SNNs, so it was decided to keep to surrogate gradients. It might be an interesting research topic to see if this same claim holds for SNNs. So for now, surrogate gradients are the best-known (and best performing) way to perform training in SNNs.

These two solutions are implemented in a well-documented and well-performing framework. It comes with most standard network layers (pooling, convolution, dense), and some SNN-specific extra's (such as axonal delays) and is thus the most flexible out of the three options.

3.5.4 Framework Comparison Results

So, the frameworks were judged on three properties: speed, flexibility and biological plausibility, see Table 3.2. For the sake of this research, which was to combine SNN and NAS, full biological plausibility was deemed the lowest priority out of the three selection criteria, after speed and flexibility. This was decided because both NAS and SNN are very slow compared to 'conventional' ANNs, and prioritising biological plausibility would require too much simulation time. This decision was made because the goal of the research is to see if NAS and SNN can be combined, rather than focus on biological plausibility. So, the selection criteria became speed, flexibility and biological plausibility, in order of priority. Looking at Table 3.2, this seems to mean the choice

Table 3.3: Comparing the different SNN frameworks on speed. All experiments were done on N-MNIST with a simple 34x34x2 (N-MNIST input) - dense 128 - dense 10 network.

Framework	**Batch size**	**CPU/GPU**	**# epochs**	**Time / epoch (hh:mm:ss)**
DECOLLE	72	CPU	1	00:01:42
DECOLLE	72	GPU	1	00:02:11
DECOLLE	4	GPU	1	00:45:34
BindsNET	72	CPU	1	00:16:06
BindsNET	72	GPU	1	00:14:43
BindsNET	4	GPU	1	00:28:45
SLAYER	4	GPU	100	00:00:01
SLAYER	72	GPU	100	00:00:01

should be SLAYER.

In order to make sure the speed indication is correct, each framework was suited with the same simple network and fed with N-MNIST. The results of this comparison are shown in Table 3.3, which clearly shows that SLAYER is significantly faster than the other two networks.
Because of this, SLAYER was selected in the end, allowing for the highest speed, and significant flexibility.

3.6 N-MNIST

N-MNIST [7] is a dataset consisting of recordings of the well-known MNIST dataset by an event-based camera. This means that the camera does not output frames, it outputs change events. The camera is moved in three saccades (movements) of 100ms each (for a total of 300ms). The output is a recording of 300 timesteps (1ms per timestep) containing both on- and off events on two channels. Each on event signifies a changing pixel, each off events signifies that a previously changing pixel has stopped changing (only fires once after a change, not consistently while no change is measured). The saccades are shown in Figure 3.5. N-MNIST is a dataset that is used in many SNN papers, as it is a good 'proof of concept' dataset that is quite simple, but is often able to show if novel approaches have any potential of succeeding on bigger, more representative, datasets.

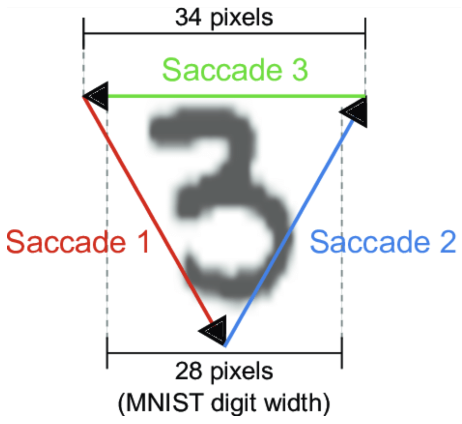


Figure 3.5: N-MNIST saccadic movements

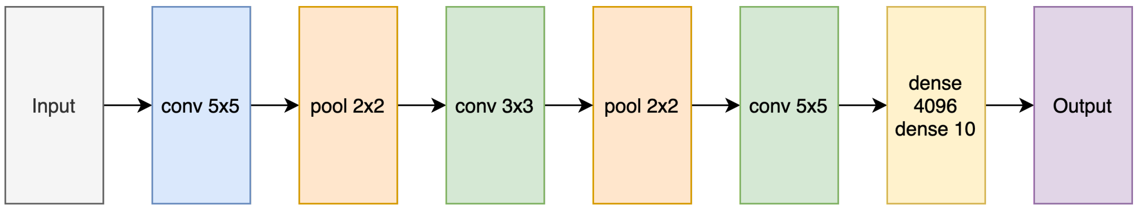


Figure 3.6: SLAYER baseline network architecture

3.7 SLAYER baseline accuracy

The goal of the Master Thesis is to see if NAS can be used to find novel, more accurate SNN architectures. To see if an NAS-proposed architecture is actually more accurate than a non-NAS architecture, it is required to have a baseline. With the choice for SLAYER, the baseline was decided to be the highest verified accuracy based on the original SLAYER paper. This resulted in the architecture as shown in Figure 3.6. With this architecture, 99.44% accuracy was achieved on N-MNIST [7].

Chapter 4

Neural Architecture Search (NAS)

4.1 Introduction

Neural Architecture Search (NAS) is about giving a computer the task to search for a neural network architecture. It is part of a bigger concept called AutoML, which is about automatically creating a full neural network (architecture, hyperparameters, etc) based on only a supplied (and classified) dataset. Background information on NAS can be found in Section 4.2. During the feasibility phase of this research three NAS frameworks were selected for further experimentation: NAO [13], SGAS [14] and ENAS [15].

After elaborate testing of the various frameworks during and shortly after the feasibility phase of this research, it was decided SGAS (Sequential Greedy Architecture Search) was most suited for this research. SGAS very nicely separated the three different aspects of NAS (Search Space, Search Strategy and Performance Estimation), allowing a complete redefinition of the Search Space to search for SNN architectures, a partial redefinition of the Performance Estimation, and no change to the Search Strategy. Also, it was the fastest framework of the 3.

4.2 Background

NAS always consists of the same three aspects, see Figure 4.1:

- Search Space
- Search Strategy
- Performance Estimation

This section will give a brief introduction on these three aspects.

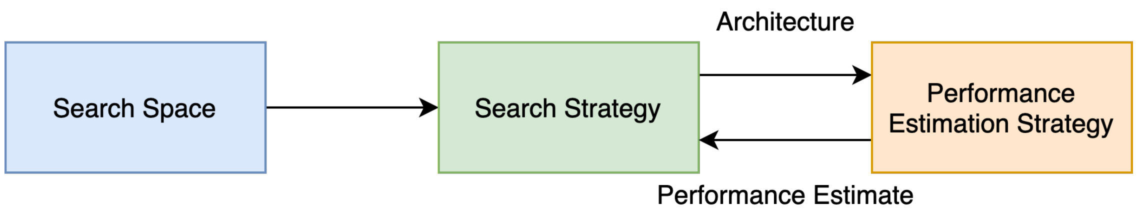


Figure 4.1: The three steps making up NAS

4.2.1 Search Space

The search space dictates which architectures can be found by the NAS method. So, the search space is all possible architectures that can be explored during the search for an appropriate architecture. Even with a few parameters, the number of possible architectures is often unfeasibly large to test and verify one by one, so NAS requires a search strategy to not have to evaluate all these architectures.

4.2.2 Search Strategy

The search strategy defines how the search space is traversed. It determines which architecture to evaluate, how to change the various parameters that are changeable in the search space, and how to pick the next architecture to evaluate. The goal of the search strategy is to find the best performing architecture inside the search space as quickly as possible.

4.2.3 Performance Estimation

Finally, to assist the search strategy in evaluating architectures, NAS uses performance estimation. The goal of performance estimation is to evaluate an architecture as quickly as possible, in order for the search strategy to be able to evaluate as many architectures as it needs, in a fraction of the time it would require to fully train and evaluate an architecture in the 'normal' way (through interference). There are generally four categories of performance estimation strategies:

- Lower fidelity estimates: e.g. reduce the resolution of the dataset, or use a subset of the data
- Learning curve extrapolation: e.g. extrapolate the learning curve after training for just a few epochs to estimate the performance after a full training run
- Weight inheritance: Assumes that similar architectures share similar weights, so use previously trained weights on slightly different networks to 'warm start' a new network
- One-shot models: Train a big network once, and then extract parts of that network, while keeping the same weights that were trained in the big network, and evaluate those parts

4.3 SGAS Flexibility

This section will mention the various hyperparameters that can be configured within SGAS. Chapter 5 discusses how these were implemented/chosen for this research.

4.3.1 Fixed Layout

The first step in NAS is to select a search space. SGAS, just like many other NAS frameworks [16, 17, 18, 19] searches for cells, rather than entire architectures. It searches for two different cells, one called 'reduction', which halves the dimensions and doubles the number of channels. The other, 'normal', keeps the dimensions and number of channels the same. These cells are then combined in a fixed structure, which is shown in Figure 4.2. It consists of:

- **Input**: The dataset input
- **Stem**: A hyperparameter 'preprocess' (set of) operation(s) for the data
- **Cells**: A sequence of normal and reduction cells, the number of cells and the type and order of the cells
- **Classification**: A hyperparameter 'postprocess' (set of) operation(s) for the data

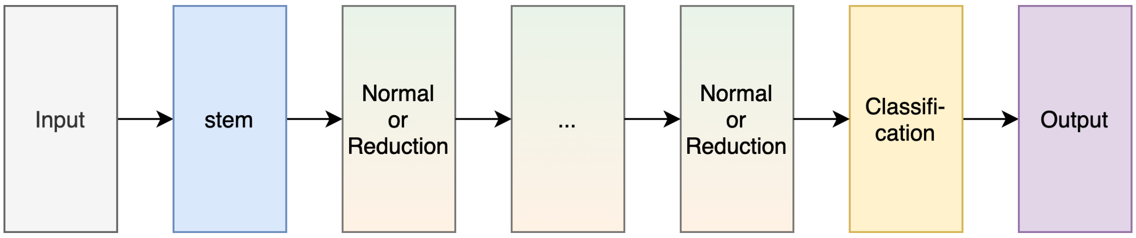
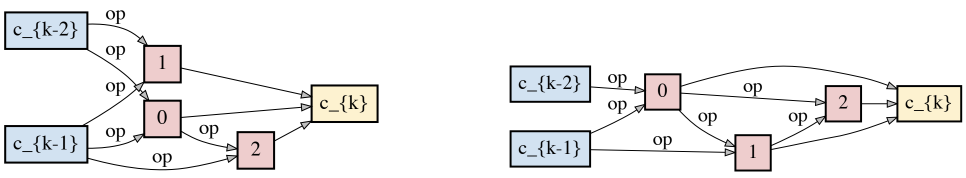


Figure 4.2: SGAS fixed architecture layout



(a) example 1

(b) example 2

Figure 4.3: Two example configurations of a 3-step SGAS cell

- **Output**: The classification output

All hyperparameter operations (stem and classification) can be anything from empty to multiple layers, as long as they operate on their inputs and output the expected number of channels and dimensions.

4.3.2 Cells

As mentioned previously, SGAS searches for two different cells, a 'reduction' and a 'normal' cell. These cells consist of a number of *steps*. Each step in a cell can best be seen as a 'buffer' for data. See Figure 4.3. Each arrow consists of an `operation` (see 4.3.3) and the resulting data of an operation is summed inside the numbered blocks in the figure. This intermediate data is then used as input for the arrows from that block to other blocks. So, each cell can be seen as a tiny network in itself, and the number of steps in a cell is the maximum number of steps data takes from the input to the output. Each step can be connected to either of the inputs, or to one of the previous steps, through an operation chosen by SGAS. The normal cell by default keeps both the dimensions (applies `stride = 1` to the `operations`) and number of channels the same. The reduction cell by default halves the dimensions (applies `stride = 2` to the `operations`), and doubles the number of channels.

4.3.3 Operations

Each arrow inside a cell (see Figure 4.3) is assigned an operation. These operations can be chosen by a user, and can be as simple or as complex as the user decides, as long as they make sure to keep the dimensions if they are used in a normal cell (stride = 1), and halve the dimensions if they are used in a reduction cell (stride = 2).

4.3.4 Pre- and post process (set of) operation(s)

The stem and classification blocks shown in Figure 4.2 are user-definable blocks that can, just as operations, be as simple (or even non-existent) or complicated as preferred by the user. Normally, the stem is used to so some preprocessing on the data to get it in some format that has been proven (or empirically shown) to work well for a specific dataset. The classification is normally a fully connected layer, that maps from the output of the last cell to the classification. Some networks consist of a sequence of fully connected layers, some only use one. The output of the classification block is a representation of the 'choice' the network has made.

4.3.5 Performance Estimation

The Performance Estimation, the final step of NAS, can be flexibly changed as well. SGAS cleanly separates searching for a network and the training of the searched network. This allows experimentation and definition of the Performance Estimator without interfering with the eventual training. By default, SGAS exploits Weight Inheritance (see Section 4.2.3) while searching for an architecture, by re-using weights learned when searching for the right operations inside a cell.

Chapter 5

Combining SNN and NAS

5.1 Introduction

Chapter 4 discussed SGAS and its flexibilities. The goal of this thesis is to combine NAS and SNN, in order to see if the combination can result in novel, more accurate SNN architectures. This chapter discusses the steps that were taken and the motivation for those steps.

5.2 Search Space: Operations

As mentioned in Section 4.3.3, SGAS tries to choose the best operations within a cell, to find the best overall architecture. The first change to combine NAS and SNN was to do a complete overhaul of the operations that made up the search space. This is also the main integration point of SLAYER, as the standard SGAS operations are all ANN operations, which needed to be replaced.

The operations that were initially decided on were the following:

- `none`: Remove an arrow from the cell (do not transfer data/spikes)
- `skip_connect`: Transfer all data one-to-one over an arrow, do not perform any operation on the data. *Note: for a reduction cell, this is not possible, as it changes the dimensions and the number of channels. In a reduction cell, the operation is changed to a `factorized-reduce`.*
- `pooling`: A 3x3 sum pooling operation
- `convolution`: A 3x3, 5x5 or 7x7 kernel convolution layer

These operations were chosen to be similar to their ANN equivalent operations in SGAS (albeit less complicated, SGAS had operations like `DilutedConvolutions`, whereas in this research, normal `Convolution` was used). Also, all SGAS operations contained `ReLU` and `BatchNormalisation` within each operation. `ReLU` was removed as SNNs do not require an activation function (see Section 3.2), and `BatchNormalisation` was removed as it is generally regarded as being ineffective for training temporal SNNs. One recent paper [20] discusses revisiting this sentiment on `BatchNormalisation`, and achieves interesting results with it, but it was only discovered well after the original decision to remove `BatchNormalisation` was already made. This might be an interesting direction for future work as well (see Section 7.2.3).

For the final simulations, the operations were separated for reduction and normal cells, so each could use their own operations, in order for the choices to make more sense (a 3x3 pooling layer with stride = 1 does not make much sense, as it does not apply any pooling, and it behaves more like a convolution layer with fixed weights). This separation was fuelled by the first couple of tests not achieving nearly the SLAYER baseline accuracy (they did not achieve accuracies over 97.71%, see also the first sections of Chapter 6), in order to include the SLAYER baseline network (Section 3.7) in the search space.

This resulted in the following choices:

Normal cell (stride = 1):

- `none`: Remove an arrow from the cell (do not transfer data/spikes)
- `skip_connect`: Transfer all data one-to-one over an arrow, do not perform any operation on the data.
- `convolution`: A 3x3, 5x5 or 7x7 kernel convolution layer

Reduction cell (stride = 2):

- `none`: Remove an arrow from the cell (do not transfer data/spikes)
- `pooling`: A 2x2 sum pooling layer
- `convolution`: A 3x3, 5x5 or 7x7 kernel convolution layer

These operations were still based on the initially chosen operations but were also validated on existing SNN architectures that reported high accuracies [1, 21, 22]. The baseline SLAYER architecture (Section 3.7) uses 3x3 and 5x5 convolution, and 2x2 pooling, which are all present in the final choices. Other SNN architectures, for example the ones that use LeNet [23] also use 5x5 convolution and 2x2 pooling, which are both available. (Actually, LeNet uses a 5x5 convolution operation with padding = 2, and one with padding = 0. In order to keep dimensions at a stride of 1, the 5x5 convolution operation always uses padding = 2. Similarly, the channels in the convolution operation adhere to the SGAS Cell 'rules' regarding normal and reduction cells). It also supports most operations that are used in [21], the only missing operations are `Skip convolutions` and `average pooling`. SGAS, as explained in Section 4.3.2 only supports operations that keep the same dimension at stride = 1, and that halve the dimension at stride = 2. This means that it is impossible to search for LeNet-like or SLAYER baseline-like fully connected layers that discard dimensions. So, those kinds of layers have to be fixed, in the `classification` (Section 4.3.4 and Figure 4.2) part of the architecture.

5.3 Search Space: Cell

After the choice of operations, the cell is the next significant choice that needs to be made (see Section 4.3.2 for a detailed explanation on the parameters present in cells). Initially, varying numbers of steps have been tested, in an attempt to find the optimal number of steps. Of course the number of steps closely relates to the number of normal and reduction cells in the network. The more steps, the fewer cells are needed to achieve the same level of complexity in the architecture, and vice-versa. Unfortunately, these architectures achieved maximum accuracies around 96% on N-MNIST, which is a significant downgrade compared to the 99.44% baseline.

As mentioned more elaborately in Section 5.7, this lower accuracy does not necessarily mean that NAS does not work. So, in an attempt to make sure that the baseline model was part of the search space, it was decided to do a complete rewrite of the cells.

The first step of the rewrite was to change the cell-rules. Previously, reduction cells doubled the number of channels, and normal cells kept them the same. However, all aforementioned SNN papers but also LeNet-like networks (e.g. [23]) that share similar normal/reduction structures, usually follow the following general rules:

- **Normal**: Keep the dimensions, double the number of channels
- **Reduction**: Halve the dimensions, keep the number of channels

This is also what is used in the SLAYER baseline. So, that was the first adjustment.

Secondly, when comparing with known SNN architectures, there was, specifically for MNIST, no architecture that used data other than from the previous 'cell' (/layer). So, in order to make it a little 'easier' for SGAS to find the baseline architecture, the cell was refactored to only take inputs from the previous cell, rather than from both the previous and the previous previous cells. Even though it seems weird to be making it easier for SGAS to find the baseline architecture, the goal with this step was to verify if SGAS was actually able to find an architecture with better performance (or the same architecture) as the SLAYER baseline. If SGAS is able to find the same architecture as the baseline, it shows that SGAS is at least able to achieve competitive results as state of the art, given the right search space. If SGAS is able to find an architecture that performs better than the baseline, it shows that SGAS is able to find better architectures than state of the art, and if it finds an architecture that performs worse, it shows that it is not (yet) on par with human design. There was also some preprocessing present in each cell to make sure the dimensions and channels of both inputs were the same, which made it so that the baseline model could never be found (this preprocessing consisted of a convolution operation from the input's number of channels to the expected number of channels). In the end, these changes resulted in single-step cells with only a single input. These cells make it so that the full SLAYER baseline model is in the Search Space, at least from the perspective of the cells.

5.4 Search Space: Stem and classification steps

Initially the stem was chosen to be a 3x3 convolution filter, to change the input dimensions from 34x34 (N-MNIST's default dimensions) to 32x32, which is nicely divisible by 2 for the reduction steps (some issues in SLAYER's implementation were found and solved regarding layers with dimensions that did not cleanly divide by 2). Later, to make sure the exact SLAYER baseline was achievable in the search space, this was changed to a 5x5 convolution filter.

The classification part started off as a `number of output channels` to `number of classes` fully connected layer, inspired by both ANN and SNN architectures that worked similarly. To include the SLAYER baseline in the search space, this was changed from the original fully connected layer mentioned before to the exact classification as used by the SLAYER baseline (see Section 3.7 and Figure 3.6).

5.5 Search Space: Number of layers

As mentioned in Section 5.3, the number of layers and the number of steps in a cell are very closely related. Initial experiments showed that the default values in SGAS (4 steps, 6 layers) resulted in networks that were simply too big to fit on a single Nvidia 2080Ti GPU. This was due to the fact that SNN networks need to take care of the time dimension (so they are bigger by a factor of the number of timesteps, which is 300 for N-MNIST). Also, during the architecture search phase, all operations are present in the network until a decision is made, making the network also several factors (slightly higher than the number of operations, due to some overhead) bigger than normal. The results of the initial experiments can be found in the next chapter. Eventually, when attempting to make sure the SLAYER baseline architecture was present in the search space, the number of layers was fixed to be 4, with the order: reduction, normal, reduction, normal cell.

5.6 Performance estimation

Because SGAS trains all operations simultaneously during architecture search, it inherently exploits weight inheritance (Section 4.2.3). So, after each decision, and after each epoch of training, it is not necessary to start from scratch again, but the network can just continue training with the previously trained weights.

Two extra performance estimation strategies are employed in the final design. Both of these strategies are in the space of 'lower fidelity estimates' (Section 4.2.3). The first one is to use a significantly reduced dataset. Instead of the full 60k train/10k test, it was decided to use the same smaller dataset that SLAYER provides with their framework. This is a 4k train/700 test set, which is enough data to get a feeling if the network trains well or not, but by far not enough to achieve the baseline accuracies. This choice already reduced the dataset by a factor ±15. The other strategy was to reduce the time dimension of the data. Elaborate experiments were performed to find which reduction performed best. The explanation of the various reductions, and corresponding accuracies, can be found in Chapter 6. Based on these results, it was decided to go for option '2c', which achieves another 3x reduction in dataset. This makes the entire speed-up to be ±45x.

5.7 Comparison to SLAYER baseline

When comparing the output of SGAS to the SLAYER baseline, it is important to note a few things. On the one hand, if the exact SLAYER baseline architecture exists in the search space, it is now (see Section 3.7) known that 99.44% accuracy is achievable when selecting the correct operations. Different choices of operations might lead to different outcomes, but as long as the outcome is below this accuracy, that means NAS has failed. On the other hand, when the architecture in Figure 3.6 is **not** in the search space, and the accuracy is lower, that does **not** necessarily mean that NAS has failed, as it might not be possible to achieve the 99.44% accuracy in that search space. So, as long as the search space does not contain the exact SLAYER baseline architecture, it is impossible to definitively say if NAS has failed or not.

Chapter 6

Results

6.1 Introduction

This chapter discusses the results of the various experiments that were performed. First, it starts with results on performing various data resolution reduction optimisations, for the performance estimation of SGAS. Then, it shows the results of various different experiments with corresponding settings on SGAS + SLAYER.

6.2 Data Reduction

In order to make sure the performance estimation of SGAS is done as quickly as possible, while still being representative for the bigger dataset, various optimisations were tested and compared to the non-reduced dataset.

There were 5 different categories of optimisations (only for training, testing is done on the full samples):

- {1, 2, 3}: Skip 1, 2, 3 time step(s), so throw away all information in those time step(s).
- {1, 2, 3}c: Add the data of 1, 2, 3 time step(s) to the current time step, thereby effectively reducing the number of time steps per sample, but throwing away as little information as possible (spikes can not be more than 1, so if a pixel fires for multiple different concatenated time steps, that information is lost)
- m{1,2,3}: N-MNIST is a recording of the MNIST dataset with an event- rather than frame-based camera (see Section 3.6). The recording is 300ms long, and consists of three different movements. These optimisations tested to see what would happen if training is done on only one of the movements.
- r: This test trained with a randomised subset of the sample. So, out of the 300ms, it would take a randomised (but consecutive) 100ms of data, and it would use that for training.
- s: The s here stands for 'small'. These were the results of the reduced dataset mentioned in the previous chapter: 4k training samples, and 700 testing samples (compared to 60k/10k respectively).

From the results in Table 6.1, it was decided to combine options 2c and s. 2c shows test accuracy comparable to no optimisation at all, while reducing each sample by a factor of 3. Reducing the number of samples for training is a very standard NAS procedure, and is done in many NAS methods [16]. The main goal while performing architecture search is to find an architecture that trains and performs well on the dataset, so choosing a small but representative dataset to verify if an architecture is able to train and infer is good enough. Even though Table 6.1 and Figure

Table 6.1: N-MNIST dataset reduction experiments after two epochs of training with SLAYER

Reduction	Train loss	Train acc	Test loss	Test acc	Data reduction %
none	0,26	97,16%	0,26	96,94%	0%
1	0,11	97,51%	0,25	97,17%	50%
1c	0,10	97,77%	0,52	95,48%	50%
2	0,07	97,13%	6,51	95,76%	67%
2c	0,05	98,28%	4,42	96,40%	67%
3	0,08	96,40%	6,38	87,91%	75%
3c	0,07	97,24%	7,01	92,75%	75%
m1	0,07	97,39%	6,91	78,39%	67%
m2	0,14	94,46%	0,39	54,54%	67%
m3	0,16	92,95%	22,7	26,98%	67%
r	0,16	93,54%	4,63	91,05%	67%
s	5,01	40,61%	2,51	68,81%	93%
s (10 epochs)	0,08	99,60%	0,37	95,42%	93%

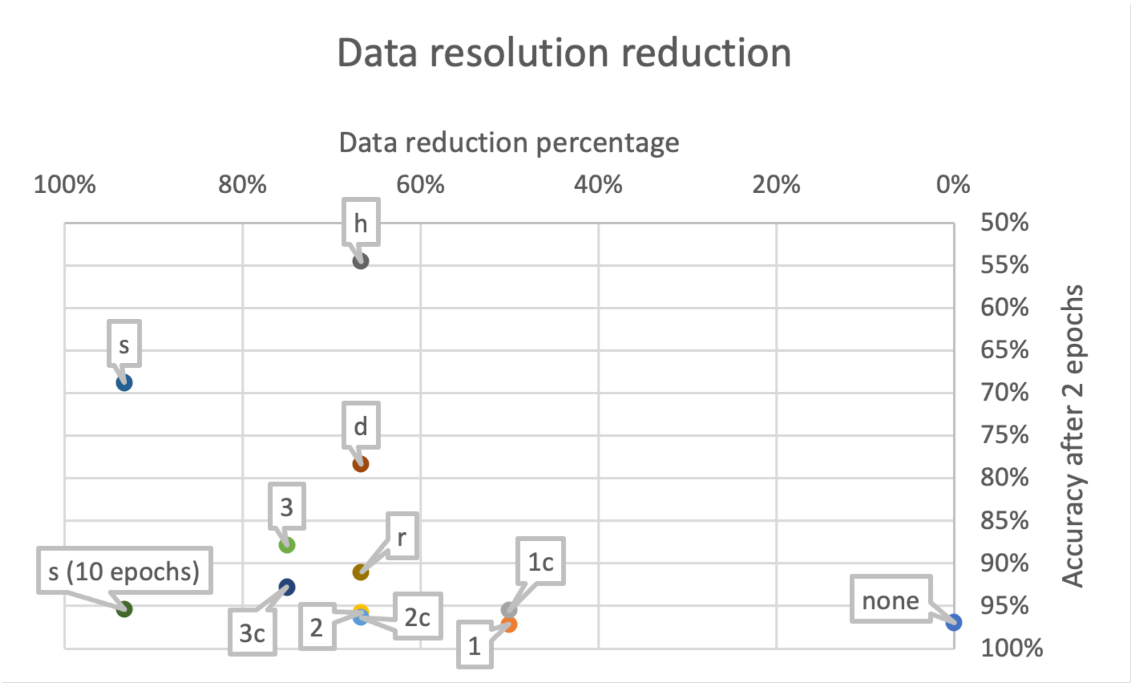


Figure 6.1: N-MNIST dataset reduction experiments visualised after two epochs of training with SLAYER

6.1 shows significantly sub-par results with s_slayer, SGAS trains for more than 2 epochs, and performance increases over time, see the result after 10 epochs of training in Table 6.1 and Figure 6.1 as well.

6.3 SGAS + SLAYER Results

The following sections will discuss various simulation results after implementing SLAYER in SGAS. The results are divided in four sections:

- Large cells (many steps), few layers
- Small cells (few steps), many layers
- Small, single-input cell, many layers (in order to include the SLAYER baseline in the search space)
- Forcing the SLAYER baseline model

At the very start of experimentation it became clear that, with SNNs added size requirements due to the time dimension, it was impossible to experiment with large cells and many layers, as even with a `batch size` of 1, it would not fit on a GPU. It was decided to only discuss these four sections, even though many more experiments were performed. As the other experiments achieved comparable results to the results discussed below, it was deemed most relevant to mention the results achieved by the extremes in the experiments. This also provides input to see what works and what does not for the future work section (Chapter 7.2).

6.3.1 Large cells, few layers

The more steps inside a cell, the more decisions SGAS has to make, and thus the more flexibility it has in finding novel, well-performing architectures. Large cells can be more intricate, which allows for more complex operations on the data from the input to the output of a cell. As datasets like N-MNIST do not require very elaborate, complicated networks, this combination of large cells (and thus much freedom for SGAS) and few layers is expected to be able to find decently-performing networks. With the choice of 4 steps in a cell (resulting in 14 total choices for operations), and a total of 5 operations to choose from (see below), this results in $5^{14} \approx 6.1 \cdot 10^9$ configurations *per cell type*. With these large cells, it was decided to reduce the number of layers to two, and to have both those layers be reduction cells, rather than also adding normal cells. This results in a maximum number of operations from input to output of 8 (number of steps · number of layers), which is bigger than most reported N-MNIST architectures ([23, 21, 7]). But, due to the inclusion of the `skip_connect`, SGAS is able to reduce the number of operations if it would decide to (by steps of 2, as two identical cells are used).

Parameters

- **Stem:** 3x3 convolution from 2 to 48 channels
- **Classification:** Sum pooling to reduce the dimensions to 1x1, followed by a dense layer to 10 outputs
- **Number of steps:** 4
- **Number of layers:** 2
- **Operations:** `sum_pool_3x3, conv_3x3, conv_5x5, none, skip_connect`
- **Normal/Reduction rule:** Reduction doubles channels, halves dimensions, normal do not care (as it is only using reduction cells)

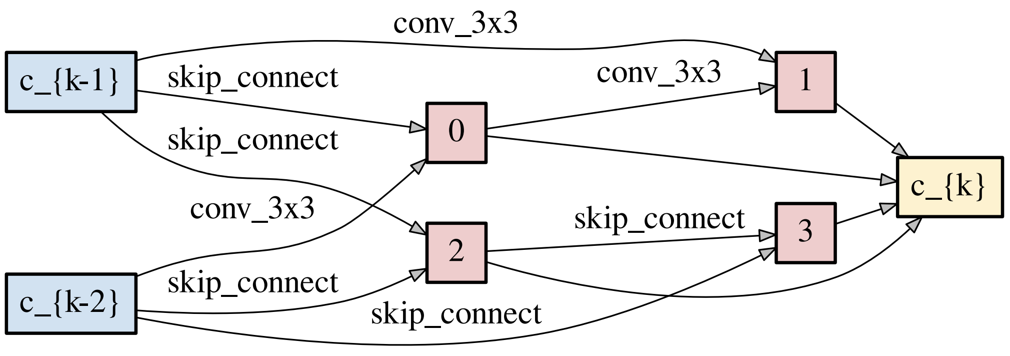


Figure 6.2: 4 steps, 2 layers, reduction cell

Results

After over 2 days and 6 hours of running, SGAS came with the first results, which can be seen in Figure 6.2. This reduction cell is the architecture found by SGAS. From all available operations, it decided to only use 3x3 convolution and skip_connects. The skip_connects for reduction cells are actually a little different than just passing of data, as the dimensions need to be halved, and the number of channels doubled. SGAS [14] proposes to replace the `Identity` operation (just pass the data, without any changes to the number of channels or the dimensions, which would not work in a reduction cell with these rules) with a `FactorizedReduce`, which is modeled by two 1x1 convolution filters with stride 2, where one filter operates on the even pixels of the input, and the other on the odd pixels. The result is concatenated in the channels dimension, thereby doubling the number of channels (due to the two filters), and halving the dimensions (due to the stride of 2 and the concatenation on the channels dimension).
Using this layout of the reduction cell and training from scratch gives the results shown in Figure 6.3. This is the result of training for 68 epochs, which took over *4 days and 15 hours*. Even though the training accuracy was still slowly increasing (testing accuracy hovered around 95% since epoch 28), the experiment was ended after that time as it was clear the 99.44% was not going to be achieved in reasonable time. Maximum training accuracy was 96.96%, and maximum testing accuracy was 95.45%.

So, unfortunately, with all this freedom, SGAS was unable to find a more accurate architecture than the SLAYER baseline.

6.3.2 Small cells, many layers

After the previous experiment, it was decided to try out the other extreme: many layers, with tiny cells. The tiniest possible cell is a 1-step cell, where there is just one operation to be chosen per input. To give SGAS a bit more freedom, it was decided to test with a 2-step cell, resulting in only $5^5 = 3125$ possible cell configurations. Actually, here, the operations were also slightly changed. It was noticed that a 3x3 pooling operation with stride 1 or 2 (which SGAS uses in their work, but the reasoning for this was not clear) was just a convolution filter with fixed weights. To make sure SGAS could actually decide on a pooling operation, it was decided to change the 3x3 pooling with a 'normal' 2x2 pooling with a stride of 2, and only include that option for reduction cells. This results in 3125 possible configurations for a reduction cell, and 1024 (4^5) for a normal cell, so $3.2 \cdot 10^6$ total configurations. The overall structure was inspired by the baseline SLAYER model, using 5 layers; normal, reduction, normal, reduction, normal, in that order.

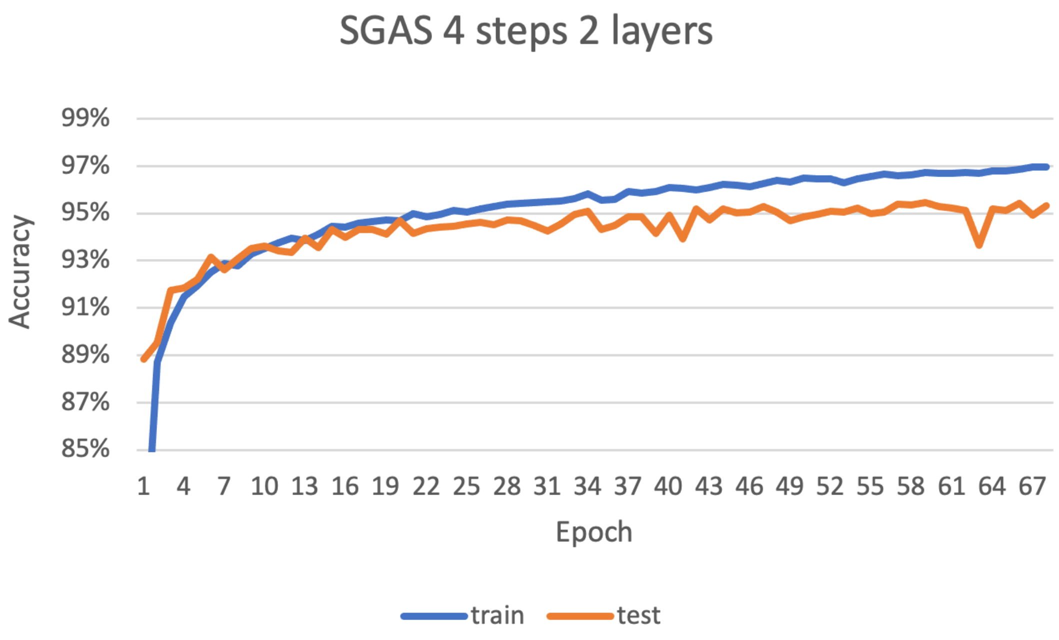


Figure 6.3: 4 steps, 2 layers, accuracy over time

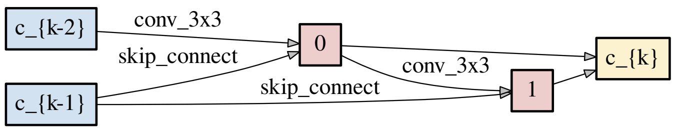
Parameters

- **Stem:** 3x3 convolution from 2 to 48 channels
- **Classification:** Sum pooling to reduce the dimensions to 1x1, followed by a dense layer to 10 outputs
- **Number of steps:** 2
- **Number of layers:** 5
- **Operations:** `sum_pool_2x2 (reduc only), conv_3x3, conv_5x5, none, skip_connect`
- **Normal/Reduction rule:** Reduction doubles channels, halves dimensions, normal keeps both number of channels and dimensions

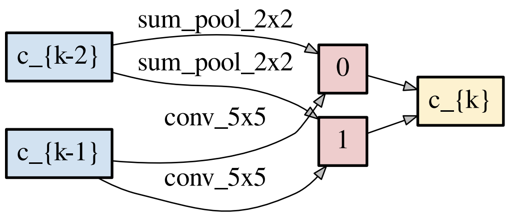
Results

After just over 1 day and 20 hours of running, SGAS came with the results, which can be seen in Figure 6.4. These cells are the cells found by SGAS. It decided to only use 3x3 convolution and skip_connect in the normal cell, but now it changed up the reduction cell, opting for the new sum pooling option and 5x5 convolution. Even though a network very similar to the SLAYER baseline was still in the search space (the exact network was not due to the cell-rules, and stem and classification operations), it decided to go for a significantly different architecture. Of course, this is entirely possible, since the goal of the research is to find novel, well-performing architectures, but it remains to be seen if the accuracy is actually similar or better than the SLAYER baseline.

Using this layout of the cells and training from scratch gives the results shown in Figure 6.5. This is the result of training for 36 epochs, which took just over *8 days and 5 hours*. It is clear that training is significantly slower than the previous experiment. This showed that the layer-overhead in SGAS is quite significant (as the time per epoch is almost 4x as high in the 2-step, 5-layer



(a) normal cell



(b) reduction cell

Figure 6.4: 2 step, 5 layers, normal and reduction cell

case than in the 4-step, 2-layer case). This is due to the fact that each cell also needs to do preprocessing to make sure the c_{k-1} and c_{k-2} inputs have the same dimensions and number of channels. This adds two more operations (one per input) to the cell, adding 50% size to a cell with these parameters, which is quite significant with 5 layers (compared to a ±15% increase in the previous situation). The testing accuracy seemed to be slightly increasing still when training was halted, but not enough to realistically achieve results similar to the SLAYER baseline. In the end, training achieved 99.73%, and testing achieved 97.71%, which is significantly better than the 4-step 2-layer situation, but still far away from the SLAYER baseline.
But, unfortunately, also with the reduced freedom, SGAS was unable to find a more accurate architecture than the SLAYER baseline.

6.3.3 Small, single-input cell, many layers

Seeing as that the test accuracy was still not nearly the same as that of the SLAYER baseline, the next step was to make sure that the SLAYER baseline model was fully present in the Search Space. This required quite some rewrite in the SGAS framework, as to achieve this, it was decided to reduce a cell from 2 to 1 input (dropping the c_{k-2}-input), which SGAS originally had no support for. Another change, which was also discussed in the previous chapter, was to change the normal/reduction cell rule (see below). Finally, as with 1 input and 1 step a skip_connect or 'none' operation does not really do anything other than completely ignore a layer or ignore any input, they were removed as well. This reduced the number of different configurations that SGAS can work with to only a very, very moderate amount. The reduction cell can choose between 4 operations, the normal cell between 3, resulting in $4 * 3 = 12$ different configurations.

Parameters

- **Stem:** 3x3 convolution from 2 to 16 channels
- **Classification:** Flatten the output of the last cell, feed it to a dense layer with 4096 neurons, then feed that to a dense layer with 10 neurons for the final classification
- **Number of steps:** 1

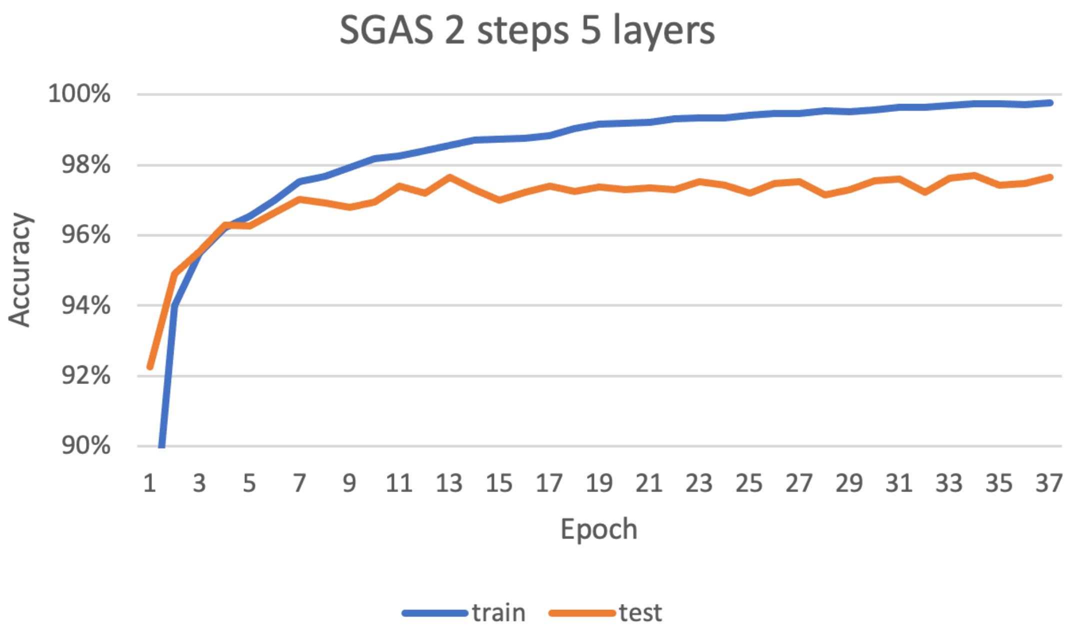
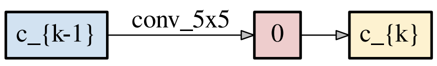
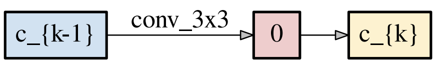


Figure 6.5: 2 step, 5 layers, accuracy over time



(a) normal cell



(b) reduction cell

Figure 6.6: Single-input, 1 step, 4 layers, normal and reduction cell

- **Number of layers:** 4
- **Operations:** `sum_pool_2x2 (reduc only), conv_3x3, conv_5x5, conv_7x7`
- **Normal/Reduction rule:** Reduc. keeps channels, halves the dimensions, normal doubles the number of channels, keeps the dimensions

Results

After just under 5 hours (clearly seeing the time benefit of the reduced search space), the cells in Figure 6.6 were found. Even though 2x2 pooling, which is the 'standard' choice for reduction-like (N-)MNIST cells in most literature, was available, SGAS decided a 3x3 convolution would be a better choice.
Using this layout of the cells and training from scratch gives the results shown in Figure 6.7. This is the result of training for 120 epochs, which took just over *2 days and 9 hours*, however, the maximum accuracy was achieved at epoch 77, after just over 1 day and 13 hours into training. Clearly, these results are the best yet, but it still performs worse than the SLAYER baseline. This seems to mean that NAS is unable to find an architecture that performs better than, or on par with, the best human selected architecture. The final training accuracy was 99.29%, and the testing accuracy was 99.03% at epoch 77. Train accuracy increased to up to 99.54%, but test accuracy stayed around 99.03%.

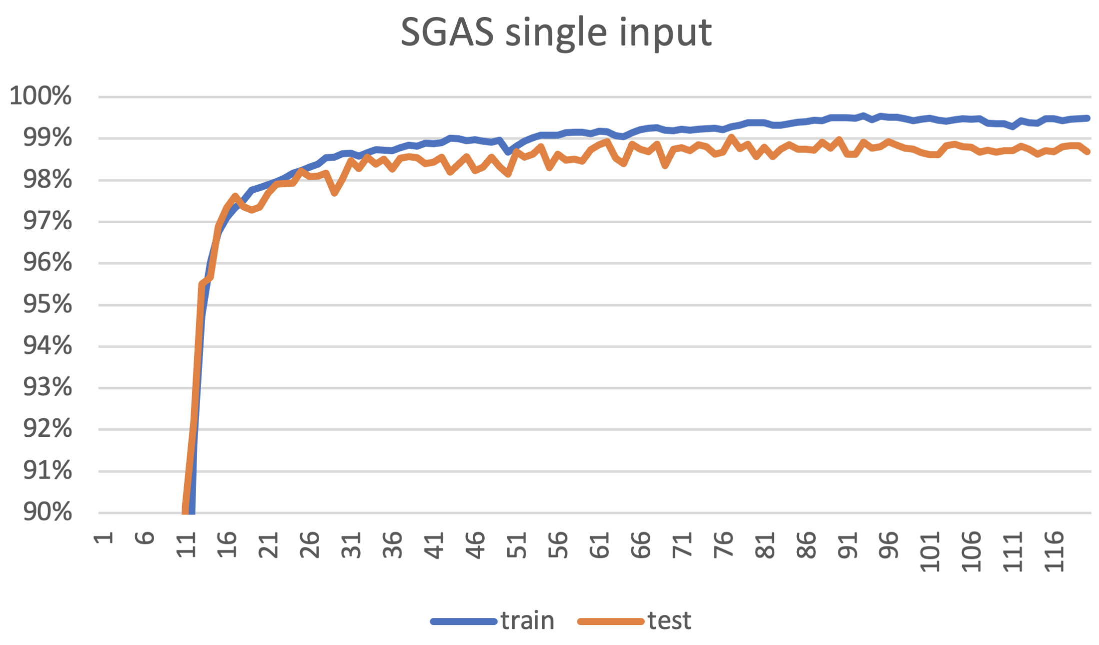


Figure 6.7: Single-input, 1 step, 4 layers, accuracy over time

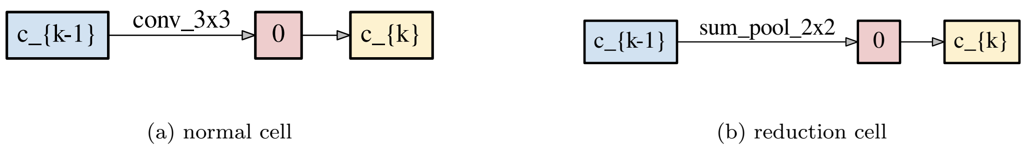


Figure 6.8: SLAYER baseline cells

Unfortunately, even with a significantly reduced search space, SGAS was unable to find either the same model as the SLAYER baseline, or a competitive option. Even though with these parameters, the accuracy was the highest of all searches, it remained short of the 99.44% that the SLAYER baseline achieved. This seems to show that SGAS is (at least for now) unable to find good SNN architectures. More on that in the next chapter.

6.3.4 Forcing the SLAYER baseline model

Finally, an experiment was performed to see what the result would be if the search phase was skipped, and SGAS was given the SLAYER baseline (cells can be seen in Figure 6.8) with the exact same parameters as the previous section. This should not be different from the non-SGAS SLAYER baseline, as training of a network is independent of SGAS. However, as can be seen in Figure 6.9, it *did* give a different result. Even though the accuracy is better than all other previous experiments, it is still far off from the original SLAYER baseline. Training went up to 99.95%, and testing achieved 98.18%, which is still far off from the 99.44% that was achieved with the same structure, but higher than all other experiments. This seems to show that something in the current SGAS implementation, or the integration between SLAYER and SGAS is off, as this accuracy should have been the same, or at least very similar. Unfortunately, this problem only became clear near the end of the research, so there was not much time to do a thorough comparison between the SGAS-trained network and the actual baseline. Initial comparison showed exactly

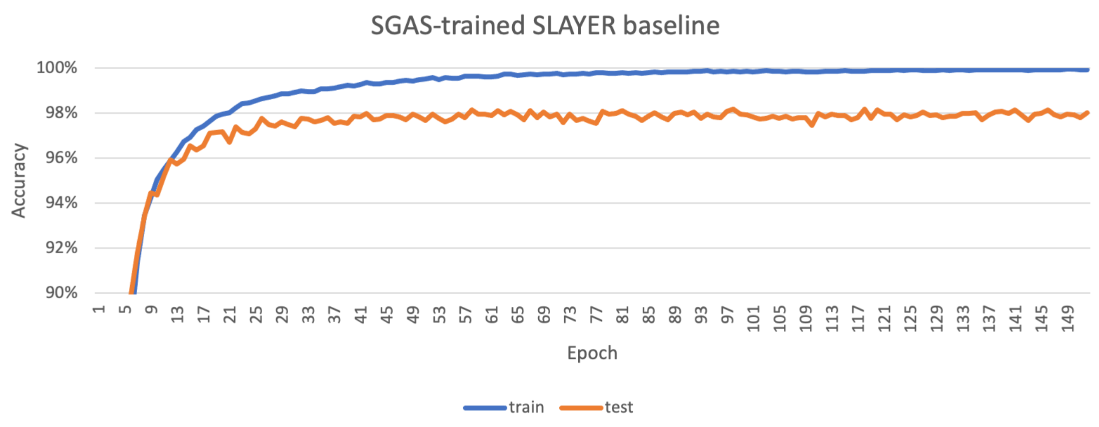


Figure 6.9: SLAYER baseline trained by SGAS, accuracy over time

the same network, with exactly the same order of operations. The main difference was how the network is structured, with SGAS encapsulating the chosen operation in a cell, and all cells in a `ModuleList`. Theoretically, this should not influence the outcome, but it might due to some unforeseen interaction within SLAYER, as SLAYER relies on a `spikeLayer()` to calculate the spike times and to keep track of the membrane potentials, and due to the structure of SGAS, each nested level comes with its own `spikeLayer()`, rather than have one level that all shares the same layer. See also the Future Work at Chapter 7.2 for more on this issue.

Interestingly, this does mean that NAS seems to have **worked**. Even though the results in Section 6.3.3 do not achieve the state of the art results achieved by the SLAYER baseline, the fact that what should be the exact SLAYER baseline architecture performs *worse* than the NAS architecture in the current implementation shows that NAS has worked. What the results in this section show is that there is an implementation difference between pure SLAYER and the new framework implemented for this research. But when comparing results from this framework, it is clear that the NAS results (99.03%) are better than the baseline results (98.18%). So, it seems that (apart from the implementation difference that results in the accuracy discrepancy between this framework and the pure SLAYER baseline), NAS has been able to find a novel, (relatively) more accurate architecture.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Even though SGAS was unfortunately not able to find an architecture that was more accurate than the SLAYER baseline, it was able to find an architecture that was better than the SLAYER baseline model implemented in the new framework. The new framework was unable to achieve the SLAYER baseline accuracy of 99.44%, which was not solved during the research. But, when comparing results from within the framework, SGAS clearly came up with a better network (99.03% test accuracy) than the SLAYER baseline model (98.18%), showing that even though it does not achieve state of the art accuracy, within its own bounds, it is able to find a novel, more accurate architecture. However, due to the discrepancy between the SGAS-trained SLAYER baseline and the true SLAYER baseline accuracies, it is not possible to definitively say that NAS is able to find truly novel, more accurate networks, as it currently is unable to achieve state of the art performance. Also, due to the fact that both NAS methods and SNN simulation require significant resources in size and time, and the combination requires even more, searching for an architecture for even a simple dataset such as N-MNIST requires hours of searching in a tiny search space and days in a bigger search space. Size-wise, most competitive NAS methods solve the problem of having to evaluate many individual architectures by trying to combine multiple architectures and evaluating them in one go. This is also exactly what SGAS does, it introduces an operation that they call `MultiOp`, which is a weighted combination of all possible operations. On the one hand, this makes it so that it evaluates all architectures in the search space at once, saving time, but on the other hand, it increases the size of a cell (during search) by a factor slightly bigger than the number of operations (due to some added overhead). At the same time, SNN introduces a time dimension to its data, which in the case of N-MNIST contains 300 time steps. Combining these properties of both NAS and SNN, the network size easily increases by several orders of magnitude. The same holds for the time requirements. SNNs require more time to evaluate due to their time dimension, and the fact that the next timestep is dependent on the previous one, so there is not much room for optimisations there. NAS, similarly, needs to evaluate many different architectures. Either it does so by actually independently evaluating the architectures, or it does so by combining multiple architectures in one network (like SGAS), but both approaches take significantly more time than just training a single predefined architecture. The size requirements limit the flexibility of the search space (as was discussed in the previous chapter, as large cells with many layers would not fit on the GPU anymore), and limits the batch size. The time requirements make it so that it takes a long time to get decent results. Even with a 45x reduced dataset with the performance estimation that was chosen, almost all experiments still took ±2 days of non-stop calculation on the TU/e's Nvidia 2080Ti with 11GB of VRAM. So, it is uncertain if NAS is *actually* able to find novel, more accurate architectures, due to the performance discrepancy between the SGAS-trained and SLAYER-trained baselines. The most time-consuming parts are the architecture search and the training, which can both be done offline. So even though both NAS and SNN, and especially the combination, have orders of magnitude higher size and time constraints than their non-NAS ANN counterparts, this is all done offline, where spending extra time or resources is often not too much of a problem if it actually achieves better results. So the conclusion of the research is that NAS *might* be **able** to find novel, more accurate SNNs, but it is not possible to make a definitive claim until the performance discrepancy between the two SLAYER baseline implementations is solved.

7.2 Future Work

7.2.1 General

Clearly, the first thing that needs to be solved is the discrepancy between the SGAS-trained SLAYER baseline and the SLAYER-trained one. There is no clear reason why there should be such a difference in performance, as the networks appear to be the same, and the learning is performed in the same way. For any future work, this should be the first focus. If that is fixed, the framework is still in place to quickly re-run previous experiments, which will definitively show if the current SGAS implementation is actually able to find novel, more accurate networks.

After fixing that problem, another interesting avenue to explore is to play around with the performance estimation. It is very plausible that the reduced dataset and/or the time-dimension reduction greatly influences SGAS's ability to find the best network, as working with a too harsh performance estimator is a known pitfall in NAS. The best way to test this is to do an SGAS run without any performance estimation, so on the full dataset, thereby getting an understanding on the differences between running on a full and on a reduced dataset. This would, however, require significant compute resources, as currently even the small cell discussed in Section 6.3.2 took 1 day and 20 hours of running, which, assuming it scales linearly, would take about 82.5 **days** of training. This would either require a framework rewrite to support multiple GPUs, or some future GPU that is able to solve this in a more reasonable timeframe.

These two points would be the most interesting to explore to verify if SGAS is actually able to find SNN networks. The next few sections discuss various ways to further exploit the possibilities introduced by being able to use NAS for SNNs.

7.2.2 Changing the loss function

NAS, just like most neural networks, attempts to minimise a loss function during training. Currently, in SGAS, the loss function is decided by the accuracy of the network it tries to create. However, there is no reason why the loss function should *only* focus on the accuracy. One of the main promises of SNNs is their premise to be able to work with significantly less energy, or to achieve the same level of 'intelligence' as normal ANNs with just a fraction of the neurons. Finding a good loss function that incorporates not only the accuracy, but also size and/or energy efficiency (number of spikes) can result in interesting new architectures that might not have an accuracy that is state of the art, but that is able to exploit the unique capabilities of SNNs regarding energy-efficiency or size.

7.2.3 Widening the search space

Changing the operations

Currently, mainly straightforward operations are in the search space. On the one hand, the available operations are complicated enough to achieve 99.44% (Section 3.7) accuracy on N-MNIST, but for more complicated datasets they might not be sufficient. Adding extra operations, such as average pooling, max pooling (if a feasible way to implement this for SNNs is found), or Batch-Normalisation (see Section 5.2) might result in a more flexible search space, and thereby novel and potentially more accurate networks.

More towards AutoML

Currently, quite a lot of the structure of the network is still a hyperparameter (number of steps, number of layers, etc), meaning that quite a few decisions are still made by humans. AutoML focuses on automatically generating and configuring an entire network, including hyperparameters, for some dataset it's given. Incorporating more AutoML-like features, such as the possibility to also search for the number of steps and number of layers, greatly increases the search space, and the bigger the search space, the higher the chance that NAS will find a novel network with high accuracy.

Evolutionary search

Another, completely different, approach to widening the search space is to look at evolutionary NAS methods. These searches usually build the network step by step in an evolutionary process. The reason this might be interesting is that most evolutionary methods have an unbounded search space. By gradually, through evolution, increasing the complexity of the network, such a method does not evaluate all possible networks, but rather builds one that is constantly expands in an attempt to increase the accuracy. While doing this, it might very well come across networks that are currently never considered either because they did not work for ANNs (but do for SNNs), or because nobody has thought of such an architecture yet. The main downside of this kind of approach is that it is usually very slow. For example, in the current SGAS search space, one of the operations was a 3x3 convolution filter. This is actually quite a complicated concept, which requires very specific connections from one neuron to the next. As most evolutionary methods do not start with such complicated concepts, but rather work with individual neurons and individual connections, it would take a lot of time, and it might not even be feasible, to ever find something like a 3x3 convolution filter.

LSMs

Finally, independent of the previous points, more and more focus in SNNs is geared towards LSMs. These Liquid State Machines are basically a recurrent neural network with spiking neurons. LSMs are based on knowledge and research on how mammalian brains seem to work. Making sure that an NAS method for SNNs that is tasked to find novel architectures also includes LSMs in its search space might result in interesting new concepts and architectures.

7.2.4 More complicated datasets

SGAS was originally built for CIFAR-10, CIFAR-100 and ImageNet-like datasets. These datasets are significantly more difficult than the N-MNIST dataset. The idea was originally to see if SGAS and SLAYER could be combined to find a highly accurate N-MNIST architecture, and then to move to more complicated datasets. However, as this highly accurate architecture was never found, the jump to more complicated datasets was never made. Even though there is no clear reason why SGAS would be less performant on 'easier' datasets, it might be interesting to explore its performance on more complicated datasets. The SLAYER baseline already achieved 99.44% accuracy, which does not leave much room for improvement. Maybe SGAS works better with datasets that are more complex, and therefore historically have a lower accuracy, so that there is more room for improvement.

Bibliography

[1] S. B. Shrestha and G. Orchard, "SLAYER: Spike layer error reassignment in time," in *Advances in Neural Information Processing Systems 31* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), pp. 1419–1428, Curran Associates, Inc., 2018. iv, iv, 7, 8, 17

[2] M. E. Raichle and D. A. Gusnard, "Appraising the brain's energy budget," *Proceedings of the National Academy of Sciences*, vol. 99, no. 16, pp. 10237–10239, 2002. 2

[3] L. Abbott, "Lapicque's introduction of the integrate-and-fire model neuron (1907)," *Brain Research Bulletin*, vol. 50, pp. 303–304, 1999. 4

[4] A. L. Hodgkin and A. F. Huxley, "A quantitative description of membrane current and its application to conduction and excitation in nerve," *The Journal of Physiology*, vol. 117, no. 4, pp. 500–544, 1952. 4

[5] P. A. Merolla, J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha, "A million spiking-neuron integrated circuit with a scalable communication network and interface," *Science*, vol. 345, no. 6197, pp. 668–673, 2014. 6

[6] Y. LeCun and C. Cortes, "MNIST handwritten digit database," *none*, 2010. 6

[7] G. Orchard, A. Jayawant, G. K. Cohen, and N. Thakor, "Converting static image datasets to spiking neuromorphic datasets using saccades," *Frontiers in Neuroscience*, vol. 9, p. 437, 2015. 6, 10, 11, 22

[8] H. Hazan, D. J. Saunders, H. Khan, D. Patel, D. T. Sanghavi, H. T. Siegelmann, and R. Kozma, "Bindsnet: A machine learning-oriented spiking neural networks library in python," *Frontiers in Neuroinformatics*, vol. 12, p. 89, 2018. 7

[9] J. Kaiser, H. Mostafa, and E. Neftci, "Synaptic plasticity dynamics for deep continuous local learning (decolle)," *Frontiers in Neuroscience*, vol. 14, p. 424, 2020. 7

[10] W. Gerstner and W. M. Kistler, *Spiking Neuron Models: Single Neurons, Populations, Plasticity.* Cambridge University Press, 2002. 9

[11] R. Jolivet, T. J., and W. Gerstner, "The spike response model: A framework to predict neuronal spike trains," in *Artificial Neural Networks and Neural Information Processing — ICANN/ICONIP 2003* (O. Kaynak, E. Alpaydin, E. Oja, and L. Xu, eds.), (Berlin, Heidelberg), pp. 846–853, Springer Berlin Heidelberg, 2003. 9

[12] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," 2019. 9

[13] R. Luo, F. Tian, T. Qin, E. Chen, and T.-Y. Liu, "Neural architecture optimization," in *Advances in Neural Information Processing Systems* (S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, eds.), vol. 31, pp. 7816–7827, Curran Associates, Inc., 2018. 12

[14] G. Li, G. Qian, I. C. Delgadillo, M. Müller, A. Thabet, and B. Ghanem, “Sgas: Sequential greedy architecture search,” 2020. 12, 23

[15] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, “Efficient neural architecture search via parameter sharing,” 2018. 12

[16] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” 2018. 13, 20

[17] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, “Progressive neural architecture search,” 2018. 13

[18] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” 2019. 13

[19] H. Liu, K. Simonyan, and Y. Yang, “Darts: Differentiable architecture search,” 2019. 13

[20] Y. Kim and P. Panda, “Revisiting batch normalization for training low-latency deep spiking neural networks from scratch,” 2020. 16

[21] C. Lee, S. S. Sarwar, P. Panda, G. Srinivasan, and K. Roy, “Enabling spike-based backpropagation for training deep neural network architectures,” *Frontiers in Neuroscience*, vol. 14, p. 119, 2020. 17, 22

[22] E. Stromatias, M. Soto, T. Serrano-Gotarredona, and B. Linares-Barranco, “An event-driven classifier for spiking neural networks fed with synthetic or dynamic vision sensor data,” *Frontiers in Neuroscience*, vol. 11, p. 350, 2017. 17

[23] B. Rueckauer and S. Liu, “Conversion of analog to spiking neural networks using sparse temporal coding,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1–5, 2018. 17, 22