Mofifoluwaso David Adegboye

NAND Networks

Computer Science Tripos Part II Dissertation

Christ’s College, 2025

Declaration of Originality

Proforma

Candidate Number:	**2347C**
Project Title:	**NAND Networks**
Examination:	**Computer Science Tripos Part II Dissertation**
Word Count:	11967
Code Line Count:	3233
Project Originator:	**Jason Brown and the candidate**
Project Supervisor:	**Jason Brown**

Original Aims

The goal of the project was to develop a new Machine Learning technique, coined "NAND networks", which has applications both in Combinational Logic Circuit Synthesis and Tiny Machine Learning. This is a research project, so the core aims were conservative, with the extension aims being extremely ambitious.

Completed Work

The project was a success. Both core criteria were met, and two out of four of the extension criteria were met. The results are very promising.

Special Difficulties

None.

Contents

List of Figures

List of Tables

1 Introduction

In this project, I invent, develop, and evaluate "NAND networks" — a method for automating combinational logic circuit synthesis with machine learning (ML). This project is motivated by applications in Tiny Machine Learning (TinyML) and combinational logic circuit synthesis. Evaluation is integrated throughout, with the results from NAND networks evaluated against the state-of-the-art (SOTA) in both domains.

1.1 Motivation

Neural Networks (NNs) are powerful, but often expensive. With a single hidden layer, sufficient width, and a non-linear activation function, they can learn any function $f : \mathbb{R}^m \to \mathbb{R}^n$. [4][23]. Deeper networks perform even better [24].

NNs tend to have improved performance as they scale. Over time, the number of parameters in SOTA ML models has grown exponentially. This is an expensive method of improving performance, in terms of money, training time, and environmental impact.

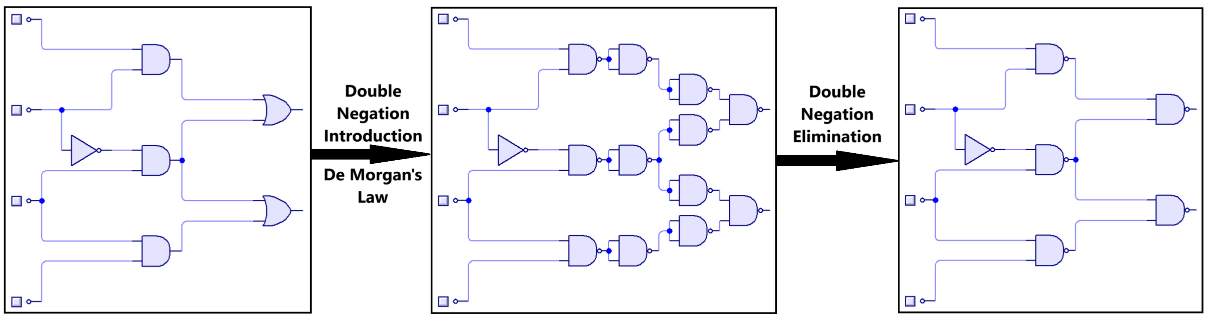


Figure 1: SoP to NAND network.

The circuits above are equivalent representations of a discrete function $f : \{0,1\}^m \to \{0,1\}^n$, where $m = 4$ and $n = 2$

With a single hidden layer, a network of NAND gates can express any discrete function $f : \{0,1\}^m \to \{0,1\}^n$. This follows by the Sum of Products (SoP) form being functionally complete. In SoP, the inputs and their complements feed into the product layer, whose outputs feed into the sum layer. Using the identities $x.y \equiv \overline{\overline{x.y}}$ and $x{+}y \equiv \overline{\bar{x}.\bar{y}}$, both layers can be expressed with NAND gates. Thus, any SoP circuit can be transformed into a shallow NAND network, as shown in Figure 1. This assumes that the input layer includes both variables and their complements (e.g., the inputs are $\{x, y, \bar{x}, \bar{y}\}$, not $\{x, y\}$). This is one of many parallels between NAND networks and classical NNs, which motivate this project.

Note that k-input NAND gates are used here, not necessarily two-input NAND gates.

1.1.1 TinyML

TinyML refers to the development and deployment of ML models on edge devices, which have limited power and memory. Thus, power efficiency and small models are prioritised. Evaluating a combinational logic circuit is significantly cheaper than inference on a classical NN, in terms of money, time, and environmental impact. If the costly matrix multiplications in classical inference can be converted to a comparable number of logic gates, it would be much cheaper — at the expense of losing the precision that matrix multiplications give, perhaps leading to worse (but cheaper) results. Thus, this project explores the application of ML techniques to learn NAND networks, which may be used in place of classical NNs, potentially benefiting TinyML applications.

1.1.2 Combinational Logic Circuit Synthesis

The search space for the optimal combinational logic circuit grows exponentially with the number of inputs, just as its truth table does. This makes the synthesis of non-trivial logic circuits a difficult task, which includes searching an exponential search space, [2]. This is one of the strengths of NNs, particularly if there is some underlying structure.

Adders, branch predictors, and other logic circuits are often hand-designed. Automating their synthesis offers two potential benefits:

1. Discovering novel designs beyond human intuition.

2. Saving human time having to design circuits.

1.2 Previous Work and Related Fields

While NAND networks as investigated in this project are novel, there have been other ML-based approaches to combinational logic circuit synthesis. Petersen et al. [18] explored a different approach to a similar problem. They had fixed connections between logic units in a set architecture, and used gradient descent to learn what the logic units should be. In this project, logic units are fixed as NAND gates, and I use gradient descent to determine where the connections between logic units should be.

Tang et al. [22] used "pseudo-neurons" as an SoP network. They used softmin functions as AND, softmax as OR. NAND networks would be equally expressive, but the addition of additional layers would enable the synthesis of deeper circuits. A shallow NAND network is functionally equivalent, as seen in Figure 1.

This project also relates to NN compression. Some common techniques used here are Binarized Neural Networks (BNNs), quantization [20], pruning [7], knowledge distillation [9], and low-rank factorisation [11]. They all reduce the size of the trained model, making them better suited to be deployed on edge devices.

BNNs are classical neural networks, where the weights and activations are ± 1. In training, the weights are real-valued, and a step function is used to binarize the network for inference. To use this in gradient descent, Courbariaux et al. use a "straight-through estimator" [3].

Quantization (and BNNs) share NAND networks' goal of smaller models, leading to faster inference, and lower memory usage. One trade-off is a slight decrease in accuracy on unseen data. The decrease in accuracy can be mitigated by quantization-aware training (QAT).

Pruning is the removal weights deemed unimportant based on their magnitude or their contribution to the loss. This increases the sparsity of the network, which leads to faster inference and less memory usage. Although, it is not a trivial task to convert the increased sparsity into accelerated inference.

Training a NAND network by gradient descent requires the NAND gates to be modelled continuously. In the final step, this is converted into a discrete circuit; this could be seen as pruning any weights that are negative, or as quantizing/binarizing the weights.

2 Preparation

2.1 Problem Description

In this project, I investigate whether gradient descent can be used to fit a Boolean function $f : \{0,1\}^m \rightarrow \{0,1\}^n$ with a NAND network. Some extensions include guiding this search to find "simpler" circuits.

The system uses a fixed number of NAND gates (arranged in a certain structure). Truth tables are used as the training data — with binary inputs, and their corresponding binary outputs. The weights specify which NAND gates are connected. Statistical techniques are used to initialise these weights, and gradient descent is used to optimise them so that the resulting NAND network computes the truth table.

The input here is the truth table. There are many hyperparameters, including but not limited to: the network architecture (which NAND gates can be connected), initialisation (which gates start connected), loss function, optimiser and learning rate. The dependent variables are the circuit that the system returns, and any readout statistics from that circuit, such as its training loss or testing accuracy.

2.1.1 Network Architecture

The network architecture uses m inputs, k hidden layers — with varying numbers of NAND gates, and n outputs. The hidden layers and outputs themselves are NAND gates. As seen in Figure 1, given that the hidden layers are wide enough, NAND networks *should* be able to express any function. To learn this function, I train the connections between the nodes. "Nodes" refer collectively to the inputs and NAND gates. In this section, I explore different regimes for connecting the nodes.

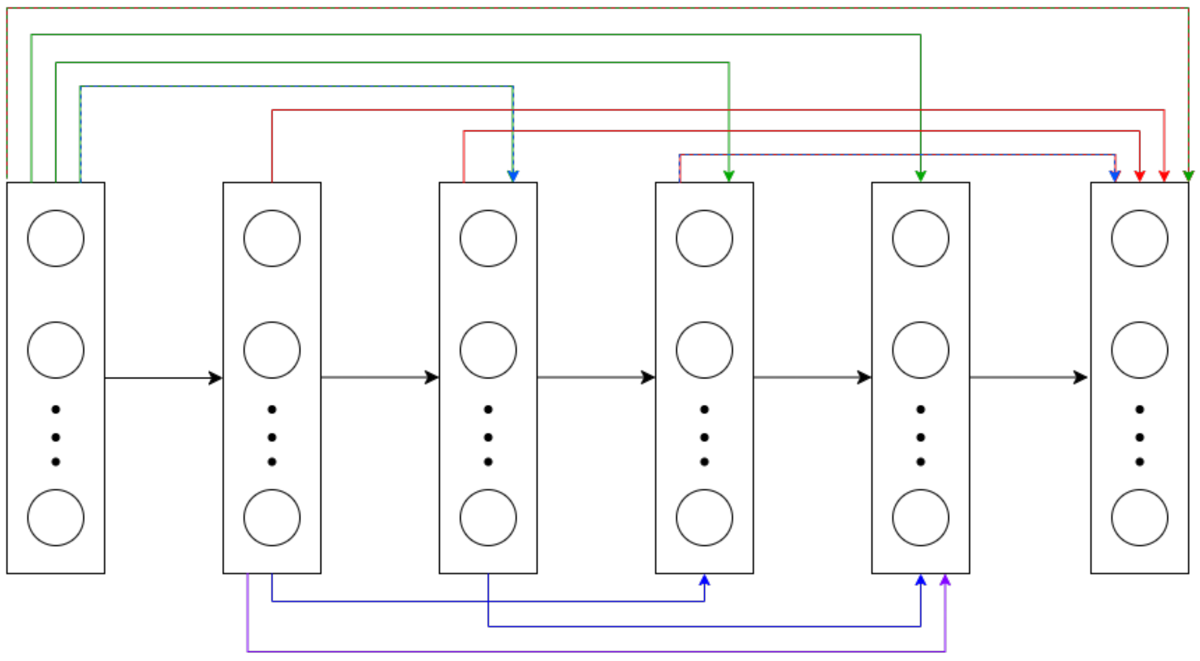


Figure 2: Network Architectures

Arrows in this figure represent dense connections between the layers

One option is to emulate NNs with dense connections. In this regime (black arrows in Figure 2), every NAND gate has x inputs, where x is the number of nodes in the previous layer. Early experiments showed this to perform poorly. This is because strictly layered NAND gates have oscillatory behaviour (if $layer_i$ is predominantly 0s, then $layer_{i+1}$ will be predominantly 1s, and vice-versa).

Another option is to have a globally connected graph (with the restriction of no lateral or backward connections). This would be using every arrow in the figure. Real combinational logic circuits do not have strict layers of logic gates. This is emulated by enabling connections between any layers, meaning that the system is more expressive, but it leads to larger weight matrices.

A third option is to allow connections to the two previous layers (black and blue). I call this twice-connected. This eliminates the issue with the dense connections, since if $layer_{i-2}$ predominantly outputs 0s, and $layer_{i-1}$ 1s, then $layer_i$ would have a some balance of 0s and 1s as inputs. It also avoids the large weight matrices of the globally connected regime.

I can also introduce residual connections (green) or connect each layer to the output (red). As more connections are allowed, the weights matrices grow in size, but the system is able to learn more expressive circuits. The residual connections and the connections to the output in particular are quite important for gradient descent [8].

2.1.2 Forward Pass

The forward pass must be differentiable to enable gradient descent. First, I make the NAND gate differentiable. This is called continuous relaxation [18], (3).

$$\text{AND}(\mathbf{x}) = \prod_i x_i \tag{1}$$

$$\text{NOT}(x) = 1 - x \tag{2}$$

$$\text{NAND}(\mathbf{x}) = \text{NOT}(\text{AND}(\mathbf{x})) = 1 - \prod_i x_i \tag{3}$$

The goal is to optimise connections between nodes, so the wires must also be modelled. Therefore, I replace x_i in (3) with $f(x_i, w_i)$, to incorporate the wires in the model. It is the w_i, the wires (or weights) that are optimised by gradient descent. The x_i represent either a discrete input ($x_i \in \{0, 1\}$) or a continuous output from a NAND gate ($x_i \in [0, 1]$).

$w_i \in (-\infty, \infty)$ represents if the NAND gate is connected to the i^{th} node. $f(x_i, w_i)$ should give the effective input the NAND gate receives from the i^{th} node to which it could be connected. To derive f, I interpret the weights as probabilities with the σ (4) function, function commonly used in ML to map from real numbers to activations between 0 and 1. Therefore, I reason about $g(x_i, p_i)$ (6).

$$\sigma : (-\infty, \infty) \to (0, 1), \quad \sigma(x) \equiv \frac{1}{1 + e^{-x}} \tag{4}$$

$$p_i := \sigma(w_i) \equiv P(\text{wire}_i \text{ is connected}) \tag{5}$$

$$g : [0, 1]^2 \to [0, 1], \quad g(x_i, p_i) := f(x_i, w_i) \tag{6}$$

There are two desired properties for g:

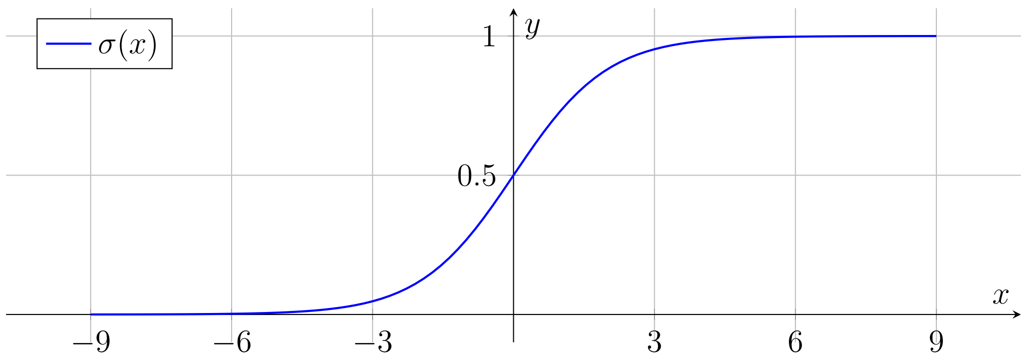


Figure 3: Sigmoid function

1. $$g(x_i, 1) = x_i \tag{7}$$

   If the wire is connected, x_i is the effective input to the NAND gate.

2. $$g(x_i, 0) = 1 \tag{8}$$

   If the wire is disconnected, 1 is the effective input to the NAND gate. This is because $1.x \equiv x$, thus for a disconnected wire, x_i should have no effect on the gate.

The domain of g is $[0,1]^2$ — a unit square. The above properties define the behaviour for the top and bottom boundaries — the discrete cases where the wire is there or not. The rest of the square can be modelled by interpolating according to the probability of the wire being there:

$$\begin{aligned} g(x_i, p_i) &= P(\text{wire}_i \text{ is connected}) g(x_i, 1) + P(\text{wire}_i \text{ is disconnected}) g(x_i, 0) \\ &= p_i x_i + (1 - p_i)1 \quad (\text{by } 5, 7 \text{ and } 8) \\ &= \sigma(w_i)x_i + \sigma(-w_i) \quad (\text{by } 5 \text{ and } \sigma(-x) \equiv 1 - \sigma(x)) \\ f(x_i, w_i) &= \sigma(w_i)x_i + \sigma(-w_i) \quad (\text{by } 6) \end{aligned} \tag{9}$$

Recalling that f is defined to replace x_i in (3), the final expression for modelling a NAND gate's continuous output is:

$$\text{NAND}(\mathbf{x}, \mathbf{w}) = 1 - \prod_i (\sigma(w_i)x_i + \sigma(-w_i)) \quad (\text{by } 3 \text{ and } 9) \tag{10}$$

Where i indexes the nodes in previous layers.

2.1.3 Weight Indexing

The NAND network is layered, meaning that it requires two indices to identify a single node. The layer index, and the gate index. Thus, four indices are needed to identify a wire, two for the source node, and two for the destination NAND gate.

I index by destination NAND gate, then by the source node. This means $[i_0, i_1, i_2, i_3]$ denotes the wire connecting node i_3 in layer i_2 to NAND gate i_1 in layer i_0. Note, $[i_0, i_1]$ is the destination, and $[i_2, i_3]$ is the source, not vice-versa. This is intentional and is a natural choice (see 3.2.1).

Since $[i_0, i_1]$ refers to the destination NAND gate, i_0 can never refer to the input layer because an input cannot be a destination (or a NAND gate). Thus, $i_0 \in \{0, 1, 2, \ldots, d\}$, where d is the number of hidden layers. This means $i_0 = 0$ refers to the first hidden layer (not the input layer), and $i_0 = d$ refers to the output layer.

Similarly, there are limitations on i_2 — depending on the scheme used (Figure 2). For example, in the twice-connected regime, if $i_0 = 0$, then $i_2 = 0$ (since only inputs connect to the first hidden layer). For subsequent hidden layers ($0 < i_0 < d$), $i_2 \in \{0, 1\}$, where $i_2 = 1$ denotes a connection from the previous hidden layer, and $i_2 = 0$ from the one before it. For the output layer ($i_0 = d$), $i_2 \in \{0, 1, 2, ..., d\}$.

2.1.4 Initialisation

The goal is an initialisation where given an equal number of 0s and 1s as inputs (which is enforced by concatenating the variables and their complements), each NAND gate outputs a 0 or 1 with equal probability. This should ensure a high-entropy starting point, which is beneficial for ML [21]. I found that this implies that each NAND gate should have an average of one incoming connection.

If a NAND gate has one effective input, then it acts as an inverter. If that input is equally likely to be a 0 or 1, then the output is equally likely to be a 0 or 1. If a NAND gate has two independent inputs, both equally likely to be 0 or 1, then there would be only a 25% chance of outputting 0. Generally, for n inputs, the probability of outputting 0 is 0.5^n. This shows the NAND gate's bias towards outputting 1. Thus, to maximise entropy, the NAND gate must have one effective input.

Therefore, if a NAND gate has n inputs, each wire should have a $\frac{1}{n}$ probability of being connected. By (5), this means that w must be generated such that $E(\sigma(w)) = \frac{1}{n}$. Using a normal distribution to sample w:

$$
\begin{aligned}
w &\sim \mathcal{N}(\mu, \sigma^2) && (11)\\
\mathrm{PDF}(w) &= \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{w-\mu}{\sigma}\right)^2} && (12)\\
\mathrm{E}(w) &= \int_{-\infty}^{\infty} w\mathrm{PDF}(w)\, dw \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} w e^{-\frac{1}{2}\left(\frac{w-\mu}{\sigma}\right)^2}\, dw \text{ (by 12)} && (13)\\
\mathrm{E}(\sigma(w)) &= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} \sigma(w) e^{-\frac{1}{2}\left(\frac{w-\mu}{\sigma}\right)^2}\, dw \\
&= \frac{1}{\sigma\sqrt{2\pi}} \int_{-\infty}^{\infty} \frac{e^{-\frac{1}{2}\left(\frac{w-\mu}{\sigma}\right)^2}}{1+e^{-w}}\, dw \text{ (by 4)} && (14)
\end{aligned}
$$

This cannot be solved analytically, but can be approximated by a sigmoid with the appropriate temperature (Figure 4). Typically, sigmoid functions have temperature parameter T, here I parametrise by $k = \frac{1}{T}$.

Using this approximation, I completed the derivation from (14), recalling that the goal is to solve for $E(\sigma(w)) = \frac{1}{n}$.

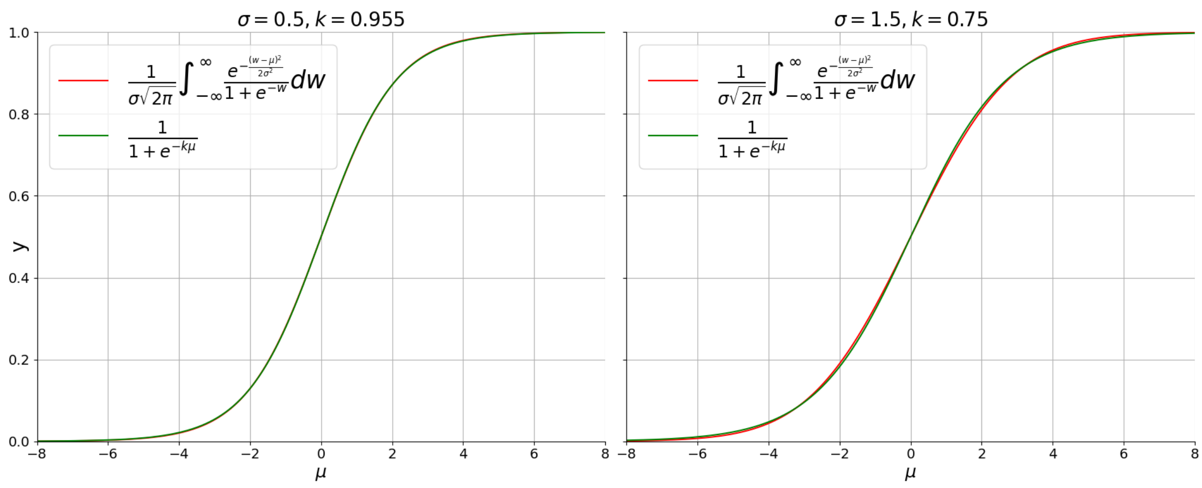


Figure 4: Approximating the integral with a sigmoid

$$
\begin{aligned}
\frac{1}{1+e^{-k\mu}} &\approx \frac{1}{n} \\
1+e^{-k\mu} &\approx n \\
e^{-k\mu} &\approx n-1 \\
-k\mu &\approx \ln(n-1) \\
\mu &\approx -\frac{\ln(n-1)}{k} \qquad (15)
\end{aligned}
$$

Given σ, $k(\sigma)$ (which is fitted graphically) and n (which is observed from the architecture), I can approximate the appropriate μ, and sample w which should maximise the starting entropy.

Notice that n would depend on the layer in which the NAND gate is located. For a NAND gate in the first hidden layer, n would be the number of inputs. If it were in the final layer, n would be the number of source nodes (inputs + hidden NAND gates). This would result in a different distribution by layer.

An alternative is to calculate $\bar{n}$, the average number of inputs to each NAND gate, then generate all the weights from the same distribution. Experimentation showed that using the same distribution globally works better.

2.2 Requirements Analysis

After reviewing the literature (see 1.2), I refined the success criteria. I had two core criteria and four extensions. Since this is a research project, the core criteria are conservative, while the extension criteria are bold. The success criteria were chosen such that the core criteria would be an interesting proof of concept, and the extension criteria could offer advances in both combinational logic circuit synthesis and TinyML. Table 1 shows a number of deliverables, along with their determined risk and priority.

The MVP is the core of the project, encompassing all components in 2.1. This deliverable includes: deriving the mathematics, implementing it in code, and learning a circuit. The deliverable was a simple learnt circuit (2-bit adder).

Main deliverables	**Priority**	**Risk**	**Difficulty**
Minimum Viable Product (MVP)	High	Low	High
Optimisations	High	Low	High
Post-training optimisations (*)	Medium	Low	Low
Intra-training optimisations (*)	Medium	Low	Medium
Learning an image classifier (*)	Medium	High	Medium
Convolutional layers (*)	Low	High	High

Table 1: A priority, risk and difficulty analysis for the main deliverables of the project. Components highlighted with (*) are extensions.

Optimisations span the project timeline; here, they refer to a huge refactoring from the MVP into cleaner, more maintainable, and faster code. This style is due to the research nature of the project. To minimise risk, it was essential to show practically that this is a viable method of learning circuits. The deliverable here was a more complex circuit (8-bit adder).

Post-training optimisations refer to a function created to remove redundancies from the learnt circuit. The deliverable for this section is data on the efficacy of the function.

Intra-training optimisations refer to penalties added to the main loss function, designed to optimise for desirable features in a circuit. These proved essential for achieving my results, and represent a key area for future work. I develop five such penalties, and the deliverable for this section is data on their efficacy.

Learning an image classifier entails repurposing the system designed to learn circuits, to learn how to classify images. The deliverable here is data on testing accuracy along with training times, and the size of the model.

Convolutional layers offer certain desirable features for image processing. I developed functions to emulate this with NAND networks, but failed to beat the results achieved without convolutional layers. This makes convolutional layers another key area for future development.

2.3 Tools used

I wrote this project in `Python`, because it is a very popular language for ML, and hence has a rich ecosystem.

I used the `JAX` module. Despite its smaller ecosystem and steeper learning curve compared to `PyTorch` and `TensorFlow`, it encourages functional programming and can create high-performance, parallelisable code. Alongside it I used `Optax`. These are modules which I have never used before, so part of the challenge was learning how to use these new modules, which brought with it a new style of programming.

I have many hyperparameters — some of which I even deprecated during the development. I store them in `yaml` files because they share syntax with `Python` dictionaries. They also enable me to use `yaml.safe_load`, to parse them into a `Python` dictionary.

I use `matplotlib` to visualise the results. This is a standard plotting library in `Python`, so it integrates well with the rest of the project.

I used `Visual Studio Code` for development, since it offers useful tools for `Python` including `IntelliSense`. This is especially helpful in conjunction with `Python`'s static type annotations. `Python` uses strong, dynamic types, and adding static type hints gives

`IntelliSense` more information to use, making programming smoother. I use `Git` for version control.

For most of the project, I used my laptop's GPU (NVIDIA 3070 Ti), but towards the end I was granted permission to use university servers.

2.4 Software Engineering Methodology

Due to the research nature of the project, I adopt the Agile methodology:

Plan → *Design* → *Develop* → *Test* → *Review*

I do not include the "Deploy" stage typically included in the agile framework. In the design stage, I use flowcharts and pseudocode to conceptualise how the maths derived in the planning stage can be translated into the `Python` code written in the develop stage. In the test stage, I validate that the developed feature works as expected, and in the review stage I evaluate the test results to inform future "plan" stages.

2.5 Testing

ML systems are typically difficult to validate as bug-free. However, since the goal for the custom circuits and the adders is 100% accuracy, it is simple to validate correctness. For the image classification tasks, I compare against SOTA results — if they are comparable, it is likely to be bug-free.

2.6 Starting Point

All code was written from scratch.

3 Implementation

3.1 Repository Overview

My project repository has the following layout:

```
NAND networks/
├── configs/ ............................ contains different configurations for tests.
├── test_results/ .... contains the outputs from tests, also used to store graphs and
│   scripts to parse results.
├── tests/ contains test scripts, which take config files from configs/ use them to call
│   main.py, and write the results to files in test_results/.
├── utils/
│   ├── adders_util.py/ .. contains useful functions for running tests to learn adders.
│   ├── custom_util.py/ .. contains useful functions for running tests to learn custom
│   │   circuits.
│   └── image_util.py/ ......... contains useful functions for running tests for image
│       classification tasks.
├── main.py/ .......... contains all core features and is where the circuits are learnt.
└── README.md ...................................... contains project description.
```

The repository is structured to separately store testing scripts, configuration files, test results, and the core program. This allows users to modify a config file, run the corresponding test, and view the results in the appropriate file. To simply learn a circuit, users can run `main.py` directly. Instructions and notes are provided in the README.

3.2 Core Features

I begin by discussing the implementation of the MVP and Optimisations in Table 1.

3.2.1 Weights Data Structure

The first major design choice is how to construct the weights data structure. This must be 4-dimensional (2.1.3). However, there are some further considerations. Recall that the four indices are:

1. (i_0) The layer index of the output NAND gate
2. (i_1) The position index of the output NAND gate
3. (i_2) The layer index of the input node (NAND gate or input)
4. (i_3) The position index of the input node

Figure 5 shows a simple globally connected NAND network. In this network, the weights array has 2 rows (for the two layers of NAND gates), meaning that $i_0 \in \{0, 1\}$.

The first issue is that these layers have different numbers of NAND gates, so the rows of the array would have different lengths, and the range of i_1 would vary. More precisely, for the first layer of NAND gates, $i_1 \in \{0, 1, 2\}$, and for the second layer, $i_1 \in \{0, 1\}$. This could be avoided by ensuring that each layer has an equal number of NAND gates. However, the architecture has a significant impact on performance (Figure 15), so being restricted to a rectangular architecture is undesirable.

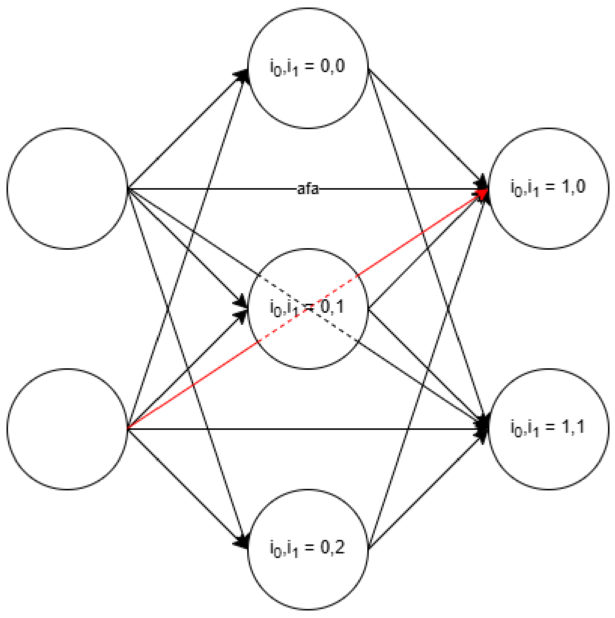


Figure 5: An example globally connected NAND network

The red arrow represents a weight where $i_0 = 1$ (2nd NAND gate output layer), $i_1 = 0$ (1st NAND gate in that layer), $i_2 = 0$ (1st node input layer), $i_3 = 1$ (2nd input in that layer).

Furthermore, as described in 2.1.3, the range of i_2 will vary. For example, in Figure 5, when $i_0 = 0$, $i_2 = 0$. But when $i_0 = 1$, $i_2 \in \{0, 1\}$.

Finally, i_3 suffers similarly to i_1. In fact, if the architecture were adjusted to ensure constant ranges for both i_1 and i_3, there would need to be an equal number of inputs and outputs. This is certainly undesirable. For example, an n-bit adder has $2n$ inputs and $n + 1$ outputs.

Thus, to represent any arbitrary architecture, there are two options:

1. Use `Python` lists. This allows elements of the lists to be different types, and therefore will be able to accommodate the varying sizes needed.
2. Use padding. Use a `jax.numpy` (`jnp`) array of the maximum size that is needed, and fill the excess with a default value, which the system ignores. This is another way to enable the varying sizes.
3. A combination of 1 & 2 (a list of `jnp` arrays, or a list of lists of `jnp` arrays, etc.)

This is more than just an internal representation choice — it alters the code that may be written. Option 2 allows for efficient parallelisation with functions such as `jax.vmap`, but requires storing excess padding weights. Option 1 restricts me to the slower `Python for` loops, but ensures that only necessary data is stored. This is a trade-off between time and space.

I found that all of the parallelism needed is intra-layer. The forward pass between layers is sequential — the output from $layer_i$ is used to compute the output for $layer_{i+1}$. This means that the weights data structure does not need to be a single `jnp` array — it is acceptable to iterate through a `Python` list of `jnp` arrays for the layers, as the layers are accessed sequentially in the forward pass.

I also found that most of the padding would come from using a single `jnp` array. The data structure must be 4-dimensional, and for `jnp` arrays, each element must have the same shape. Converting the outermost index to a `Python` list, each layer can be a 3-dimensional `jnp` array with its own shape, removing the two largest sources of padding. For example, in Figure 5, the single `jnp` array would have shape (2,3,2,3) — 36 weights for 16 potential wires. Switching to a `Python` list of 3-dimensional `jnp` arrays, the first `jnp` array would have shape (3,1,3), and the second (2,2,3). (We keep the range of i_3 as the maximum number of nodes in any source layer as this is useful for the forward pass). With this approach, only 21 weights are stored. Even in this small example, $> 40\%$ fewer weights are stored — the memory gains only improve as the size of the network grows.

In short, using a `Python` list of 3-dimensional `jnp` arrays allows both i_1 and i_2 to vary, getting most of Option 1's memory benefits, and most of Option 2's speed benefits. This is how the weights are stored going forward.

3.2.2 Initialisation

As discussed in 2.1.4, assuming a normal distribution of the weights, I derived a relationship between the parameters to maximise entropy. This result ensures that if the inputs have an equal number of 0s and 1s (which is enforced by concatenating the variables and their complements), then the outputs of subsequent layers are equally likely to be 0s and 1s — inductively ensuring that the whole network preserves this balance.

In 2.1.4, I assumed $w \sim N(\mu, \sigma^2)$, and chose μ such that $E(\sigma(w)) \approx \frac{1}{n}$. Another approach to maximising the entropy is to convert the weights from $w \in (-\infty, \infty)$ to real wires, with a step function — i.e. any weights $w > 0$ would become a wire. By (5), this corresponds to transforming a weight w_i, where $P(wire_i \text{ is connected}) > 0.5$, into a real wire. Taking this approach in the initialisation means that I now choose μ and σ such that $P(w > 0) = \frac{1}{n}$. I continue with a similar derivation to 2.1.4, replacing the condition $E(\sigma(w)) \approx \frac{1}{n}$ with $P(w > 0) = \frac{1}{n}$.

$$\begin{aligned}
w &\sim N(\mu, \sigma^2) \\
P(w > 0) &= \frac{1}{n} \\
P(w - \mu > -\mu) &= \frac{1}{n} \\
P(\frac{w-\mu}{\sigma} > -\frac{\mu}{\sigma}) &= \frac{1}{n} \\
P(Z > -\frac{\mu}{\sigma}) &= \frac{1}{n} \quad \text{where } Z \sim N(0,1) \\
P(Z < -\frac{\mu}{\sigma}) &= 1 - \frac{1}{n} \\
-\frac{\mu}{\sigma} &= \Phi^{-1}(\frac{n-1}{n}) \\
\mu &= -\sigma\Phi^{-1}(\frac{n-1}{n}) \qquad (16)
\end{aligned}$$

Yet another approach to initialisation, is to continue with the condition $E(\sigma(w)) \approx \frac{1}{n}$. By replacing the normal distribution with a beta distribution, I can obtain an exact solution, where $E(\sigma(w)) = \frac{1}{n}$. The beta distribution has the useful property that its range is $(0, 1)$, matching the sigmoid's range. Therefore, I choose $\alpha(n, \sigma)$ and $\beta(n, \sigma)$

such that the mean of the beta distribution is $1/n$, then apply an inverse sigmoid to map the samples into weights. Note that here σ is not the standard deviation of the final weights, but the standard deviation of the sigmoid of the final weights.

$$\begin{aligned}
w &\sim Beta(\alpha, \beta) \\
\frac{\alpha}{\alpha+\beta} &= \mu(= \frac{1}{n}) \qquad (17) \\
\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)} &= \sigma^2 \qquad (18) \\
\text{Let } S &= \alpha+\beta \qquad (19) \\
\frac{\alpha}{S} &= \frac{1}{n} \text{ (by 17 and 19)} \\
\alpha &= \frac{S}{n} \qquad (20) \\
\beta &= S - \alpha \text{ (by 19)} \\
\beta &= S - \frac{S}{n} \\
\beta &= \frac{S(n-1)}{n} \qquad (21) \\
\frac{(S/n)(S(n-1)/n)}{S^2(S+1)} &= \sigma^2 \text{ (by 18, 20 and 21)} \\
\frac{S^2(n-1)}{S^2(S+1)n^2} &= \sigma^2 \\
\frac{n-1}{(S+1)n^2} &= \sigma^2 \\
S &= \frac{n-1}{\sigma^2 n^2} - 1 \\
\alpha n &= \frac{n-1}{\sigma^2 n^2} - 1 \text{ (by 20)} \\
\alpha &= \frac{1}{n}(\frac{n-1}{\sigma^2 n^2} - 1) \qquad (22) \\
\beta &= \alpha(n-1) \text{ (by 19 and 20)} \qquad (23)
\end{aligned}$$

To implement this, I define a function for each distribution that takes n, σ and k, and returns a `jnp` array of independent samples from the appropriate distribution. The k parameter is only used for the first distribution discussed, but is included in the other function signatures, so they all have the same type, allowing them to be used interchangeably.

3.2.3 Feed Forward

The feed forward function follows directly from what was derived in (10). Starting with the inputs, and going from left to right, I compute the output of each layer. This is the most critical code in the whole repository — any issues here will be felt in both training and testing. As such, this code is particularly notable for the "Optimisations" mentioned in Table 1. It uses a lot of `JAX` functions to improve performance, and went through a lot of iterative development to reach the stage it is at now. As a result, it went from taking hours to learn the initial proof of concept circuits (2-bit adders), to learning 8-bit adders in minutes.

One thing that makes this code so critical is how often it will be called and which contexts it will be called in. The first step in the forward pass is to define a function to convert the weights data structure from containing floats $w \in (-\infty, \infty)$ to containing the probabilities of the wires being there. Recall that g (6), uses the probabilities p_i, not w_i directly. With this perspective, I can easily switch out the function $\sigma : (-\infty, \infty) \to (0, 1)$ for a step function $H : (-\infty, \infty) \to [0, 1]$, and the rest of the code works smoothly.

In Figure 27 (Appendix), four functions are defined that may be used in the forward pass. The first is the sigmoid, which according to (5) converts w_i to p_i. I also define a version with a temperature parameter, although this temperature parameter is defined globally, so its only input is still `x`.

The third is the step function. We no longer ask "How likely is the wire to be there?", but "Is the wire there?". More precisely, according to the model, the question is "Is there at least a 50% probability that the wire is there?". This is used when calculating the testing accuracy. In order for this system to be used for combinational logic circuit synthesis, I must be able to convert these continuous weights into discrete readouts, and I use the step function for that.

$$X \sim \text{Bernoulli}(p_i) \quad\iff\quad P(X = x) = \begin{cases} p_i & \text{if } x = 1, \\ 1 - p_i & \text{if } x = 0, \end{cases}$$

Another option is to use a Bernoulli distribution. This follows directly from (5), placing the wires based on the probability assigned to them. With the step function, the system always takes the more probable option — if the network gives a wire a 51% probability, it will always be placed. With the Bernoulli distribution, it is placed 51% of the time. Defining a seeded Bernoulli distribution with `JAX` is not simple, due to how it works under the hood, and so this function in particular was quite difficult to get to.

The `and_helper` function in Figure 28 (Appendix), computes $\prod g(x_i, p_i)$ over the $\vec{x}$ and $\vec{w}$ passed into it. It uses the `weight_activation_dict` to calculate p_i, using a sigmoid function, step function, or Bernoulli distribution. This function has a decorator on line 1, "`@partial(jax.jit, static_argnames="weight_activation")`". This decorator "Just-In-Time" (`JIT`) compiles the function, transforming it into efficient code, optimised for the processing unit on which the program is run (GPU, TPU, etc.). This comes at a one-time cost. When the function is first called, extra time is spent `JIT` compiling it. If the function is called enough times, and the speed-up from it being `JIT` compiled is large enough, then this can make the code orders of magnitude faster.

With the `static_argnames` argument, each time this function is called with a new value for `weight_activation`, it is `JIT` compiled again. The one-time cost of `JIT` compilation would lead to worse performance than no `JIT` compilation for a parameter like `w`, since this is being constantly changed. But with the `weight_activation` parameter, there are only four different options, and so there is still a notable speed-up. Using partial applications to `JIT` compile in this way is equivalent to defining four separate functions where `weight_activation` $\in \{$"cont", "disc", "rand", "temp"$\}$ and `JIT` compiling them separately. But with this method, code is not repeated.

The `forward` function (Figure 29) (Appendix) calculates the output for a single NAND gate. It does this by computing the "AND" of each layer from which it receives an input, in parallel. It then returns one minus the product of the "AND"s to get the "NAND". I use `jax.vmap` here to vectorise the code. This takes each row of the `xs` matrix going into the NAND gate, and each row of the `ws` matrix connecting them to the NAND gate, and passes them in parallel into the `and_helper` function.

The code given is how the `feed_forward` function would work for a globally connected regime. For this, I start with the input, and pad it to be a `jnp` array of length `i_3`. `i_3` is defined based on the architecture, to be the maximum width of any layers. Then for each layer, its output is calculated in parallel, by calling `jax.vmap` on `forward`. This takes all of the `xs` learnt from the previous layers so far, and the relevant weights. These outputs are then stacked onto the `xs` array, ready to be used to calculate the next layer's output. For other regimes, I change the indices of `xs` that I pass in, e.g. for a classical dense network, I would just pass in `xs[-1]`, the previous layer.

3.2.4 Padding

For both `weights` (3.2.1) and `xs` (Figure 30, Appendix), I use padding to ensure that `JAX` speed-ups can be used. In the case of the weights, the value I pad with is $-\infty$ (-`jnp.inf`). This ensures that any padding weights act as disconnected wires, not affecting the computation. In the case of the xs, the value I pad with is 1. This ensures that any padding x values would not affect the computation. If $x_i = 1$, the output is not affected, similarly for if $w_i = -\infty$, (10). Note for any function $f : (-\infty, \infty) \to [0, 1]$ to be a sensible "`weight_activation`", it must satisfy $f(-\infty) \coloneqq 0$ and $f(\infty) \coloneqq 1$, and this is true for the four functions defined in Figure 27. This means that even when not using the sigmoid, any padding weights would not affect the computation.

3.2.5 Loss

At its core, the loss function takes some input, an expected output, does the `feed_forward` pass, and calculates the binary cross-entropy (BCE) between the prediction and the expectation. BCE is a natural choice for a two-choice classification problem, which is the case here. Assuming that the model's output is the probability it thinks the output should be 1 (which follows naturally from 5), then it can be shown that minimising BCE maximises the probability of observing the training data, given the weights. This also requires the simplifying assumption that each instance of training data is independent.

$$p(s|w) = \prod_{i=1}^{n} [h_w(x_i)]^{y_i} [1 - h_w(x_i)]^{1-y_i} p(x_i)$$

Figure 6: The probability of observing the training data given the weights. s represents the training data. w represents the weights. h_w represents the feed forward function with the appropriate weights. x_i is an individual example of training data. y_i is its appropriate label

$$\begin{aligned}
w_{opt} &= \underset{w}{\operatorname{argmax}}[\log(p(s|w))] \\
&= \underset{w}{\operatorname{argmax}}[\sum_{i=1}^{n} \log([h_w(x_i)]^{y_i}) + \sum_{i=1}^{n} \log([1 - h_w(x_i)]^{1-y_i}) + \sum_{i=1}^{n} \log(p(x_i))] \\
&= \underset{w}{\operatorname{argmax}}[\sum_{i=1}^{n} y_i \log(h_w(x_i)) + \sum_{i=1}^{n}(1 - y_i) \log(1 - h_w(x_i))] \\
&= \underset{w}{\operatorname{argmin}}[-\sum_{i=1}^{n} y_i \log(h_w(x_i)) + (1 - y_i) \log(1 - h_w(x_i))] \qquad (24)
\end{aligned}$$

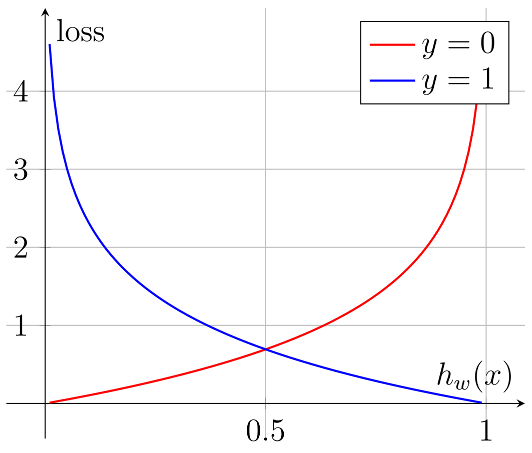


Figure 7: BCE

Figure 7 shows a graph of BCE for a single example of x. In that figure, x values are clipped to the range [0.01, 0.99], so the y values are limited. Allowing them to be in the full range of [0, 1] would put the y values in the range [0, ∞), i.e., the loss would be infinite if the network is wrong with 100% confidence. This would cause the gradient to be -∞, resulting in `jnp.nan` propagating through the network. To prevent this, just like I did for the graph, the network's prediction is clipped to be in the range [ϵ, 1-ϵ]. This ϵ should be as small as possible to match the true BCE function as closely as possible, but not so small that I end up with `jnp.inf` in the loss, resulting in `jnp.nan` in the gradients, which in itself would result in `jnp.nan` in the weights after updating by the gradients.

3.2.6 Gradient Descent

Algorithm 1 (Appendix) shows a simple version of gradient descent. This is the process of updating the current weights based on their gradient vector. Algorithm 2 (Appendix) is an extract from the original Adam paper [12], and the version of gradient descent used here. It incorporates the concepts of momentum (first moment) and adaptive learning rate scaling (via the second moment). These give it great all-purpose performance, and make it a good choice for quick research and development.

I may also include some form of batching. To do this, on each pass I shuffle the data, then do one step of gradient descent on each batch of data. Table 2 shows some benefits and drawbacks of batching, compared to no batching (Online learning, batch size=1) and Full Batch (batch size=n, where n is the total number of data points).

The two concerns for this project are the effect of batching on training time and generalisation. Batching implicitly adds a regularisation term to the loss function [15] [19]. With the right batch size, the regularisation effect is optimised for the best generalisation. This is achieved by creating the best balance of minimising the implicitly added regularisation term, and minimising the true loss. The training time is optimised by choosing the largest batch size that the system can handle.

	Online Learning	**Batching**	**Full Batch**
Gradient noise	Highest	Moderate	Lowest
Implicit Regularisation	Highest	Moderate	Lowest
Convergence Stability	Lowest	Moderate	Highest
Memory usage	Lowest	Moderate	Highest
Hardware efficiency	Lowest	Good	Best (*)

Table 2: Comparison of batch regimes in gradient-based learning.

(*) Up to hardware limitations.

I use `jax.grad` to create the gradient vector. This transforms a function into a gradient function. It returns a function which has the same parameters as its input function, and gives the gradient vector of the function, with respect to the specified parameters. This is used to calculate the gradient vector of the loss with respect to the weights. This would gives $\nabla_w \mathcal{L}$, which is needed to run `adam`. I use `optax` to create, store and update the first and second moment terms that `adam` uses.

`jax.grad` uses auto-grad, which causes `JAX` to trace the loss function, and keep them in a computational graph. It then uses the chain rule to walk backwards through the graph and compute the gradients.

3.3 General Extension Features

I now discuss some features I added, which improve performance (better quality circuits and/or faster training time), for both learning adders and image recognition.

3.3.1 Post-training Optimisations

The BCE loss is only incentivised to increase the accuracy. In this section, I discuss what can be done after training to optimise the circuits. The aim is to investigate ML techniques in circuit synthesis, not how to optimise combinational circuits only using NAND gates. For this reason, this code was not prioritised, so there are certain optimisations that I do not do. Also, since this code only runs once, and I only use it for smaller circuits, I do not apply as much stepwise refinement here as elsewhere.

There are three optimisations I make and one which I do not make, but could be used to produce even better circuits:

1. Remove unused gates (I do this)

2. Remove duplicate connections (I do this)

3. Remove unnecessary double negation (I do this)

4. Absorption (I do not do this)

To do optimisations 1 and 2, I define a function which converts the weights data structure into a list of strings — one string for each output. I start by converting the inputs into strings. Then working from left to right (from inputs to outputs), I construct strings for the hidden NAND gates. This is where optimisation 2 happens. For an individual NAND gate, I first identify the nodes going into it and their strings. These go into a set to remove duplicates. I then canonicalise them. Without this step I could end up with terms like $\overline{A.B}$ and $\overline{B.A}$, not seen as duplicate. I then convert this ordered list into a string, and this is where step 3 happens, being careful to do optimisations like $\overline{\overline{A}} \rightarrow A$ and $\overline{\overline{\overline{A.B}}} \rightarrow \overline{A.B}$, but not $\overline{\overline{A.B}} \rightarrow A.B$.

The absorption law has two commonly seen forms: $A.(A+B) \iff A$ and $A + A.B \iff A$. The first works because $A \Rightarrow A + B$ and the second because $A.B \Rightarrow A$. From the first, if we have $\overline{X.Y}$ where $X \Rightarrow Y$, this can be reduced to $\overline{X}$. This can be checked with the truth tables of X and Y.

3.3.2 Fan-in Penalties

The fan-in of a gate is the number of inputs it has. When building circuits, gates with lower fan-ins are preferred for a multitude of reasons. As a result, I take some effort in this project to produce circuits not just with low propagation delay (a low number of layers), but also with low fan-ins. This is partially done by the post-training optimisations discussed above, but it is mostly achieved with the fan-in penalties described in this section.

$$\texttt{ReLU}(x) := \max(0, x) \tag{25}$$

Before discussing how to implement the fan-in penalties, I first define `ReLU` (25). `ReLU` is a common activation function used in ML. This is used in all but one of the penalty functions here. The useful property is that it maps negative numbers to 0. Thus, if I first apply transformations that make any terms that I do not want to contribute negative, I can then apply the `ReLU` to mask them out.

Figure 31 (Appendix) shows the max fan-in penalty. This calculates a continuous estimate of the fan-in, by summing the probabilities of the wires going into each NAND gate. To see this, note that this approximation is exact in the case where the probabilities are 0 or 1. I then subtract the parameter "`max_fan_in`" element-wise and pass the result through a `ReLU` function. The result is an array which is 0 where the fan-in estimate is less than the `max_fan_in` parameter, and the difference between the fan-in estimate and the `max_fan_in` parameter otherwise. The final line of the function takes a weighted sum of all of the penalties, and it is used to more aggressively penalise the NAND gates with the highest fan-in estimates.

I similarly define a `mean_fan_in_penalty` function, where I directly sum the fan-in estimates, and divide by the number of NAND gates in the network. I then subtract some `mean_fan_in` parameter, and return the `ReLU` of the result. This has the effect of applying a penalty to restrict the fan-in of all of the NAND gates, only if the current continuous estimate for the mean fan-in is above the specified threshold.

It is worth noting the `temperature` parameter in Figure 31. This is used to scale the weights before passing them into the sigmoid function. If `temperature` < 1, this has the effect of pushing the probabilities closer to 0 or 1. As a result, the continuous approximations of fan-ins for each NAND gate are more representative of what the true fan-in would be when the network is discretised.

To implement all the penalties, I multiply them by a coefficient and add the result to the BCE loss. These coefficients are important hyperparameters, which give a simple way to directly affect computed gradients. Higher coefficients for this mean that the network may aggressively limit fan-in, potentially at the expense of the BCE loss. Too low and I get the opposite affect, so hyperparameter fine-tuning is key here.

3.3.3 Continuous Penalty

The system often learnt circuits where the BCE loss was low, but this did not always lead to high testing accuracy. This is a common issue in ML, and can be caused by over-fitting, or a host of domain-specific issues. For example, the network could be learning to be more confident with its correct answers without changing the predictions for the incorrect samples. In this project, when the loss was low but the accuracy was still low, a contributing factor was the discretisation step. The loss is calculated with $\sigma(w)$, but the accuracy is calculated with `step`(w). Where $|w|$ is low, there could be a difference of up to 0.5 in these values, leading to the loss not being as negatively correlated with the accuracy as desired.

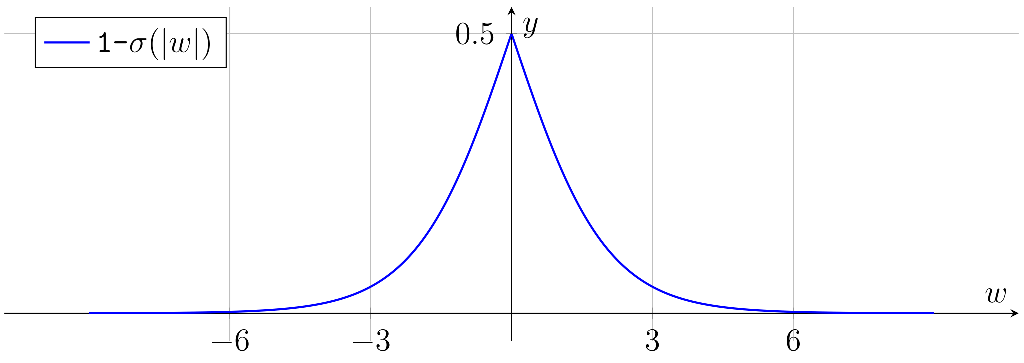


Figure 8: Continuous penalty function

To counteract this, I add a penalty which penalises weights with a low magnitude (Figure 8). This would make the network more certain about which wires to place, making $\sigma(w)$ closer to `step`(w), and also means that a low BCE loss is now more likely to imply a high testing accuracy.

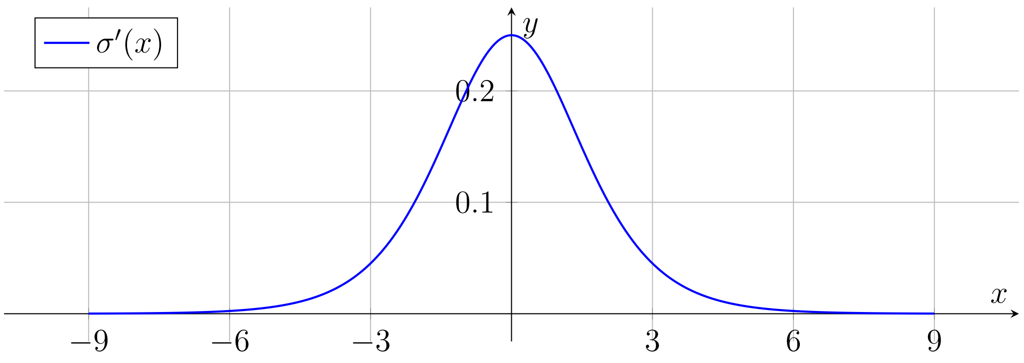


Figure 9: Derivative of sigmoid

This is another example of hyperparameter fine-tuning being essential. Minimising this penalty means pushing the weights to extreme values — the saturation region. Figure 9 shows the derivative of the sigmoid. Where the inputs to the sigmoid have high magnitudes, the gradients approach 0. This makes gradient descent more difficult for the BCE loss, by uniformly increasing confidence.

Another thing to note here is that if BCE loss has determined that a weight is wrong, this continuous penalty will actively work against BCE loss. This is because a weight being wrong means that it is positive when it should be negative, or vice versa, so the BCE loss' gradient would be pushing it towards 0, to flip the sign. But the continuous penalty's gradient pushes it away from 0, to increase its confidence.

3.3.4 Min / Max Gates Used Penalties

The final penalty I add is to control the number of gates per layer. Without this, some of the learnt networks used very few gates in the deeper layers, and many gates in the shallower layers. This is due to a problem called gradient vanishing [10], which is somewhat, but not completely, mitigated by including residual connections and connecting all layers directly to the output.

Using NAND gates in deeper layers has a number of advantages. These gates would be able to have a lower fan-in, which is good for circuit synthesis. Using deep layers is also key for performance in ML [24]. This means that for this project, the ability to use NAND gates in deeper layers is paramount.

To enable this, I design a function, "`get_used_array`" (Figure 33, Appendix), which calculates the probability of each NAND gate being used, based on (5). This can then be used in a larger function to penalise low probabilities in deeper layers — or indeed penalise probabilities that are too high if the desire is to limit certain gate usage.

I first define a probabilistic OR based on:

$$\begin{aligned} &P(A \vee B) \equiv P(A) + P(B) - P(A \wedge B) \\ &P(A \vee B) \equiv a + b - ab \quad \text{Given } a = P(A) \text{ and } b = P(B)\text{, and assuming independence} \end{aligned} \tag{26}$$

I also define:

1. A node is used $\iff$ it is connected to the inputs $\wedge$ it is connected to the outputs.

2. All inputs are vacuously connected to the inputs.

3. All outputs are vacuously connected to the outputs.

"`get_used_array`" calculates two values for each node, n. The first, $P_{input}(n)$, gives the probability that it is connected to the inputs. The second, $P_{output}(n)$, gives the probability that it is connected to the outputs. By definition 1 above, multiplying these values for each node gives the probability that it is used.

By definitions 2 and 3 above, it follows that

$$P_{input}(i) = 1 \quad \text{for inputs } i \tag{27}$$

$$P_{output}(o) \coloneqq 1 \quad \text{for outputs } o \tag{28}$$

These will be base cases for inductively defining P_{input} and P_{output}.

$$P_{input}(n) \coloneqq \bigvee_j P(w_{nj})P_{input}(j) \quad \text{for NAND gates } n \tag{29}$$

The definition above inductively defines P_{input}. j indexes all nodes connected to n, w_{nj} is the wire connecting source node j to NAND gate n, and $\bigvee$ is the n-ary probabilistic OR (26). Note that this definition of OR shares the classical OR's associativity and commutativity, making this unambiguous.

In "`get_used_array`", this value is calculated for every node in the network, with the inductive definition natrually being translated into a for loop. To programme the n-ary probabilistic OR, note:

$$\begin{aligned} P(A \vee B) &\equiv a + b - ab \\ P(A \vee B) &\equiv 1 - (1-a)(1-b) \end{aligned} \tag{30}$$

It follows naturally that:

$$\bigvee_i p_i \equiv 1 - \prod (1 - p_i) \tag{31}$$

I similarly define:

$$P_{output}(n) \coloneqq \bigvee_j P(w_{jn})P_{output}(j) \quad \text{for source nodes } n \tag{32}$$

In this definition, I use w_{jn} — the wire connecting source node n to NAND gate j.

The array that `get_used_array` returns is used to define both the max gate usage penalty, and the min gate usage penalty. I can use the same transformation followed by a `ReLU` technique used in 3.3.2. For example, to penalise low gate usage, I first define the number of nodes to use in each layer. I can sum the gate usage matrix over the first axis, and this gives an array where each element is an approximation of how many nodes are used in that layer. I can now subtract this element-wise from the first array which I pass in. The result would be negative for any layers where enough nodes are used, but would say how many more nodes need to be used otherwise. Passing this through a `ReLU`, it is now an effective penalty.

3.3.5 Readout Statistics

Looking ahead to the testing stage, many of these penalties developed are already useful readout statistics (or can easily be reworked into one). The network can be evaluated by how long it takes to find circuits, and the accuracy of the circuits found.

With this functional programming style, I can change some parameters and get a different function that is useful for different reasons. For example, I can repurpose the `get_used_array` function, passing in a step function for the weight activation. This can give a live readout of the gates currently being used by the circuit.

3.4 Adder Extension Features

In this section, I discuss features added to aid in learning adders. Just as in classical ML, it is often possible to gain significant leaps in performance by acknowledging the domain and designing the system to represent that more naturally.

3.4.1 Feature Augmentation

Feature augmentation is commonly used in ML. It entails precomputing features which may help the network, and concatenating them to the feature vector. Consider a 3-bit adder trying to compute $A_2A_1A_0+B_2B_1B_0$. Without feature augmentation, I just use the inputs and their complements. At the start of training, the network has no understanding of which bits are most significant and which bits relate to each other. As a result, it is just as likely to connect A_2 and B_0 to the same NAND gate as A_0 and B_0, for example. But for adders, it is more useful to connect A_0 and B_0 to the same NAND gate. Figure 10 shows how I augment the feature vector, so that the network can learn more easily. This greatly improved performance.

$$\begin{array}{r} A_2\,A_1\,A_0 \\ +\,B_2\,B_1\,B_0 \\ \hline \end{array} \quad \Rightarrow \quad \begin{array}{cc|cc|cccc} A_2 & B_2 & \overline{A_2} & \overline{B_2} & \overline{A_2B_2} & \overline{A_2\overline{B_2}} & \overline{B_2\overline{A_2}} & \overline{\overline{A_2}\,\overline{B_2}} \\ A_1 & B_1 & \overline{A_1} & \overline{B_1} & \overline{A_1B_1} & \overline{A_1\overline{B_1}} & \overline{B_1\overline{A_1}} & \overline{\overline{A_1}\,\overline{B_1}} \\ A_0 & B_0 & \overline{A_0} & \overline{B_0} & \overline{A_0B_0} & \overline{A_0\overline{B_0}} & \overline{B_0\overline{A_0}} & \overline{\overline{A_0}\,\overline{B_0}} \end{array}$$

Figure 10: Input representation for 3-bit adder. The left hand side represents the original inputs. The middle represents the complements that are always added to the input vector. The right hand side represents the features I augment with.

The fundamental building blocks of most adder architectures are the carry and propagate bits, calculated from the original bits. In Figure 10, the carry bits would be $C_1 = A_0B_0$, $C_2 = A_1B_1$, and $C_3 = A_2B_2$, and the propagate bits would be $P_1 = A_0 \oplus B_0$, $P_2 = B \oplus B_1$, and $P_3 = A \oplus B_2$.

This feature augmentation is just one 1-input NAND gate away from the carry bits typically used (e.g. $C_0 = \overline{\overline{A_0B_0}}$). Similarly, it is just one 2-input NAND gate away from the propagate bits (e.g. $P_0 = \overline{\overline{A_0\overline{B_0}}.\overline{B_0\overline{A_0}}}$).

3.4.2 Surrogate Network

Each layer of the network can be augmented with hard-coded functions taken from previously learnt circuits. For example, if the system first learns a k-bit adder, and then an n-bit adder, where $n > k$; then adding some extra NAND gates in each layer, which are hard-coded to compute what the terms k-bit adder used, may improve performance. I do not add extra NAND gates to the output layer. This did improve performance in many cases, but other features like the "`min_gates_used_penalty`" proved to be more useful.

3.5 Image Classification Extension Features

I similarly developed domain-specific features for image classification tasks. To develop and test this, I used the MNIST dataset. I use a step function to convert the images into binary images. This allows me to use them as inputs to the NAND networks.

3.5.1 Pooling Filters

Similarly to the feature augmentation used for the adders, the images can be augmented with pooling. Figure 11 shows an example of this. There are many useful filters that are

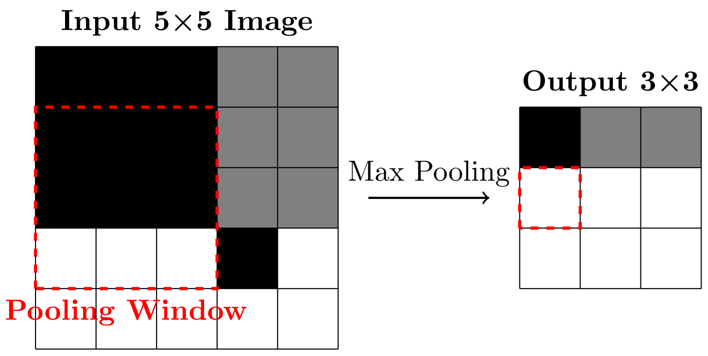


Figure 11: Filter example

An example of applying a 3 × 3 max pooling filter to a 5 × 5 greyscale image. In this example, there is no padding, and the stride is 1.

often used in image processing, but what makes the min and max pooling filters promising is they be simply implemented with NAND gates. Since the images are binarised, a max pooling filter is simply an AND gate, and a min pooling filter is an OR gate. I previously showed how these can be implemented with NAND gates; see 1.1.

3.5.2 Convolutional Layers

Figure 34 and Figure 35 (Appendix) show the functions I developed to emulate convolutional layers. As mentioned earlier, this was unsuccessful; it is unclear whether this is due to a bug in the code or an underlying limitation of NAND networks.

4 Evaluation

Due to the research nature of the project, the evaluation was done in parallel with the implementation. I first evaluate simple circuits that I generated to validate that the idea is feasible. After adding each feature, I ran experiments to understand which combinations of hyperparameters work well.

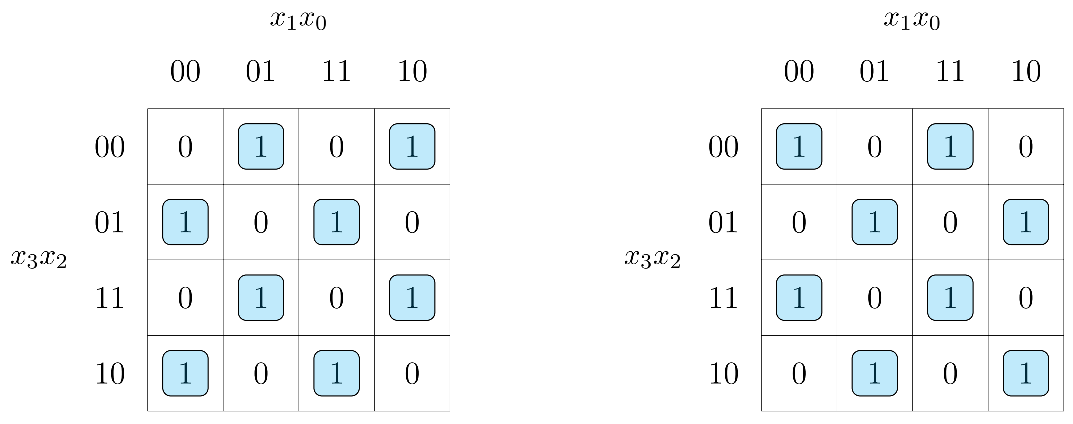
4.1 General Circuits

My first core criterion was to:

> Create a system that can learn a circuit to compute any Boolean operation with up to 4 inputs (any 16-row truth table)

This criterion was met. To test this exhaustively, it is sufficient to test one example from each “NP” class. There is some literature [1] about “NPN” classes. These are sets of Boolean functions. f and g are in the same NPN class if one can be transformed to the other by some combination of input Negation (N), input Permutation (P) and output Negation (N). Because I give the system the set of inputs and their complements, it is agnostic to input permutation and negation. However, the output negation may affect the NAND gates it can learn. Thus, for this test, I need to test at least one function from each NP class (input negation and permutation).

Since there are 222 NPN classes for 4-input truth tables, it follows that there are at most 444 NP classes. This is because each NPN class may or may not split into 2 NP classes by removing the output negation equivalence. For each class, I test a “canonical representation”, and its complement. This is sufficient to show that the success criterion was met.



(a) XOR (b) XNOR

Figure 12: 4-input XOR and XNOR Karnaugh Map

To carry out this test, I first ran some smaller experiments to learn good values to use, some of which are described in the next section (4.2). I used the normal distribution derived in 3.2.2, with $\sigma = 3.5$. I used an architecture of $[8, 16, 1]$, meaning that there were 8 inputs (the 4 inputs and their complements), 16 hidden NAND gates, and 1 output. It is known that at least 8 NAND gates are needed in the hidden layer. This would be to learn the NP class including $A \oplus B \oplus C \oplus D$ and the class including its complement (Figure 12). I give the test a timeout of 60 seconds, meaning that for each class, if it does not find a solution within 60 seconds, it restarts and tries with a new random initialisation.

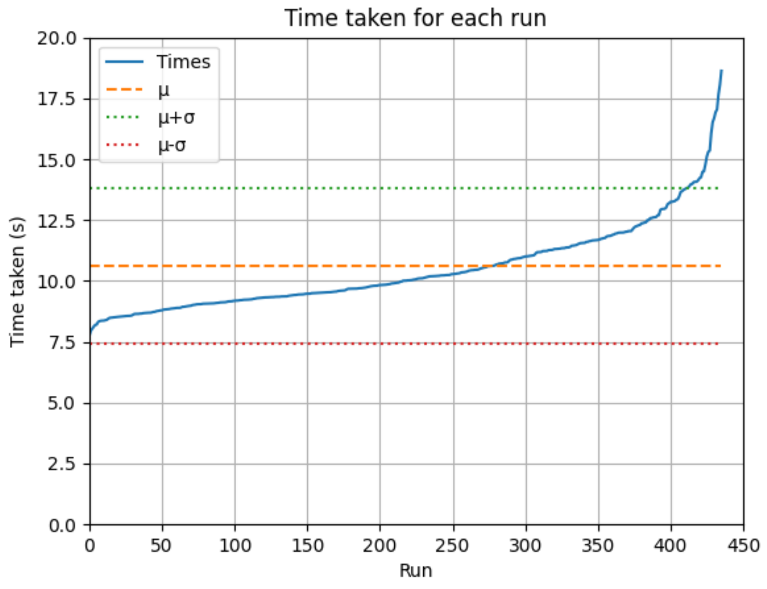


Figure 13: Training time graph

Of the 444 functions attempted, 436 were solved within 60 seconds. These took an average of 10.6 ± 3.19 seconds. Of the 8 that were not solved in 60 seconds, one was solved in just over 60 seconds before the restart, six were solved after one restart, and the eighth after two restarts. This means that I can say with confidence that the system can be used to learn any 4-input functions. Figure 13 shows the total time taken for each of the 436 functions it learnt within 60 seconds. This includes the one-time costs of initialisation, and `JIT` compilation.

The tests also show the importance of initialisation. The eight functions that were unable to learn within 60 seconds were because they had poor initialisation. I verified that this was the case, because after restarting with a new initialisation, they could be learnt in a reasonable time. The low standard deviation shown in the graph validates that 60 seconds is a sensible timeout for this test.

Before discussing how I met the second criterion, I first discuss a second experiment. I randomly generated truth tables for an 8-input logic function. The goal of this experiment is to show that with 1 hidden layer, sufficient width, and sufficient training time, the system can simply memorise truth tables, and return a circuit which can be derived by SoP (Figure 1).

The issue here is unreasonably high fan-ins. One thing NAND networks excel at is breaking these high fan-in gates down in an efficient way. This means that if more hidden layers are made available, they can efficiently use NAND gates in all of the layers, with each NAND gate having reasonable fan-ins. This is particularly prevalent when we include a `max_fan_in_penalty`, as described in 3.3.2. There is no SoP equivalent of this. This is the NAND network learning about a deeper underlying structure in the data. This experiment verifies that this is learning an underlying structure, by showing the behaviour is not present in randomly generated truth tables.

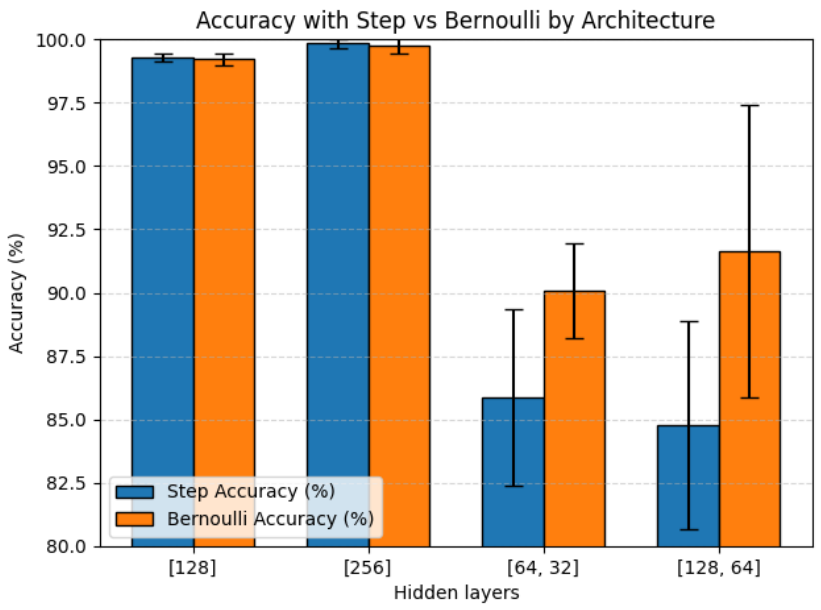


Figure 14: Accuracy by architecture for randomly generated 8-bit truth tables

Note the y-axis starts at 80%

To ensure that it must use both layers, I use the `max_fan_in_penalty`, with a coefficient of 1, and a `max_fan_in` of 16. Figure 14 shows how the network struggles to learn with 2 layers for randomly generated truth tables. I show in 4.2 that this is not the case for circuits with structure.

I used a 10 minute timeout for this test — with 5 repeats for each architecture. Step accuracy is the accuracy of the circuit when we use the step function to discretise it (any positive weights become wires). Bernoulli accuracy is the accuracy of the circuit when we sample a Bernoulli distribution with $p = \sigma(w)$ to discretise it.

4.2 Adders

My second and final core criterion was to:

> Create a system that can learn a circuit to compute more complex Boolean operations (e.g. an 8-bit adder).

This criterion was met. Before showing the circuit, I first show how I arrived there. First, I repeat the test in Figure 14, but with a 4-bit adder. This also has eight inputs, and everything else about the test is the same. There are two differences, however:

1. The data is now structured.

2. There are now 5 outputs.

Having five outputs should make it harder to learn. I am asking the system to learn 5 times as much data in the same amount of time. This test shows that NAND networks are better at learning structured data than unstructured, which may imply that they share many other benefits that regular NNs have. Figure 15 shows that it indeed learnt

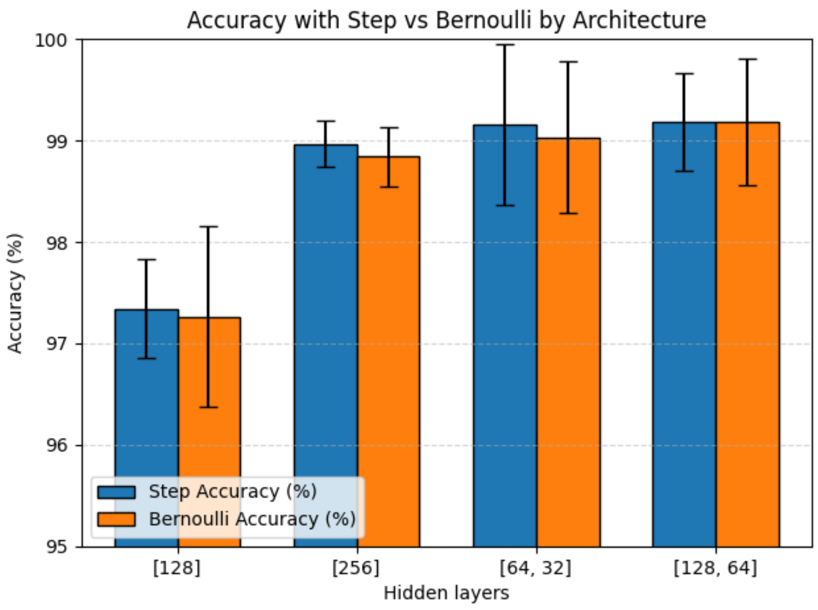
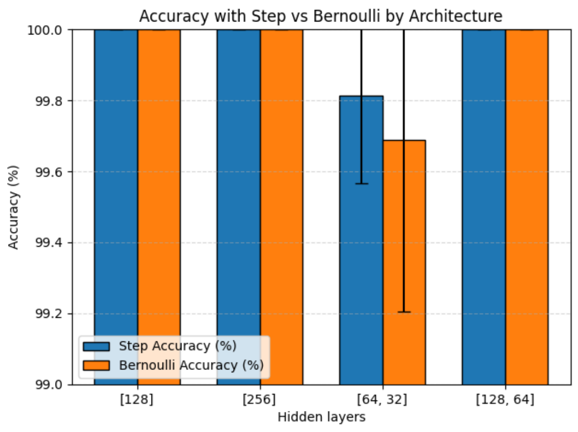


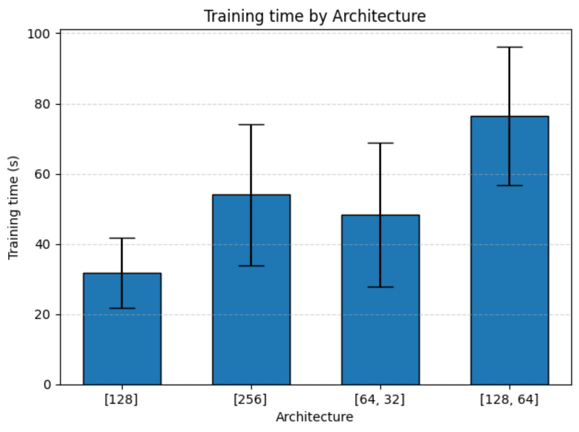
Figure 15: Accuracy by architecture for 4-bit adders

Note the y-axis starts at 95%

much better with the 4-bit adders. In fact, it was particularly promising that we had even better performance with 2 layers, rather than much worse performance. Another thing to notice from these results is the relative performance of the Bernoulli accuracy vs the step accuracy.



(a) Accuracy graph



(b) Training time graph

Figure 16: Results with feature augmentation

I now repeat the same experiment, but with the feature augmentation described in 3.4.1. The results in Figure 16 show how effective the feature augmentation is. Recall that the timeout I set for the earlier experiments was 10 minutes. With feature augmentation, the circuits are learnt significantly faster (Figure 16b), and better (Figure 16a). 18/20 learnt to 100% accuracy. The poor results for the [64, 32] architecture are due to the layers not being wide enough, which makes the runs very dependent on getting a good initialisation.

Before sharing more results, or doing more tests, I first observe a domain-specific simplification. Assuming I already have some optimal (n-1)-bit adder, I need only learn the two most significant bits (MSBs) of an n-bit adder.

$$\begin{array}{r} A_3\,A_2\,A_1\,A_0 \\ +\,B_3\,B_2\,B_1\,B_0 \\ \hline C_3\,S_3\,S_2\,S_1\,S_0 \end{array} \quad \Rightarrow \quad \begin{array}{r} A_2\,A_1\,A_0 \\ +\,B_2\,B_1\,B_0 \\ \hline C_2\,S_2\,S_1\,S_0 \end{array}$$

Figure 17: 4-bit adder from 3-bit adder

Figure 17 shows the outputs that a 4 and 3-bit adder share — S_2, S_1 and S_0. This relationship generalises to n and $n-1$ bits. S_{n-1}, S_{n-2}, ..., S_0 are functions of A_{n-1}, A_{n-2}, ..., A_0 and B_{n-1}, B_{n-2}, ..., B_0, so it can be computed with the efficient $(n-1)$-bit adder we have already learnt. I need to learn both S_n and C_n for this n-bit adder. In general, this means that assuming that we have learnt an efficient $(n-1)$-bit adder, it is sufficient to learn just the 2 MSBs of the n-bit adder.

4.2.1 Hyperparameter search

I use 4-bit adders for a hyperparameter search. This is because there are only 256 training examples, so they are much simpler to learn than the 8-bit adders, which are the goal. Because the results in Figure 16 are so positive, going forward, I always use feature augmentation for adders. As just discussed, going forward I also only learn the two MSBs.

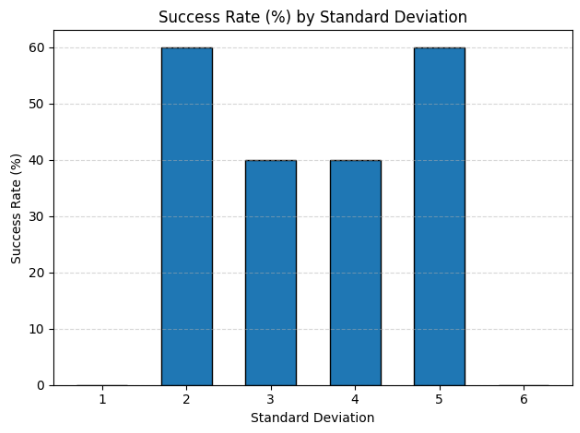
The next test I do is to identify the optimal architecture, and initialisation. As shown in Figure 16b, with feature augmentation, the system can quickly learn with both 1 hidden layer, and 2 hidden layers. The goal is no longer to learn circuits; I have verified that the system can do that. The goal now is to learn efficient circuits. So training time is now much less of a concern — the metrics going forward will include maximum fan-in, average fan-in, number of layers, and number of NAND gates.

In this test, I give the system a constant 256 NAND gates, and vary how they are arranged. Earlier testing showed that it is best to have more NAND gates in earlier layers, and less NAND gates going forward, so I keep this constant, as well as the truth table (2 MSBs of a 4-bit adder).

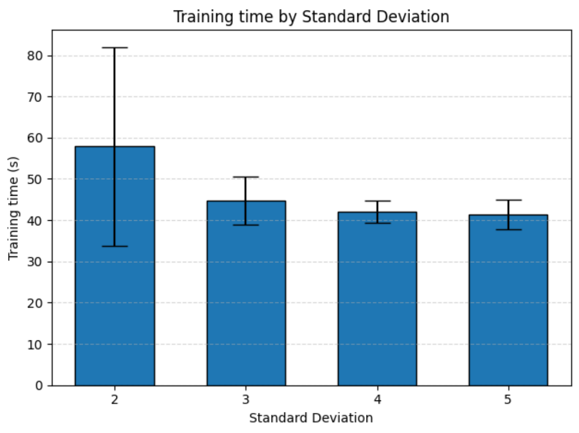
Since the aim is optimal circuits, I also used the `maximum_fan_in` penalty in this run. I found 1 to be a good value for the coefficient, so for one layer, I set the max fan-in to 6, and for two layers, I use 4. Smaller runs showed that the network struggled to learn with lower fan-ins than this, at least in the case of the 4-bit adder.

Just as before, I observed that when the goal is 100% accuracy, restarting with a new initialisation can be effective. Figure 18b shows that the circuits that reached 100% within two minutes, did so much before two minutes, just as in the case of the 4-input functions in 4.1.

I also learnt that the Bernoulli accuracy often did much better than the step accuracy (Figure 19). The development of the `continuous_penalty`, and much of the hyperparameter tuning done earlier, was from the perspective of learning a continuous circuit, and using the step function to convert it into a discrete circuit. When I changed my perspective, inspired by how LLMs use temperature, I realised that using the step function to discretise the circuit is the equivalent of using a temperature of 0.



(a) Success rates



(b) Training time graph

Figure 18: Results with (16), and hidden layers [160, 96]

The Bernoulli accuracy function is a method that enables a more natural transformation from the continuous function to a discrete circuit, and this definition of accuracy follows directly from all the assumptions the model is built on — namely (5).

One of the cases where the continuous penalty was particularly useful was when there were a large number of NAND gates. This is explained with this new perspective. The more NAND gates (and therefore wires), the less probable it becomes for the more likely option to be taken each time.

For example, assume that each weight is such that $\sigma(w) \in \{0.3, 0.7\}$. This means that the step function would match a sample from the Bernoulli distribution with a 70% probability. For a small number of weights, it would be quite likely to generate the same circuit with the step function and the Bernoulli distribution. As the number of weights increases, the step function circuit becomes less likely to be generated by the probability distribution the system has learnt.

This finding led me to stop using the continuous penalty altogether. In 3.3.3, I have already described how this penalty can be detrimental to learning an accurate circuit. I have now learnt that this penalty was changing the probability distribution being learnt, to increase the likelihood that the step discretised circuit could be generated by it.

Instead of using this penalty, I evaluate both the step and Bernoulli accuracy, and in cases where the continuous penalty may have been used, I can instead use the Bernoulli distribution to discretise the circuit.

Now I learn the 8-bit adder, with a deeper intuition gained from these tests. As discussed, I start with a 1-bit adder, and build up inductively.

4.2.2 1-bit adder

This is a very simple circuit. The optimal circuit is known to be $C_0 = A_0B_0$ and $S_0 = A_0 \oplus B_0$. To express this in NAND gates, I use $C_0 = \overline{\overline{A_0B_0}}$, and there are two efficient representations of $A_0 \oplus B_0$. One is $\overline{\overline{A_0.\overline{A_0B_0}}.\overline{\overline{A_0B_0}.B_0}}$. This shares the $\overline{A_0B_0}$ used to compute C_0. The other representation is $\overline{\overline{A_0.\overline{B_0}}.\overline{B_0.\overline{A_0}}}$. This follows directly from expressing S_0 in SoP form. Both representations have their own advantages, and can be learnt by the system. For this run, I set a max fan-in of 2, did not use feature augmentation, and used the normal distribution derived in 3.2.2, with $\sigma = 3.5$. This was trained

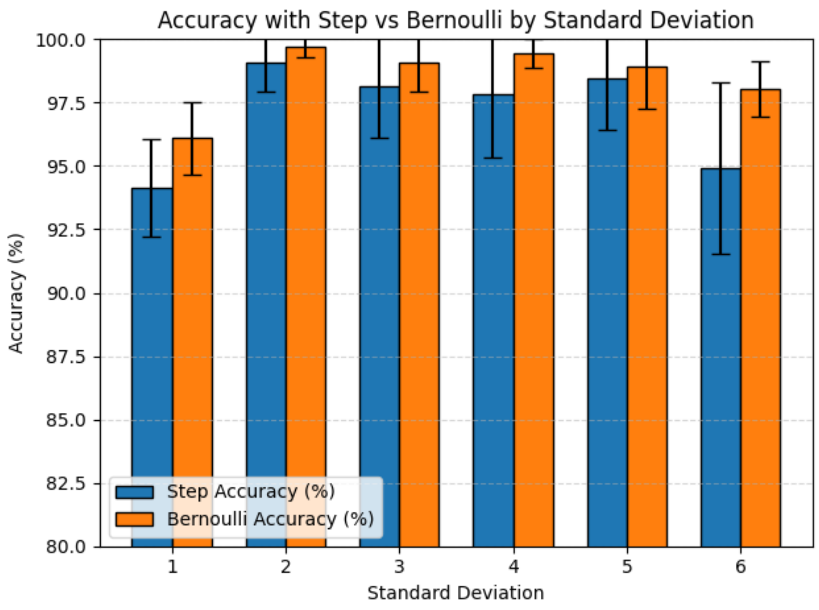


Figure 19: Accuracy by Standard Deviation (using the normal distribution derived in 3.2.2)

in 5 seconds (Figure 36, Appendix).

4.2.3 2-bit adder

In SoP form, the circuit would have a maximum fan-in of 6. This is already quite high. As a result, the aim is to learn different circuits optimising for different things. One as a direct translation of the SoP form, and the others using extra layers, but limiting the maximum fan-in. For the first circuit, I use the same hyperparameters as 1-bit, just changing the max fan-in to 6, and giving 32 NAND gates. This was trained in 6 seconds (Figure 37, Appendix).

Now to learn S_1 with a max fan-in of 4, I introduce another layer, and reuse the same architecture I know to work well from earlier tests — 160 NAND gates in the first layer, and 96 in the second. This was trained in 27 seconds (Figure 38, Appendix).

Finally, to learn a circuit with a maximum fan-in of 2, I begin to use the feature augmentation, as well as both the max and mean fan-in penalties. This was trained in 13 seconds (Figure 39, Appendix).

4.2.4 3-bit adder

The goal is now to learn C_2 and S_2 such that $A_2A_1A_0 + B_2B_1B_0 \equiv C_2S_2S_1S_0$, with S_1 and S_0 from the previous section. This is the first example where there is no optimal solution in mind, since the truth table now has 64 rows, and cannot be represented with a simple Karnaugh map. For this run, I set the maximum fan-in to 4, and I also use the mean fan-in penalty, setting the mean to 2.5. I use a coefficient of 1 for both. This was trained in 21 seconds (Figure 40, Appendix).

I also tried with a max fan-in of 3, and a mean fan-in of 2.25, and got a different circuit (in 18 seconds) (Figure 41, Appendix).

The results thus far show that I fulfilled one of the extension criterion:

The system can create circuits that optimise for desirable factors. Including minimising: delay, number of NAND gates, max number of inputs into each NAND gate and power usage

4.2.5 8-bit adder

I found that for subsequent layers, I could learn with the same 2 extra layers, and with a max fan-in corresponding to the bit being learnt. For example, I learn O_6 from the 7-bit adder, and so set the max fan-in to 7. In addition to this, I used the min gates used penalty to ensure that the second hidden layer had 7 gates used. I set the maximum number of gates in the second hidden layer to 7, and to 49 in the first layer. This reinforces the max fan-in penalty, without having to decrease the temperature. I also used the mean fan-in penalty here. I used a batch size of at most 512. These settings worked well as regularisation terms, not only leading to better circuits, but also leading to our circuits being learnt quicker. Figure 20 shows how well chosen gate usage penalties can guide the BCE loss to lower values than optimising by minimising only BCE loss.

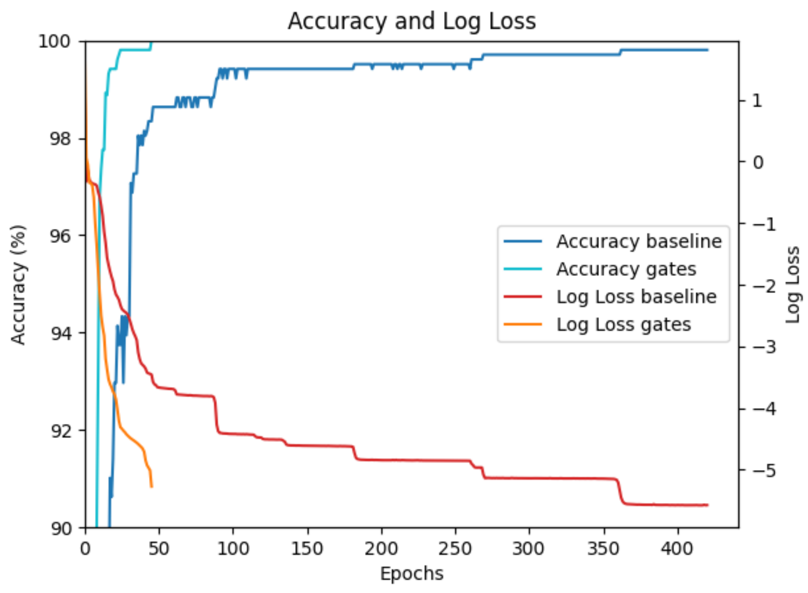


Figure 20: Accuracy and log loss in learning S_6 with and without the gate usage penalties described

All that remains is to learn the carry bit. The carry bit is logically simpler than the carry bit, and so despite needing a maximum fan-in of 8 for the 2^{nd} MSB, I can use a max fan-in of 6 for this bit Figure 45, but this time without a mean fan-in penalty.

4.2.6 Evaluation against SOTA

The circuit developed is analogous to a carry lookahead adder (CLA). With a CLA, there is a constant critical path length, which does not depend on the number of bits. However, the maximum fan-in grows linearly with the number of bits. This is exactly what we see with the NAND networks. However, beginning from a CLA, it may not be possible to achieve the critical path lengths NAND networks achieve. The critical path length for a CLA with any logic gates is 4, as seen in Figure 21. Converting it to NAND gates, and

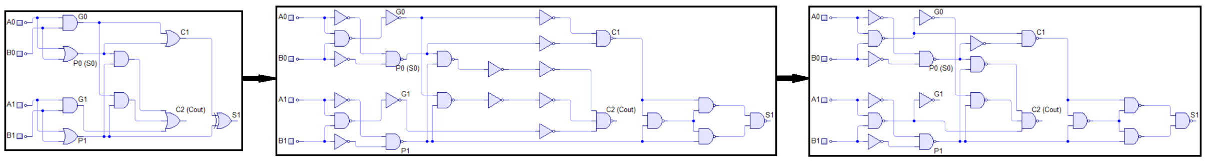


Figure 21: 2-bit CLA to NAND network.

applying double negation elimination, the critical path length becomes 7. The adders I developed have a critical path length of 5 (complement layer, feature augmentation layer, 2 hidden layers, and output layer). This beats the critical path length of 7 for the NAND gate CLA, whilst keeping the same maximum fan-in.

The other commonly used architectures in industry are those where each gate has a fan-in of 2, but the critical path length grows logarithmically with the number of bits. These are divide-and-conquer adders. I hypothesise that by limiting the mean fan-in and max fan-in to 2, as I did for the 2-bit adder, and using the appropriate number of layers, NAND networks may be able to learn such circuits.

4.3 Image Classification

Another extension criterion was:

> The system can be repurposed to be used for classical ML tasks, such as image classification.

I met this success criterion. In this section, I show how NAND networks can be used to tackle the MNIST dataset. This is a common benchmark for ML image classification, and particularly in TinyML.

I first test test three things: architecture, pooling, and the min gates penalty.

I test two architectures — [2048] and [1024, 768, 512, 256]. This is to see if the system can take advantage of the extra hidden layers, to learn some underlying structure. Furthermore, I test with and without pooling. I either add no pooling, or add both min and max pooling, each with a filter width of 3, a stride of 1, and no padding. I also test the min gates penalty. For this I try no min gate penalty, a minimum of 50% node usage, and a minimum of 100% node usage. This gives $2 \times 2 \times 3 = 12$ configurations, each of which I give 15 minutes to train. I use the findings from this to do more testing.

Since I use the same time limit for all of the runs, the more demanding set-ups have less epochs to train and do not converge. This includes those where pooling is included (our input goes from 1568 elements to 4272) and where there are extra hidden layers (since feed forward is sequential).

The metric I used to compare these set-ups is testing accuracy, as this gives good insight into the system's generalisation power. I begin by comparing the baseline, 1 hidden layer, no feature augmentation, and no min gates penalty, to set-ups where I change exactly one of these. The most notable finding of this experiment is seen in Figure 22, although it has less than half of the epochs to train, the network with 4 hidden layers reached a higher accuracy in the same time.

I now take the deeper network as the baseline, and test again to see if pooling or the min gates penalty can give a further improvement. I found the min gates penalty in particular encourages the network to make use of the NAND gates given to it, leading to a better solution Figure 23.

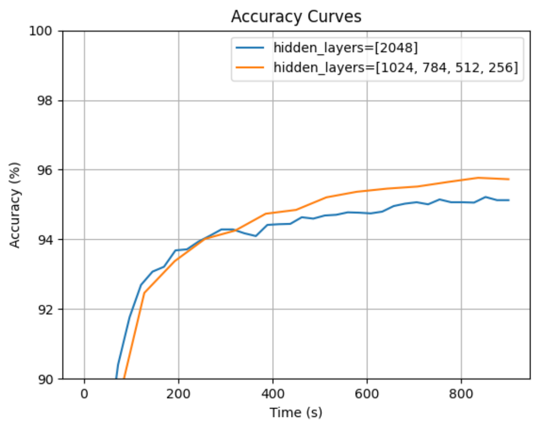


Figure 22: Testing (step) accuracy by hidden layers

Note the y-axis starts at 90%

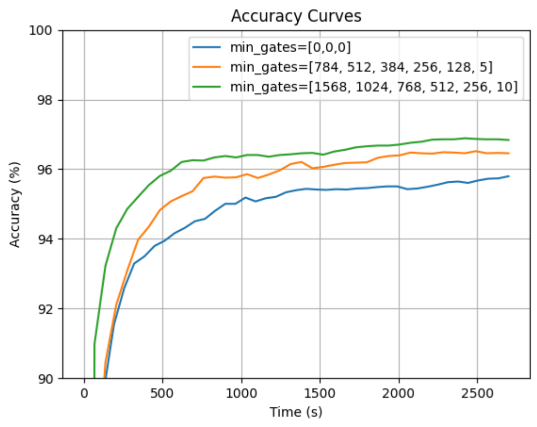


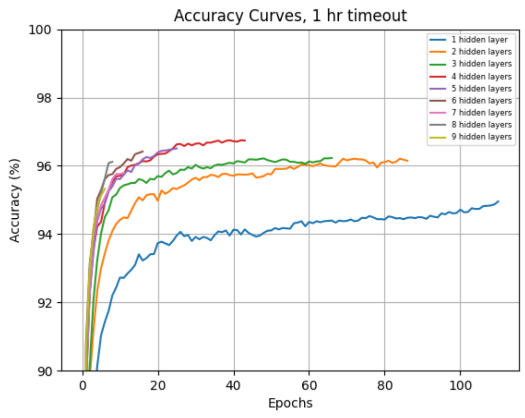
Figure 23: Testing (step) accuracy by min gates

Pooling again had an insignificant effect, and in fact, its overall effect seems to be negative because it makes each epoch take so much longer.

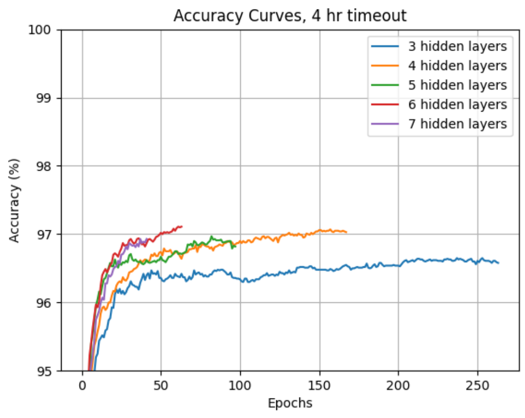
These tests show that the system is able to learn better with deeper networks, and more NAND gates. This seems obvious, but in the case that I do not set the min gates penalty, which has the worst performance (Figure 23), the BCE loss alone does not lead the system to use the full network. It uses just $[1325, 355, 100, 48, 11, 10]$ out of $[1568, 1024, 768, 512, 256, 10]$.

From this, I run further tests, to learn a good number of layers to use. I tested varying numbers of hidden layers, where each additional hidden layer has 256 more NAND gates than the last. I used a min gates array that enforces at most two nodes in each layer cannot be used. Figure 24 shows the results in allowing these tests to run for 1 and 4 hours. The results show that more hidden layers result in less epochs. They also show the diminishing returns — both from training time, and extra hidden layers.

These tests are slow to run, but I realised that just the first epoch is quite informative. Thus, I run a final test, to understand the interactions between learning rate, batch size, and the min gates penalty. All of this was with the same architecture of $[1568, 1536, 1280, 1024, 768, 512, 256, 10]$, and just one epoch. I expect that with higher



(a) Accuracy by epoch with 1 hour timeout



(b) Accuracy by epoch with 4 hour timeout

Figure 24: Results with different numbers of layers.

learning rates, the network will be quick to add or remove wires, since the steps would become quite big. But since our initialisation is sparse by design, this means that higher learning rates will lead to more wires being learnt — while lower learning rates will make more of an effort to learn the wires that are necessary. I also suspect that the excess wires learnt with the higher learning rates are more likely to be in shallower layers, since they naturally have gradients with a higher magnitude, as discussed earlier. I run tests to check these hypotheses.

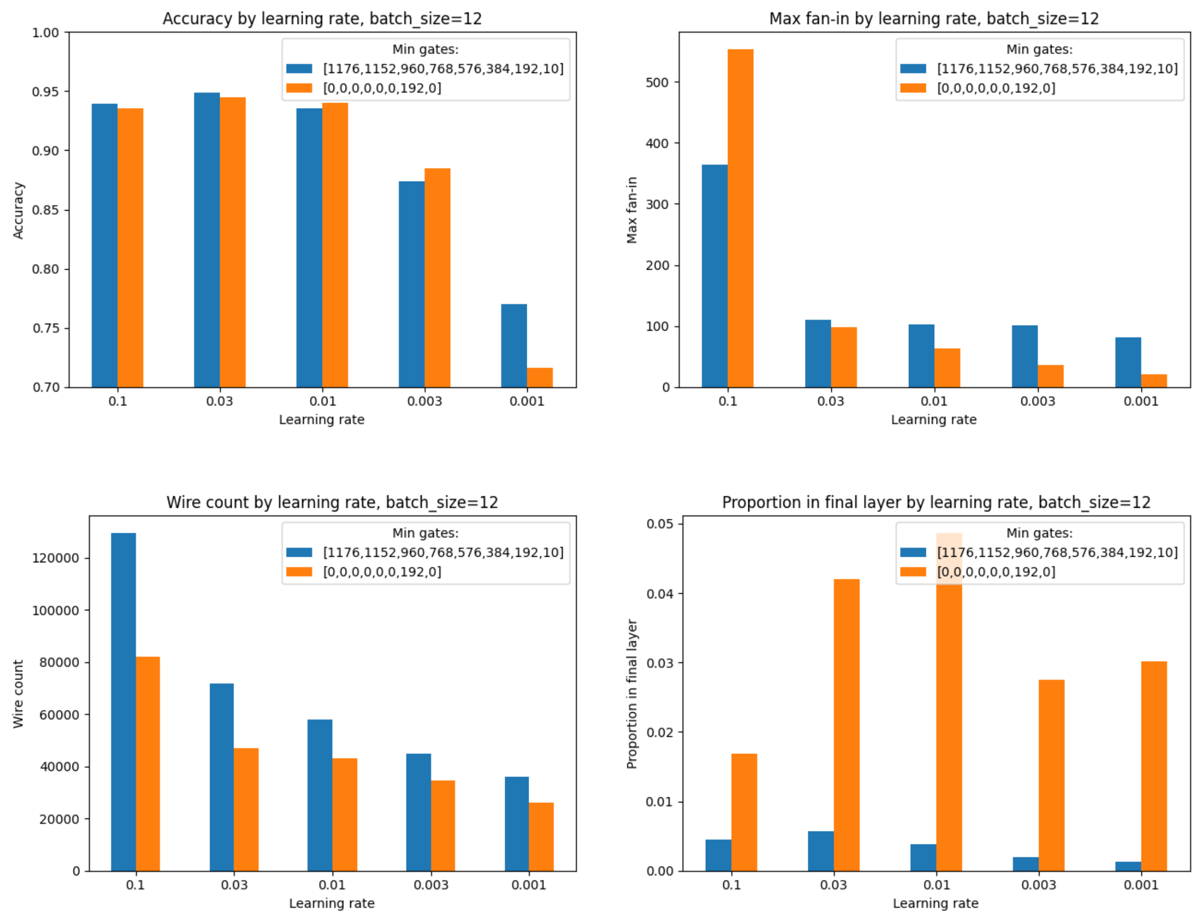


Figure 25: Statistics after 1 epoch, each trained with a batch size of 12

Figure 25 confirms many of the hypotheses. It shows that lower learning rates are more careful in adding wires, leading to sparser circuits (which is good for circuit synthesis), without sacrificing accuracy. As the learning rate gets too low however, more epochs would be needed to see similar accuracies.

4.3.1 Evaluation against SOTA

I produce 3 networks to compare against SOTA. The first will have just 1 hidden layer, with no min gate usage. This will be the smallest network. Next, I trade some size for accuracy, and use 2 hidden layers, with a min gate usage. Finally, to maximise accuracy I train a network with 6 hidden layers. Based on the tests above, I use a learning rate of 0.03, batch size of 12. For the multi-layer networks, I use a min gates penalty that enforces at least 75% of NAND gates in the final hidden layer are used.

Figure 26 shows how varying the number of hidden layers can affect log loss, accuracy, and number of epochs. It shows how the extra layers allow the network to find lower minima in the same time, despite having fewer epochs.

These results are close to SOTA (Table 3) [18] [14], but fall short. I believe the system could give better performance with more time for hyperparameter tuning, and model training.

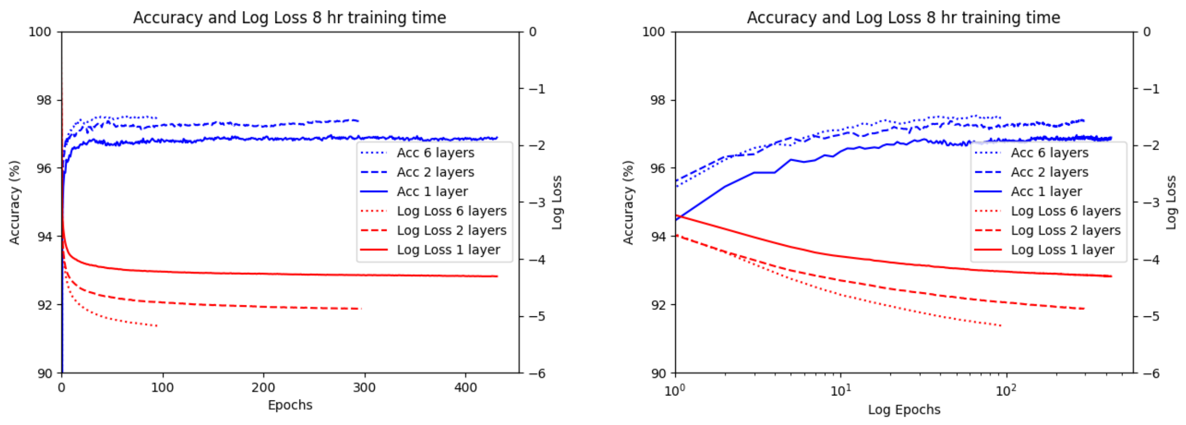


Figure 26: Accuracy and Log Loss, 8 hr training time

5 Conclusions

Based on meeting the two core criteria, and two out of four of the extension criteria, I judge the project to have been successful.

5.1 Further Development

One direction for further development would be to add more penalties. One in particular would have been the number of wires in the final circuit, which would be useful for circuit synthesis. I would also work on the PTO more. In 3.3.1, I discussed three ways I optimised the circuits post-training, and I mentioned a fourth way in which I could, but I do not. I would also work further on the convolutional layers, since if I can show good results with convolutional layers, this would give me confidence in exploring further architectures, such as attention.

The most practical step for further development is the natural next step — converting the output binary weight matrices into real combinational logic circuits. This would not simply be placing the wires where the matrix says there should be, since some of the learnt circuits have high fan-ins (Table 3). In reality, many of these gates would be decomposed into smaller gates.

5.2 Lessons Learned

The project set out to investigate "Whether gradient descent can be used to fit a Boolean function $f : \{0,1\}^m \to \{0,1\}^n$ with a NAND network", with some extensions of guiding the search to find "simpler" circuits (2.1). I have shown that this is possible. NAND networks are able to memorise truth tables of up to 4-inputs, they are able to learn adders of up to 8-bits. They even able to classify images. I also showed that this search can effectively be guided to find "simpler" circuits.

Due to the research nature of the project, and the agile software engineering methodology used, it was difficult to set appropriate success criteria before beginning the project. As a result, during the "review" stage, I decided to stop pursuing two of the extension criteria. The first was to inductively learn an adder. It was difficult to learn 8-bit adders with 100% accuracy with all of the training data, with diminishing accuracy returns as the number of epochs increased (Figure 20). This suggested that with just 75% of

Table 3: MNIST performance.

Model	Acc. (%)	# Param.	Space	OPs	FLOPs
NAND Networks			# Wires	# Gates	Max fan-in
NAND Net (1 hidden)	96.89	251 840	57 142	642	163
NAND Net (2 hidden)	97.39	5 111 188	81 467	2 402	326
NAND Net (6 hidden)	97.51	15 343 775	117 694	3 268	278
Deep Differentiable Logic Gate Networks					
Diff Logic Net (small) [18]	97.69	48 000	23KB	48 K	—
Diff Logic Net [18]	98.47	384 000	188KB	384 K	—
Sparse Neural Networks					
Variational Dropout [17]	98.08	4 000	—	(8 M)	8 K
SET-MLP [16]	98.74	89 797	—	(180 M)	180 K
Sparse Function Net [6]	94.2	$3 \times 1\,849$	—	—	> 2 K
SpArSe [5]	98.64	—	2.77KB	—	—
SpArSe [5]	96.49	—	1.44KB	—	—
BonsaiOpt [13]	95.88	—	63.9KB	—	—
μNAS [14]	99.19	—	0.48KB	—	—

the training data, it would be much more difficult. The other extension I decided not to complete was to learn Finite State Machines (FSMs). This seems achievable, but I underestimated the amount of time the rest of the project would take.

Bibliography

[1] C.R. Baugh. Generation of representative functions of the npn equivalence classes of unate boolean functions. *IEEE Transactions on Computers*, C-21(12):1373–1379, 1972.

[2] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992.

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

[4] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[5] Igor Fedorov, Ryan P. Adams, Matthew Mattina, and Paul N. Whatmough. Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers, 2019.

[6] Adam Gaier and David Ha. Weight agnostic neural networks, 2019.

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

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[9] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network, 2015.

[10] Sepp Hochreiter. The vanishing gradient problem during learning recurrent neural nets and problem solutions. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(02):107–116, 1998.

[11] Max Jaderberg, Andrea Vedaldi, and Andrew Zisserman. Speeding up convolutional neural networks with low rank expansions, 2014.

[12] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[13] Ashish Kumar, Saurabh Goyal, and Manik Varma. Resource-efficient machine learning in 2 KB RAM for the internet of things. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1935–1944. PMLR, 06–11 Aug 2017.

[14] Edgar Liberis, Łukasz Dudziak, and Nicholas D. Lane. μ nas: Constrained neural architecture search for microcontrollers. In *Proceedings of the 1st Workshop on Machine Learning and Systems*, EuroMLSys '21, page 70–79, New York, NY, USA, 2021. Association for Computing Machinery.

[15] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks, 2018.

[16] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H. Nguyen, Madeleine Gibescu, and Antonio Liotta. Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science. *Nature Communications*, 9(1), June 2018.

[17] Dmitry Molchanov, Arsenii Ashukha, and Dmitry Vetrov. Variational dropout sparsifies deep neural networks, 2017.

[18] Felix Petersen, Christian Borgelt, Hilde Kuehne, and Oliver Deussen. Deep differentiable logic gate networks, 2022.

[19] Daniel A. Roberts. Sgd implicitly regularizes generalization error, 2021.

[20] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. A comprehensive survey on model quantization for deep neural networks in image classification. *ACM Transactions on Intelligent Systems and Technology*, 14(6):1–50, November 2023.

[21] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *International conference on machine learning*, pages 1139–1147. PMLR, 2013.

[22] Zheng Tang, Makoto Kuratsu, Okihiko Ishizuka, and Koichi Tanno. A boolean algebra based learnable network. *Ieej Transactions on Electronics, Information and Systems*, 119:1223–1231, 1999.

[23] Dmitry Yarotsky. Error bounds for approximations with deep relu networks. *Neural networks*, 94:103–114, 2017.

[24] Jingwei Zhang, Tongliang Liu, and Dacheng Tao. An information-theoretic view for deep learning, 2018.

Appendices

Algorithm 1 Gradient Descent

```
Require: $\eta$ : Learning rate
Require: $\mathcal{L}(w)$ : Loss function
Require: $w_0$ : Initial weight vector
 1: $t \leftarrow 0$
 2: while $w_t$ not converged do
 3:     $t \leftarrow t + 1$
 4:     $g_t \leftarrow \nabla_w \mathcal{L}(w_{t-1})$                ▷ Compute gradient
 5:     $w_t \leftarrow w_{t-1} - \eta \cdot g_t$                ▷ Update weights
 6: end while
 7: return $w_t$
```

Algorithm 2 Adam: Adaptive Moment Estimation [12]

```
Require: $\eta$ : Learning rate
Require: $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates
Require: $\mathcal{L}(w)$ : Loss function with weights $w$
Require: $w_0$ : Initial weight vector
 1: $m_0 \leftarrow 0$ (Initialise $1^{st}$ moment vector)
 2: $v_0 \leftarrow 0$ (Initialise $2^{nd}$ moment vector)
 3: $t \leftarrow 0$ (Initialise time step)
 4: while $w_t$ not converged do
 5:     $t \leftarrow t + 1$
 6:     $g_t \leftarrow \nabla_w \mathcal{L}_t(w_{t-1})$        ▷ Get gradients w.r.t. loss at time step t
 7:     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$        ▷ Update biased first moment estimate
 8:     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$        ▷ (Update biased second raw moment estimate
 9:     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$        ▷ Compute bias-corrected first moment estimate
10:     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$        ▷ Compute bias-corrected second raw moment estimate
11:     $w_t \leftarrow w_{t-1} - \eta \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$        ▷ Update weights
12: end while
13: return $w_t$        ▷ Resulting weights
```

Sample Functions

```python
step = jax.jit(lambda x: jnp.where(x>0, 1, 0))

def sample(seed, p):
    key = jax.random.fold_in(jax.random.PRNGKey(0), seed)
    return jax.random.bernoulli(key, p)

@jax.jit
def bern(x):
    x = jnp.asarray(x, dtype=jnp.float32)
    seeds = jax.lax.bitcast_convert_type(x, jnp.uint32).ravel()
    probs = jax.nn.sigmoid(x).ravel()
    samples_flat = jax.vmap(sample)(seeds, probs)
    return samples_flat.reshape(x.shape)

temp = jax.jit(lambda x: jax.nn.sigmoid(x/temperature))
weight_activation_dict = {"cont": jax.nn.sigmoid,
                          "disc": step,
                          "rand": bern,
                          "temp": temp}
```

Figure 27: `weight_activation_dict`.

```python
@partial(jax.jit, static_argnames="weight_activation")
def and_helper(
    x: jnp.ndarray,
    w: jnp.ndarray,
    weight_activation: str="cont") -> float:
    return jnp.prod(
        1 + jnp.multiply(x, weight_activation_dict[weight_activation](w))
          - weight_activation_dict[weight_activation](w))
```

Figure 28: Python function `and_helper`.

```python
@partial(jax.jit, static_argnames="weight_activation")
def forward(
    xs: jnp.ndarray, ws: jnp.ndarray, weight_activation: str="cont") -> float:
    return 1 - jnp.prod(jax.vmap(
        and_helper, in_axes=(0, 0, None))(xs, weights, weight_activation))
```

Figure 29: Python function `forward`.

```python
@partial(jax.jit, static_argnames="weight_activation")
def feed_forward(inputs: jnp.ndarray, weights: jnp.ndarray,
    weight_activation: str="cont") -> jnp.ndarray:
    xs = jnp.array([jnp.pad(
        inputs,(0, i_3-len(inputs)), mode="constant", constant_values=1)])
    for layer_i in range(i_0-1):
        next = jax.vmap(forward, in_axes=(None, 0, None))(
            xs, weights[layer_i], weight_activation)
        next = jnp.array([jnp.pad(
            next,(0, i_3-len(next)), mode="constant", constant_values=1)])
        xs = jnp.vstack([xs, next])
    return jax.vmap(
        forward, in_axes=(None, 0, None))(
            xs, weights[i_0-1], weight_activation)[:outs]
```

Figure 30: Python function `feed_forward`.

```python
@jax.jit
def max_fan_in_penalty(weights: Network, max_fan_in: int, temperature: float
                       ) -> float:
    fan_ins = jnp.array([])
    for layer in weights:
        fan_ins = jnp.concatenate((fan_ins, jax.vmap(
            lambda x:jnp.sum(jax.nn.sigmoid(x/temperature)))(layer)))
    temp = jax.nn.relu(fan_ins-max_fan_in)
    return jnp.sum(jax.nn.softmax(temp)*temp)
```

Figure 31: Python function `max_fan_in_penalty`.

```python
@jax.jit
def continuous_penalty(weights: Network, num_wires: int) -> float:
    s = sum([jnp.sum(
        1-jax.nn.sigmoid(jnp.absolute(layer))) for layer in weights])
    return s/num_wires
```

Figure 32: Python function `continuous_penalty`.

```python
@partial(jax.jit, static_argnames="weight_activation")
def get_used_array(weights: Network, weight_activation: str) -> float:
    prob_weights = [
        weight_activation_dict[weight_activation](layer) for layer in weights]
    used_back = jnp.zeros(shape=(len(arch), i_3))
    used_back = used_back.at[len(arch)-1, :outs].set(jnp.ones(shape=outs))
    for layer in range(len(arch)-1, 0, -1):
        temp = (prob_weights[layer-1]
                * used_back[layer, :arch[layer]][:, jnp.newaxis, jnp.newaxis])
        temp = cont_or_arr(temp, axis=0)
        used_back = used_back.at[input_layers(layer)].set(
            cont_or(used_back[input_layers(layer)], temp))
    used_for = jnp.zeros(shape=(len(arch), i_3))
    used_for = used_for.at[0, :new_ins].set(jnp.ones(shape=new_ins))
    for layer in range(1, len(arch)):
        temp = (prob_weights[layer-1][:arch[layer]]
                * used_for[input_layers(layer)][jnp.newaxis,:,:])
        temp = cont_or_arr(temp, axis=(1,2))
        used_for = used_for.at[layer, :arch[layer]].set(
            cont_or(used_for[layer, :arch[layer]], temp))
    return used_back*used_for
```

Figure 33: Python function get_used_array.

```python
@partial(jax.jit, static_argnames=('n', "weight_activation"))
def forward_conv(
    xs: jnp.ndarray,
    weights:jnp.ndarray,
    s: int,
    n: int,
    weight_activation: str="cont") -> float:
    """
    Applies a filter of width `w` and stride `s` to the input array `xs`.

    Parameters:
    xs - an array of shape (old_channels, old_n, old_n), the input data
    weights - an array of shape (channels, old_channels, w, w), containing
    the filter weights
    s - the stride of the filter
    n - the new height and width of the picture
    weight_activation - a string which is "cont" or "disc", which
    determines if we use a sigmoid or a step function

    Returns:
    An array of shape (channels, n, n), the result of applying the filter.
    """
    w = weights.shape[2]
    old_channels = xs.shape[0]
    channels = jnp.arange(weights.shape[0])
    return jax.vmap(
        lambda c: jax.vmap(
            lambda i: jax.vmap(
                lambda j: 1-and_helper(
                    jax.lax.dynamic_slice(xs,
                    (0, i*s, j*s),
                    (old_channels, w, w)),
                    weights[c], weight_activation)
            )(jnp.arange(n))
        )(jnp.arange(n))
    )(channels)
```

Figure 34: Python function `forward_conv`.

```python
@partial(jax.jit, static_argnames="weight_activation")
def feed_forward_conv(
    xs: jnp.ndarray,
    weights:jnp.ndarray,
    imgs_list: List[jnp.ndarray],
    weight_activation: str="cont") -> jnp.ndarray:
    """
    Applies all of the convolutional layers to the input

    Parameters:
    xs - an array of shape (n, n), the input data
    weights - the list of weights
    imgs_list - a list of the scaled down images
    weight_activation - a string which is "cont" or "disc", which
    determines if we use a sigmoid or a step function

    Returns:
    The result of applying the convolutional layers, ready to be passed
    into the dense layers
    """
    for i, (ws, (_,_,s,n)) in enumerate(zip(weights, convs)):
        temp = forward_conv(xs, ws, s, n, weight_activation)
        xs = jnp.concatenate(
            [imgs_list[i], 1-imgs_list[i], temp, 1-temp], axis=0)
    return xs
```

Figure 35: Python function `feed_forward_conv`.

Sample Adders

Below are some output circuits from the learnt adders. Apart from the circuit itself, I include some readout data. The Used: ... Out of: ... notation specifies how many nodes in each layer the network ended up using. This is "from the network's perspective". Meaning for a k-bit adder there would be $8k$ nodes in the input layer — $2k$ for the bits being added, $2k$ for their complements, and $4k$ for the feature augmentation as in 3.4.1.

Used:	[4, 3, 2]
Out of:	[4, 16, 2]
Max fan-in:	2
Average fan-in:	1.8

$$C_0:\quad \neg\neg(A_0.B_0)$$
$$S_0:\quad \neg(\neg(B_0.\neg A_0).\neg(A_0.\neg B_0))$$

Figure 36: 1-bit adder

Used:	[8, 9, 2]
Out of:	[8, 32, 2]
Max fan-in:	6
Average fan-in:	3.36

$$\begin{aligned} C_1 :&\quad \neg(\neg(A_0.B_1.B_0).\neg(A_1.A_0.B_0).\neg(A_1.B_1)) \\ S_1 :&\quad \neg(\neg(A_1.\neg A_0.\neg B_1).\neg(A_1.A_0.B_1.B_0).\neg(B_1.\neg A_1.\neg A_0) \\ &\quad .\neg(A_0.B_0.\neg A_1.\neg B_1).\neg(B_1.\neg A_1.\neg B_0).\neg(A_1.\neg B_1.\neg B_0)) \\ S_0 :&\quad \text{From 1-bit adder(Figure 36)} \end{aligned}$$

Figure 37: 2-bit adder, max fan-in 6

Used:	[6, 5, 1, 1]
Out of:	[8, 160, 96, 1]
Max fan-in:	4
Average fan-in:	2.62

$$S_1:\quad \neg(\neg(A_0.B_0.\neg A_1.\neg B_1).\neg(A_1.A_0.B_1.B_0).\neg(\neg(A_0.B_0).\neg(A_1.B_1).\neg(\neg A_1.\neg B_1)))$$

Figure 38: 2-bit adder, max fan-in 4

Used:	[11, 3, 2, 1]
Out of:	[16, 64, 48, 1]
Max fan-in:	2
Average fan-in:	2

$$\begin{aligned} S_1: \quad & \lnot(\lnot(\lnot(\lnot(\lnot A_1.B_1).\lnot(A_0.B_0)).\lnot(\lnot(A_1.B_1).\lnot(\lnot A_1.\lnot B_1))) \\ & \quad .\lnot(\lnot(A_0.B_0).\lnot(\lnot(A_1.\lnot B_1).\lnot(\lnot A_1.B_1)))) \end{aligned}$$

Figure 39: 2-bit adder, max fan-in 2

Used:	[15, 4, 1, 1]
Out of:	[24, 256, 192, 1]
Max fan-in:	4
Average fan-in:	2.6

$$\begin{aligned} S_2: \quad & \lnot(\lnot(\lnot(A_2.B_2).\lnot(\lnot A_2.\lnot B_2).\lnot(A_1.B_1).\lnot(A_0.B_0)) \\ & \quad .\lnot(\lnot A_1.\lnot B_1.\lnot(A_2.B_2).\lnot(\lnot A_2.\lnot B_2)) \\ & \quad .\lnot(\lnot(\lnot A_1.\lnot B_1).\lnot(\lnot(A_2.B_2).\lnot(\lnot A_2.\lnot B_2)).\lnot(\lnot(A_1.B_1).\lnot(A_0.B_0)))) \end{aligned}$$

Figure 40: 3-bit adder, max fan-in 4

Used:	[18, 5, 3, 1]
Out of:	[24, 256, 192, 1]
Max fan-in:	3
Average fan-in:	2.36

$$\begin{aligned} S_2: \quad & \lnot(\lnot(\lnot(\lnot(A_2.B_2).\lnot(\lnot A_2.\lnot B_2)).\lnot(\lnot(A_1.B_1).\lnot(A_0.B_0)).\lnot(\lnot B_1.\lnot(A_1.\lnot B_1))) \\ & \quad .\lnot(\lnot(A_0.B_0).\lnot(B_1.\lnot(\lnot A_1.B_1)).\lnot(\lnot(A_2.\lnot B_2).\lnot(\lnot A_2.B_2))) \\ & \quad .\lnot(\lnot A_1.\lnot B_1.\lnot(\lnot(A_2.\lnot B_2).\lnot(\lnot A_2.B_2)))) \end{aligned}$$

Figure 41: 3-bit adder, max fan-in 3

Used: [26, 9, 3, 1]
Out of: [32, 256, 192, 1]
Max fan-in: 4
Average fan-in: 2.75

$$\begin{aligned}
S_3\text{:}\quad & \lnot(\lnot(\lnot(\lnot A_1.\lnot B_1.\lnot(A_2.B_2)) \\
& \qquad .\lnot(\lnot(A_3.B_3).\lnot(\lnot A_3.\lnot B_3)) \\
& \qquad .\lnot(\lnot(A_2.B_2).\lnot(A_1.B_1).\lnot(A_0.B_0)) \\
& \qquad .\lnot(\lnot A_2.\lnot B_2.\lnot(\lnot A_2.B_2))) \\
& \quad .\lnot(\lnot B_2.\lnot\lnot(A_2.B_2)) \\
& \quad .\lnot(\lnot(\lnot(A_3.\lnot B_3).\lnot(\lnot A_3.B_3).\lnot(A_2.B_2)) \\
& \qquad .\lnot(A_2.B_2.\lnot(A_3.B_3)) \\
& \qquad .\lnot(B_0.\lnot(\lnot A_2.\lnot B_2).\lnot(\lnot A_1.\lnot B_1).\lnot(\lnot A_0.B_0)) \\
& \qquad .\lnot(A_1.B_1.\lnot(\lnot A_2.\lnot B_2))))
\end{aligned}$$

Figure 42: S_3

Used: [33, 13, 4, 1]
Out of: [40, 256, 192, 1]
Max fan-in: 5
Average fan-in: 3.14

$$\begin{aligned}
S_4\text{:}\quad & \lnot(\lnot(\lnot A_2.\lnot B_3.\lnot B_2.\lnot(A_4.B_4).\lnot(\lnot A_4.\lnot B_4)) \\
& \quad .\lnot(\lnot(\lnot(A_4.\lnot B_4).\lnot(\lnot A_4.B_4).\lnot(A_3.B_3).\lnot(A_2.B_2)) \\
& \qquad .\lnot(\lnot(A_4.B_4).\lnot(\lnot A_4.\lnot B_4).\lnot(\lnot A_3.\lnot B_3)) \\
& \qquad .\lnot(\lnot A_3.\lnot B_3.\lnot(A_4.\lnot B_4).\lnot(\lnot A_4.B_4))) \\
& \quad .\lnot(\lnot(\lnot(A_3.\lnot B_3).\lnot(\lnot A_3.B_3)) \\
& \qquad .\lnot(\lnot(A_4.B_4).\lnot(\lnot A_4.\lnot B_4)) \\
& \qquad .\lnot(\lnot(A_1.B_1).\lnot(A_0.B_0)) \\
& \qquad .\lnot(\lnot(A_2.\lnot B_2).\lnot(\lnot A_2.B_2)) \\
& \qquad .\lnot(\lnot B_1.\lnot(A_1.\lnot B_1))) \\
& \quad .\lnot(B_3.\lnot A_3.\lnot B_2.\lnot(A_2.\lnot(\lnot A_4.\lnot B_4)) \\
& \qquad .\lnot(\lnot(A_4.\lnot B_4).\lnot(\lnot A_4.B_4))) \\
& \quad .\lnot(\lnot(A_1.B_1).\lnot(A_2.\lnot(A_2.\lnot B_2)) \\
& \qquad .\lnot(\lnot(A_3.\lnot B_3).\lnot(\lnot A_3.B_3)) \\
& \qquad .\lnot(\lnot(A_4.\lnot B_4).\lnot(\lnot A_4.B_4).\lnot(A_3.B_3).\lnot(A_2.B_2)) \\
& \qquad .\lnot(A_0.B_0.\lnot(A_4.B_4).\lnot(\lnot A_4.\lnot B_4).\lnot(\lnot A_1.\lnot B_1))))
\end{aligned}$$

Figure 43: S_4

Used:	[58, 35, 8, 1]
Out of:	[64, 256, 192, 1]
Max fan-in:	8
Average fan-in:	4.04

S_4: $\neg(\neg(\neg B_5.\neg(A_6.B_6).\neg\neg(\neg A_3.B_3).\neg\neg(A_7.\neg B_7).\neg(A_5.\neg(\neg A_4.\neg B_4)).$
$\quad\neg\neg(A_3.\neg B_3).$
$\quad\neg\neg(A_5.\neg B_5))$
$\quad.\neg(\neg A_1.\neg\neg(A_7.\neg B_7).\neg\neg(A_0.\neg B_0).\neg\neg(A_1.B_1).\neg\neg(A_2.\neg B_2))$
$\quad.\neg(B_1.\neg B_6.\neg B_3.\neg\neg(\neg A_7.B_7).\neg\neg(A_6.B_6))$
$\quad.\neg(\neg(\neg(A_7.B_7).\neg(\neg A_7.\neg B_7).\neg(\neg A_6.\neg B_6).\neg(\neg A_5.\neg B_5))$
$\qquad.\neg(\neg A_6.\neg B_6.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(\neg A_6.B_6))$
$\qquad.\neg(\neg A_5.\neg B_5.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_5.\neg B_5))$
$\qquad.\neg(\neg A_3.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(\neg A_3.B_3))$
$\qquad.\neg(\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).$
$\qquad\quad\neg(A_4.B_4).\neg(A_3.B_3).\neg(A_2.B_2).\neg(A_1.B_1))$
$\qquad.\neg(\neg B_2.\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).$
$\qquad\quad\neg(A_4.B_4).\neg(A_3.B_3).\neg(A_2.\neg B_2).\neg(\neg A_2.B_2))$
$\qquad.\neg(\neg A_4.\neg B_4.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(A_4.\neg B_4))$
$\qquad.\neg(A_6.B_6.\neg(A_7.B_7).\neg(\neg A_7.\neg B_7).\neg(A_6.\neg B_6)))$
$\quad.\neg(\neg(A_5.B_5).\neg(A_6.\neg(A_7.B_7).\neg(A_6.\neg B_6))$
$\qquad.\neg(A_0.B_0.\neg(\neg A_5.\neg B_5).\neg(\neg A_4.\neg B_4).\neg(\neg A_3.\neg B_3).\neg(\neg A_2.\neg B_2).\neg(\neg A_1.\neg B_1))$
$\qquad.\neg(A_1.\neg(\neg A_5.\neg B_5).\neg(\neg A_4.\neg B_4).\neg(\neg A_3.\neg B_3).\neg(\neg A_2.\neg B_2).\neg(A_1.\neg B_1))$
$\qquad.\neg(A_2.B_2.\neg(\neg A_5.\neg B_5).\neg(\neg A_4.\neg B_4).$
$\qquad\quad\neg(\neg A_3.\neg B_3).\neg(A_2.\neg B_2).\neg(\neg A_2.B_2).\neg(\neg A_2.\neg B_2))$
$\qquad.\neg(\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6))$
$\qquad.\neg(A_3.B_3.\neg(\neg A_5.\neg B_5).\neg(\neg A_4.\neg B_4).\neg(\neg A_3.B_3))$
$\qquad.\neg(A_4.B_4.\neg(\neg A_5.\neg B_5).\neg(A_4.\neg B_4)))$
$\quad.\neg(\neg(\neg A_6.\neg B_6.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(\neg A_6.B_6))$
$\qquad.\neg(A_0.\neg A_1.\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(A_2.B_2).\neg(\neg A_1.B_1).\neg(A_0.\neg B_0))$
$\qquad.\neg(\neg A_5.\neg B_5.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_5.\neg B_5))$
$\qquad.\neg(\neg(A_7.B_7).\neg(\neg A_7.\neg B_7).\neg(\neg A_6.\neg B_6))$
$\qquad.\neg(\neg A_3.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(\neg A_3.B_3))$
$\qquad.\neg(\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).$
$\qquad\quad\neg(A_4.B_4).\neg(A_3.B_3).\neg(A_2.B_2).\neg(A_1.B_1))$
$\qquad.\neg(\neg A_4.\neg B_4.\neg(A_7.\neg B_7).\neg(\neg A_7.B_7).\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(A_4.\neg B_4))$
$\qquad.\neg(\neg A_2.\neg(A_6.B_6).\neg(A_5.B_5).\neg(A_4.B_4).\neg(A_3.B_3).\neg(\neg A_2.B_2)))$
$\quad.\neg(\neg\neg(\neg A_6.B_6).\neg\neg(\neg A_2.\neg B_2).\neg\neg(A_4.\neg B_4).\neg(A_7.\neg(A_3.B_3)))$
$\quad.\neg(\neg\neg(A_5.\neg B_5).\neg\neg(\neg A_5.\neg B_5).\neg(\neg A_3.\neg(\neg A_7.B_7))$
$\qquad.\neg\neg(A_0.\neg B_0).\neg\neg(A_2.\neg B_2).\neg\neg(\neg A_1.B_1)))$

Figure 44: S_7

Used:	[40, 12, 5, 1]
Out of:	[64, 256, 192, 1]
Max fan-in:	6
Average fan-in:	3.36

C_7:

$$\begin{aligned}
&\lnot(\lnot(\lnot(\lnot A_7.\lnot B_7).\lnot(\lnot A_6.\lnot B_6)\\
&\qquad\quad .\lnot(\lnot A_5.\lnot B_5.\lnot(A_7.B_7).\lnot(A_6.B_6).\lnot(A_5.B_5).\lnot(\lnot A_5.B_5))\\
&\qquad .\lnot(\lnot(A_7.B_7).\lnot(A_6.B_6).\lnot(A_5.B_5)))\\
&\quad .\lnot(\lnot(\lnot A_7.\lnot B_7).\lnot\lnot(A_7.B_7).\lnot(\lnot B_7.\lnot(A_7.B_7)).\lnot(\lnot(A_7.B_7).\lnot(A_6.B_6)))\\
&\quad .\lnot(\lnot(\lnot A_7.\lnot B_7).\lnot(\lnot A_4.\lnot B_4)\\
&\qquad\quad .\lnot(\lnot A_5.\lnot B_5.\lnot(A_7.B_7).\lnot(A_6.B_6).\lnot(A_5.B_5).\lnot(\lnot A_5.B_5))\\
&\qquad .\lnot(\lnot(A_7.B_7).\lnot(A_6.B_6).\lnot(A_5.B_5).\lnot(A_4.B_4))\\
&\qquad .\lnot(\lnot A_7.\lnot(A_7.B_7).\lnot(A_7.\lnot B_7)).\lnot(\lnot B_6.\lnot(A_7.B_7).\lnot(A_6.B_6)))\\
&\quad .\lnot(\lnot(\lnot A_6.\lnot B_6).\lnot(\lnot A_5.\lnot B_5.\lnot(A_7.B_7).\lnot(A_6.B_6).\lnot(A_5.B_5).\lnot(\lnot A_5.B_5))\\
&\qquad .\lnot(\lnot A_3.\lnot B_3.\lnot(A_5.B_5).\lnot(A_4.B_4).\lnot(A_3.B_3).\lnot(A_3.\lnot B_3))\\
&\qquad .\lnot(\lnot A_7.\lnot B_7.\lnot(A_7.B_7).\lnot(\lnot A_7.B_7))\\
&\qquad .\lnot(\lnot(A_5.B_5).\lnot(A_4.B_4).\lnot(A_3.B_3).\lnot(A_2.B_2).\lnot(A_1.B_1))\\
&\qquad .\lnot(\lnot A_4.\lnot(A_5.B_5).\lnot(A_4.B_4).\lnot(A_4.\lnot B_4).\lnot(\lnot A_4.B_4)))\\
&\quad .\lnot(A_7.B_7.\lnot(\lnot A_7.\lnot B_7).\lnot\lnot(A_7.B_7).\lnot(\lnot A_7.\lnot(A_7.B_7).\lnot(A_7.\lnot B_7))))
\end{aligned}$$

Figure 45: C_7

NAND networks

The candidate

1 Project Motivation

Current methods for synthesising a circuit to match a truth table include Karnaugh maps, applying Boolean algebra laws, or a brute force search of the exponential space, with some optimisations that mean in practice, it doesn't take exponential time. The proposal is to apply machine learning (ML) to this field. As a result we gain the well-known benefits, such as being able to use gradient descent to efficiently search this exponential space, and learn patterns in the truth table.

There is also an interesting positive feedback loop here. With ML on the rise, the main limitations in training new neural networks (NNs) are data, and compute. With the size of GPUs growing, if NNs can effectively used in computer architecture design, this can further accelerate the development of NNs.

Another advantage comes from a shift in perspective. Instead of thinking about this as using ML to find a circuit, we can think about it as using a circuit to represent a NN. A field of growing importance in ML is network compression. Networks are growing in size and number of parameters, which enable them to learn more complex patterns. But for sustainability, cost, and inference time, there are many incentives to make NNs smaller. Using a logical circuit to represent a NN would drastically improve all three. This is because we're replacing matrix multiplications with NAND gates. Not only does it improve these 3 factors, but doing so also enables us to deploy these NNs on devices with less compute power available, e.g. phones.

2 Planned Approach

The plan is to replace classical neurons with "continuous" NAND gates, which due to their functional completeness, can represent any logical circuit. What's being trained is the connections between the NAND gates, these are our parameters. Our loss function would also include a term which pushes these weights towards being connected or unconnected, to ensure we end up with a valid, discrete logical circuit. The weights for the connections would have the range $[-\infty, \infty]$, and then I'd put it through a sigmoid, so 0 means disconnected, and 1 means connected. And so for a NAND gate with k inputs, x_1 to x_k, the actual inputs would be $f(x_i, w_i)$. So y, which represents the output of our NAND gate, would be modelled as such:

$$y = 1 - \prod_{i=1}^{k} f(x_i, w_i) \tag{1}$$

$$f(x_i, w_i) = x_i^{\sigma(w_i)} \tag{2}$$

If $\sigma(w_i) = 0$, $f(x_i, w_i) = 1$, and if $\sigma(w_i) = 1$, $f(x_i, w_i) = x_i$. This makes sense since for a NAND gate, we'd want unconnected to be 1, since that means it would have no effect on the output. And to push the connections to be connected or unconnected, we'd add the term $1 - \sigma(|\overrightarrow{w}|)$ to the loss function. This means the loss is minimised when the weights are extreme, which corresponds to the circuit being discrete. The exact maths could change, these functions could be replaced with any continuous functions that satisfy the $f(x, 0) = 1, f(x, 1) = x$, and adding a term to the loss function which is minimized when the weights correspond to a discrete circuit.

The core of the project which should be possible yet challenging, would be to develop a system to learn a boolean logic circuit. Possible extensions would include:

- Testing if the system could learn equations inductively. For example, a 4-bit adder has be 28 = 256 possible inputs. The test would be seeing if it could learn the right circuit with less than 256 distinct pieces of training data, and if it's unable to do this, investigating why, or seeing if I can adjust things to enable it to do tasks like this.
- Adjusting the loss function, or hyperparameters to optimize for different things. For example, a hyperparameter which specifies the max number of inputs for each NAND gate, the number of NAND gate layers allowed, the number of NAND gates allowed in each layer. Or alternatively, the loss function punishing NAND gates with lots of connections, or lots of NAND gates. These factors can affect inference time (the more layers the higher the circuit delay), and also circuit cost and power usage.
- Testing if the system can successfully be used for tasks that classical NNs are commonly used for. For example, classifying numbers on the MNIST dataset. So if we have x classes, we'd have xk NAND gates in the final layer, so we have xk outputs, in x groups. The class of the output would be the argmax of these neurons [1]. So for example, k = 10 and x=2. If class 1 has 1/10 outputs which are 1s, and class 2 has 7/10, it'd be classified as a 2.
- The description so far is analogous to an NN only using dense layers. So another potential extension would be exploring what convolutional, attention or transformer architectures could look like with this model. A potential implementation of this could be by allowing outputs to connect to inputs. This would enable flip-flops to be learnt, giving the system memory. A good test case for this extension could be the ability to learn some finite state machine (FSM). This is certainly the most ambitious extension.

3 Starting Point

I have discussed the project with my supervisor, who gave me some inspiration for the maths outlined above, and also suggested this project. I have also carried out a literature review, which also gave me some inspiration, specifically [1]. Other than this research, no work has been done on the project. There's no framework set up for the project.

4 Project Structure

I plan to use JAX, which is a module in Python. This module encourages a functional programming approach to NNs, and so the structure of the project would be defining all of the components: initialisation, the NAND gates themselves, the optimizer, and the loss function. We'd finally tie all of these components together with JAX. Furthermore, just as important as the architecture is the data. For the core of the project, the data may be quite simple to input, as it's just truth tables. For some of the extensions described above, this may become more complex. I may also use optax, which would particularly help for adjustable learning rates.

- Initialisation – We want to start with randomised weights. However, there's a certain asymmetry with NAND gates, since we only need one 0 to ensure that the output is 1, whereas we need all the inputs to be 1 to ensure that the output is 0. And so one module would be to find an initialisation strategy which means that at every layer, there's a 50
- The NAND gates and loss function – the maths for these has been described above, there's also room to explore different implementations satisfying those equations, to potentially improve performance.
- The optimizer – different optimizers have different assumptions, so the particular optimizer used may depend on if this is part of the core, or an extension. Regardless, there's room to explore different options to potentially improve performance.

- Post-training optimization – wouldn't be a part of the main JAX framework, but once a circuit has been learnt, there are typical optimisations we can do. For example, removing any NAND gates whose outputs aren't connected to anything.
- Data processing – especially for some extensions, this would come before training the network, and would include separating out data from training and testing.

5 Success Criteria

5.1 Core Criteria

- Create a system that can learn a circuit to compute any boolean operation with up to 4 inputs (any 16 row truth table)
- Create a system that can learn a circuit to compute more complex boolean operations (e.g. an 8-bit adder)

5.2 Extension Criteria

- Create a system that can inductively learn circuits with patterns (e.g. learn an 8-bit adder with only 192 items of training data instead of 256)
- The system can create circuits that optimise for desirable factors. Including minimising: delay, number of NAND gates, max number of inputs into each NAND gate and power usage. For learning circuits inductively, we can investigate the trade-off between accuracy and training time.
- The system can be repurposed to be used for classical ML tasks, such as image classification.
- (Most ambitious) the system can learn an FSM.

6 Project Timetable

Goal for work package	**Deliverable**	**Deadline**
Initial Implementation	Some non-trivial learned circuit	01-Nov
Exploring Alternatives	Data on training times and accuracies with different implementations	15-Nov
Optimising	Successful post-training optimisation	29-Nov
Initialisation	Data on training times and accuracies with different initialisations	22-Jan
Progress Report	Progress Report	07-Feb
Extensions	Some interface for specifying what to optimise for, and the system produces circuits optimised accordingly	21-Feb
Dissertation	First draft	07-Mar
Extensions	Data on accuracy for image classification, or a system that can learn an FSM	21-Mar
Dissertation	Second draft, updated based on feedback	04-Apr
Extensions	Data on accuracy for image classification, or a system that can learn an FSM	18-Apr
Dissertation	Final draft	02-May
Contingency	Contingency	16-May

7 Resource Declaration

I will be using my own laptop (HP Omen 16, 16GB RAM) for my project. My contingency plans include backing up to Google Drive and GitHub. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. If my laptop isn't powerful enough, especially for the extensions, I may also use a server, which Robert Mullins' could grant me access to.

8 References

[1] Felix Petersen, Christian Borgelt, Hilde Kuehne, Oliver Deussen, "Deep Differentiable Logic Gate Networks ". `https://arxiv.org/pdf/2210.08277`