Automatic Generation of Neural Architecture Search Spaces

David Calhas, Vasco Manquinho, Inês Lynce

INESC-ID, Instituto Superior Técnico, Portugal
david.calhas@tecnico.ulisboa.pt

Abstract

Neural Architecture Search (NAS) is receiving growing attention as the need to remove the human bias from neural network models rises. There is extensive research in trying to beat state-of-the-art NAS algorithms. However, these advances do not focus directly on the search space these algorithms explore. Here, we propose a framework that encodes the structure of a convolutional neural network, respecting the arithmetical relation of the kernel and stride sizes with the input and output shapes. This framework consists of a formula with constraints that, if given the structure of the problem (input and output shapes), can produce specification properties of a neural architecture through a solver. We show that this methodology can assemble networks with arbitrary sizes and structures, that make for unique and uniform search spaces. To compare the resulting architectures, a metric that computes dissimilarity in terms of architectural structure is proposed. We empirically show that generating dissimilar architectures, implies dissimilarities in performance. In accordance, similar architectures are similar in performance.

1 Introduction

Recently, there have been advances in NAS (Liu et al. 2018; Zoph and Le 2016; Li and Talwalkar 2020; Ying et al. 2019), where neural architectures (NAs) are either scrapped from previous works (Kandasamy et al. 2018) or manually built by the user (Elsken et al. 2019), forming a NAS search space, $\mathcal{A}$, that is explored to find the best architecture. However, the liability of these approaches is that the architectures, which assemble the NAS space, are limited and biased. In this work, NAs are generated by an algorithm, capable of producing a non biased and uniform architecture space, as it is built automatically by an algorithm as opposed to a human. This is achieved using constraint programming, more specifically, with Satisfiable Modulo Theory (SMT) and Boolean Satisfiability techniques.

The main idea of this work is to take advantage of the arithmetic of convolutional layers (Dumoulin and Visin 2016) and use the kernel, k, and stride, s, for the relation

$$O = \frac{I-k}{s} + 1 \Leftrightarrow I = (O-1) \times s + k, \qquad (1)$$

Copyright © 2022, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

to generate NAs. Consider the following setup: the input shape, I, and the output shape, O, are known. And one wants to build a function that maps I to O. One way of solving this is to simply apply an affine transformation. However, there are cases where one does not want to do the latter, be it because of: memory limitations (the memory cost of a convolution is $\mathcal{O}(\text{channels} \times k)$, and an affine transformation $\mathcal{O}(I \times O)$); interpretability of latent activations (Zhang, Wu, and Zhu 2018); or preservation of the input representation. The other way, is to use an SMT solver to retrieve solutions for k and s, which represent a convolutional NA, as illustrated in Figure 1.

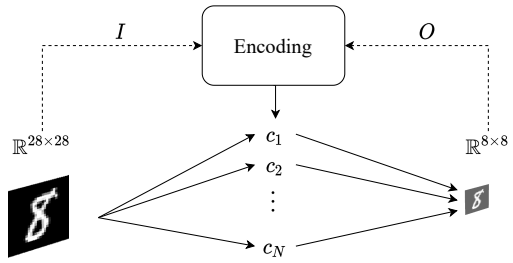


Figure 1: An encoding, of NAs, given an input, I, and an output, O, provides a solution set, $\mathcal{S}$, of NAs. For the example illustrated, $I = 28 \times 28$ and $O = 8 \times 8$, all solutions, c_i, are in accordance with $c_i : \mathbb{R}^{28\times28} \rightarrow \mathbb{R}^{8\times8}$.

The contributions of this work are:

1. show that the kernel and stride sizes of convolutional neural networks (CNNs) can be integrated in a formula, relating them with the input and output shapes, used to sample NAs with an arbitrary depth of downsampling layers (see Section 3);
2. integration of a family of hash functions (based on XOR constraints, see Section 4), shown by Chakraborty, Meel, and Vardi (2013) to sample diverse solutions from a formula. This results in an NA space with uniform and unique samples (see Section 6.2);
3. show that the proposed framework not only is capable of generating CNNs, but also able to mutate already existing architectures that are assembled with downsampling blocks, e.g., Resnet-18 (He et al. 2016) (see Section 5.2);

4. show that the structural dissimilarity of the generated NAs increases the exploratory factor of a NAS algorithm (see Section 6.3) in the solution space of the proposed formula, consequently increasing the likelihood of finding a good architecture.

The methodology proposed is integrated with the DARTS algorithm (Liu, Simonyan, and Yang 2018), showing that the framework can be easily integrated in a NAS setting. In accordance with (Elsken et al. 2019), we focus on exploring the space, $\mathcal{A}$, a NAS algorithm works on.

2 Problem Description

Consider an input space, $\mathbb{R}^I$, output space, $\mathbb{R}^O$, and an instance $\vec{x} \in \mathbb{R}^I$. I and O are K-Dimensional spaces, that is $I = I^{(1)} \times \cdots \times I^{(K)}$. Similarly $O = O^{(1)} \times \cdots \times O^{(K)}$, with $K \in \mathbb{N}\backslash\{0\}$. The k-th dimension of I is referred to as $I^{(k)}$. A convolutional layer, L_l, is seen as a mapping function, f_{L_l}, between two spaces. A CNN, L, is a set of N layers, $L = \{L_1, \ldots, L_L\}$, which is also seen as a mapping function, f_L. The latter is a chain of mapping functions, $f_L : \mathbb{R}^I \to \mathbb{R}^O, f_L = f_{L_N}(\ldots f_{L_1}(\vec{x})\ldots) \in \mathbb{R}^O$. Now let I and O be part of a formula F with $|V|$ variables, being V the set of variables, and $\mathcal{S}$ the set of N assignments to V that satisfy F. Each element of $\mathcal{S} = \{c_1, \ldots, c_N\}$ is a function such that $\forall i \in \{1, \ldots, N\}, c_i : \mathbb{R}^I \to \mathbb{R}^O$.

3 Problem Formalization

Convolutional operations can be applied in 1-Dimensional (e.g. signal), 2-Dimensional (e.g. images), 3-Dimensional (e.g. videos) spaces and so on. For the sake of simplicity, consider two 1-Dimensional spaces, $\mathbb{R}^I$ and $\mathbb{R}^O$, which are referred to as Input Space and Output Space, respectively.

The input and output of a convolutional layer[1] are related with the kernel size, k, and stride, s, respectively, obeying Equation 1 $\forall I, O \in \mathbb{N} : I \geq O$ iff $k > 0 \wedge s > 0$. A CNN in its simplest form[2] can be seen as a set of $N \in \mathbb{N}$ convolutional layers, $L = \{L_1, \ldots, L_N\}$. Every layer is characterised by its kernel size and stride,

$$\forall n \in \{1, \ldots, N\} : L_n = (k_n, s_n).$$

A CNN transformation from an input space, I, to an output space, O, is represented as: $f_L : \mathbb{R}^I \mapsto \mathbb{R}^O$. Similarly, a convolutional layer, $L_i \in L$, transformation is defined as $f_{L_i} : \mathbb{R}^{I_i} \mapsto \mathbb{R}^{O_i}$, with $I = I_1 \leq \ldots I_i \leq O_i \cdots \leq O_N = O$. Consider the special case of $L_0 : \vec{x} = f_{L_0}(\vec{x})$ being a layer that represents the input, I. Then with a setup formulated with $f_{L_0}(.) \in \mathbb{R}^I \wedge \cdots \wedge f_{L_i}(.) \in \mathbb{R}_i^O \wedge \cdots \wedge f_{L_N}(.) \in \mathbb{R}^O$ and $\forall i \in \{0, \ldots, N-1\} : O_i = I_{i+1}$, this can be represented as a SMT formula,

$$(O_N = O) \wedge \bigwedge_{l=1}^{N} (O_l \leq O_{l-1} \wedge k_l > 0 \wedge s_l > 0).$$

An SMT solver gives us a set of N tuples of (k, s) characterizing convolutional layers by solving:

$$\left(\frac{I_N - k_N}{s_N} + 1 = O\right)$$
$$\wedge$$
$$\bigwedge_{l=1}^{N} \left(\frac{I_l - k_l}{s_l} + 1 \leq O_{l-1} \wedge k_l > 0 \wedge s_l > 0\right).$$

K-Dimensional Setting - to extend the problem to K dimensions one just needs to apply the previous equation to all K dimensions. With $x^{(k)}$ corresponding to the k-th dimension of an instance x in a K-Dimensional space, such a setting is represented as

$$r = k_l^{(k)} > 0 \wedge s_l^{(k)} > 0,$$
$$H_o = \left(\frac{I_N^{(k)} - k_N^{(k)}}{s_N^{(k)}} + 1 = O^{(k)}\right),$$
$$H_h = \left(\frac{I_l^{(k)} - k_l^{(k)}}{s_l^{(k)}} + 1 \leq O_{l-1}^{(k)} \wedge r\right),$$
$$\bigwedge_{k=1}^{K} H_o \wedge \bigwedge_{l=1}^{N} H_h.$$

H_o refers to an output layer and H_h refers to a hidden layer. An SMT solver would explore a space of infinite satisfiable solutions, since $(k, s) = (1, 1)$ characterises a convolutional layer that does not mutate the dimensions from the input to the output. To avoid this, the number of layers to be accounted for is lower, n, and upper, N, bounded.

Pseudo boolean optimization - to enumerate solutions that satisfy an SMT formula with lower and upper bounds for the number of layers, one needs to have auxiliary variables that define which constraints are participating, i.e. which lth kernel and stride are eligible for the solution. For that pseudo-Boolean (Roussel and Manquinho 2009) variables, X, are introduced, where $X = \{x_1, \ldots, x_N, x_{N+1}\}$. All $x \in X$ should obey $\forall 2 \leq i \leq N : x_i = 1 \Rightarrow x_{i-1} = 1$ and $\forall 1 \leq i \leq N-1 : x_i = 0 \Rightarrow x_{i+1} = 0$. The first means if $x_i = 1$ then all the previous layers are activated, therefore it implies $x_{i-1} = 1$ and the same goes for x_{i-1} until x_1. For the second statement, if $x_i = 0$, meaning the ith layer is not activated and does not participate in the solution, then the same goes for x_{i+1} which is 0, until x_N. In addition, $x_{N+1} = 0$. Every time one wants a solution of l layers, a constraint, c, should be true, where $c = \{\sum_{i=1}^{N} x_i = l\}$ equivalent to $(\forall 1 \leq i \leq l : x_i = 1) \wedge (\forall l < i \leq N : x_i = 0)$ or simply $x_l = 1 \wedge x_{l+1} = 0$. With this and considering $H_{o_x} = (\neg x_n \vee x_{n+1} \vee H_o)$ and $H_{h_x} = (\neg x_n \vee \neg x_{n+1} \vee H_h)$, F is defined as

[1] Convolutions have more parameters that are not described in this document, such as number of channels and padding. These parameters are not considered in this description, because they are not being used in the context of this work.

[2] In the machine learning community CNN studies also call networks with more types of layers, in addition to convolutions, CNNs (e.g. networks with convolutional layers and fully connected layers).

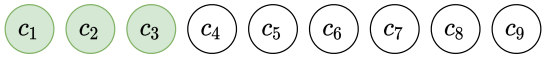
$$\forall n = 1, \ldots, N : \bigwedge_{k=1}^{K} \left[H_{o_x} \wedge H_{h_x} \right].$$

Solution enumeration - to enumerate all solutions that satisfy the SMT formula, one needs to specify the SMT solver to not give us a specific solution again. This is done by adding the solution to the SMT formula in the form of negation. If, for a given x_i, the formula is no longer satisfiable, then $x_{i+1} = 1$, while $i < N$. All the solutions are enumerated if $i = N$ and the formula is no longer satisfiable.

4 The Uniform Solution Enumeration Problem

The goal is to generate a limited number of reasonably different architectures. However, F, solved by a well known Solver (Z3-Python (de Moura and Bjørner 2008)), follows a bias that makes consecutively enumerated solutions (CES) very similar. For instance, if $n = 2$ and $N = 5$, the enumerated solutions will have the following order: first all solutions of 2 layers will be enumerated, then all solutions of 3 layers will be enumerated, so on and so forth. In addition, F has a exponential growing number of solutions (see Figure 4) making $\mathcal{A}$ (recall $\mathcal{A}$ as the set of NAs explored in a NAS algorithm) very large.

A workaround is to limit the solutions sampled using the natural enumeration bias of a solver. This would cause $\mathcal{A}$ to lack uniformity, which in consequence makes the algorithm that explores $\mathcal{A}$ to have a small exploration trait in the solution space, $\mathcal{S}$, of F. Please note that $\mathcal{S}$ is the set of all the solutions of F, whereas, $\mathcal{A}$ is a subset of $\mathcal{S}$, where the NAS algorithm operates. For instance, consider a solution space $S = \bigcup_{i=1}^{9} c_i$, where c_i, c_{i+1} are highly similar, whereas c_i, c_j, with $j \gg i$, having a higher degree of difference. To enumerate 3 solutions from $\mathcal{S}$, we define two approaches: a *biased* approach, that once a solver finds the first solution, it performs atomic changes to a minimum set of variables to give a different solution, this is the behaviour described in Figure 2a; and a *near-uniform* approach, that uses techniques to uniformly sample solutions from a solution space, as described in Figure 2b. In the rest of this Section, we describe the technique that is able to do the latter.



(a) A distribution of 3 solutions among a solution space, following the *biased* enumeration mechanism of a SAT solver.

(b) A *near-uniform* distribution of 3 solutions among a solution space.

Figure 2: Comparison between sampling from a solution space according to a biased versus uniformly sampling.

Chakraborty, Meel, and Vardi (2013) proposed the UniWit algorithm that is able to enumerate uniformly distant solutions. This algorithm uses XOR constraints to restrict the solution space, $\mathcal{S}$, of a formula, F, and randomly chooses a solution on the restricted solution space. The XOR constraints enable the sampling of solutions with a near uniform distribution among $\mathcal{S}$ (Gomes, Sabharwal, and Selman 2007). UniWit starts by defining a pivot variable, $p = 2|V|^{\frac{1}{k}}$, with V being the set of variables in F and k a constant, recommended by the original work to be set to $k = 2$. It then iteratively envolves more variables in XOR constraints that all together make an hash function,

$$h = \bigwedge_{j=1}^{i-l} \left(\left(\bigoplus_{h=1}^{n} (V[j] \wedge a[j+h-1]) \oplus b[j] \right) \Leftrightarrow \alpha[j] \right).$$

The variables involved in this equation, $\alpha, a, b \in \{0, 1\}$, are uniformly generated with length $i - l$, $|V| + i - l - 1$, $i - l$, respectively. In addition, auxiliary variables, l, i, are initialized with $\frac{1}{k} log_2 |V|$ and $l - 1$, respectively.

UniWit iteratively increments i and generates an hash function, h, until the formula $F \wedge h$ has a solution space, $\mathcal{S}$, with total number of solutions lesser than p, $|\mathcal{S}| < p$. When $|\mathcal{S}| < p \wedge |S| > 0$ the algorithm stops and returns a random choice from $\mathcal{S}$. If $|\mathcal{S}| < 1$ the algorithm returns a null solution. This corresponds to sampling one solution from F, therefore to sample M *near-uniform* solutions from F this process is repeated at most M times (UniWit may fail and return a null solution). For more details please refer to the original work (Chakraborty, Meel, and Vardi 2013). Due to the solving time bottleneck for the sampling of each solution, the total time to gather M solutions increases, as shown in Section 6.4.

5 Evaluation

In Section 6, we provide an experimental comparison on three methods:

- *All*: Solutions are enumerated until F is unsat, using the internal enumeration bias of Z3-Solver.
- *Limited-M*: Solutions enumerated, using the internal enumeration bias of Z3-Solver. Solutions are enumerated until, either M solutions are enumerated or the formula, F, becomes unsat.
- *Limited_Uniform-M*: Solutions are enumerated until, either M solutions are enumerated or $F \wedge h$ becomes unsat (recall h the hash function defined in Section 4). The order in which solutions are enumerated, ends up following a uniform solution picking among the solution space of F as described in Section 4.

In this section, two evaluation methodologies are introduced. One assesses the dissimilarity between CES. The other evaluates the generated search space using a state-of-the-art NAS algorithm (Liu, Simonyan, and Yang 2018).

5.1 Uniformity evaluation metric

In order to evaluate how similar are CESs, a bit-wise logical xor is used,

$$\text{adj_}T = \frac{1}{M-1} \sum_{m=0}^{M-1} \left[\sum c_m \oplus c_{m+1} \right]. \quad (2)$$

The term $c_m \oplus c_{m+1}$ refers to a bit-wise logical xor. The sum of all positions of the bit-array, $\sum c_m \oplus c_{m+1}$, is the manhattan distance between two vectors, $c_m, c_{m+1} \in \{0, 1\}$. To assess the similarity of CES, the mean distance of all CES (c_m, c_{m+1}) is taken.

5.2 Automatic generation based on Resnet-18

Resnet-18 (He et al. 2016) is used to address the quality of the generated architectures from both limited approaches. It is an attractive and well known architecture, and it does not consume as much computational power as its bigger versions. The way this architecture was used for automatic generation is the following: by analyzing the mutation of shapes from layer to layer, one can observe that it goes from shape 8×8 to the final shape 1×1 (specific for input with shape 28×28), being mutated by a total of 3 times corresponding to $8 \times 8 \rightarrow 4 \times 4 \rightarrow 2 \times 2 \rightarrow 1 \times 1$. Resnet-18 has a total of 18 layers, however the shape is only mutated in 3 of them. As such, we specify, in F (recall from Section 3), $n = 3 \wedge N = 3 \wedge I = 8 \times 8 \wedge O = 1 \times 1$. Given a limited number of solutions, the kernel and stride values $\forall l \in [0, N] : k_l, s_l$ are used to obtain a modification of the Resnet-18.

The changes of the residual block are shown in Figure 3. The input, x, is split in two flows, with one being processed by a convolution with $k = 1 \wedge s = 2$ followed by a convolution with $k = 3 \wedge s = 1$ that keeps the dimensions of the input (*same* padding scheme (Dumoulin and Visin 2016)) and the other only being processed by a convolution with $k = 1 \wedge s = 2$. Each convolution is followed by either a *bn* layer (stands for Batch Normalization (Ioffe and Szegedy 2015)) or *relu* activation (Rectified Linear Unit (Nair and Hinton 2010)) or both, with *relu* always following *bn* and the latter the convolution operation. In the end, both flows are joined in an addition operation followed by a *relu* activation.

5.3 Assessing the quality of the generated NA space

In addition to comparing the final test accuracy of each generated network and the original Resnet, it is pertinent to see how they evolve along the training session. As such, the method proposed in (Liu, Simonyan, and Yang 2018) is used. It consists on treating all of the NAs as a single model that outputs a prediction. Let:

- c_0 be the original Resnet-18;
- $\forall i \in \{1, \ldots, M\} : c_i$ be the set of M generated NAs by one of the limited approaches (*Limited-M* or *Limited_Uniform-M*);
- $\alpha \in \mathbb{R}^{M+1}$ the weights attributed to the NAs predictions.

Then as defined in (Liu, Simonyan, and Yang 2018),

$$\hat{y} = \sum_{i=0}^{M} \frac{e^{\alpha_i}}{\sum_{j=0}^{M} e^{\alpha_j}} c_i(x), \quad (3)$$

given an image, $x \in \mathbb{R}^{L \times W}$, with label $y \in \{0, 1\}^C$, being L the height of the image, W the width and C the number of classes of the classification problem, an NA performs a mapping $c_i : \mathbb{R}^{L \times W} \rightarrow \mathbb{R}^C$, whose output is $c_i(x)$. With such a setting, Equation 3 can be interpreted as the sum of the softmax normalized α_i multiplied with $c_i(x)$. The softmax activation provides a probability value $\frac{e^{\alpha_i}}{\sum_{j=0}^{M} e^{\alpha_j}} \in [0, 1]$ that can be interpreted by how much importance the $c_i(x)$ prediction has for the final $\hat{y}$. By optimizing α with a gradient descent method: NAs, with poor predictions, are assigned probability $\frac{e^{\alpha_i}}{\sum_{j=0}^{M} e^{\alpha_j}} \rightarrow 0$, whereas the best NAs have probability $\frac{e^{\alpha_i}}{\sum_{j=0}^{M} e^{\alpha_j}} \rightarrow 1$. The best NA is derived from α, by taking the argmaxα at the end of the training session.

Due to limited GPU availability, we only take zero order gradients, i.e. given a loss function, $\mathcal{L} : [\mathbb{R}^C, \mathbb{R}^C] \rightarrow \mathbb{R}$, α is optimized with respect to $c_i(x)$ by taking gradients $\nabla_\alpha \mathcal{L}(y, \hat{y})$ and the weights, w_i, of each NA, c_i, are optimized with respect to the input, x, with gradients $\nabla_{w_i} \mathcal{L}(y, c_i(x))$. The chosen loss function, $\mathcal{L}$, is the negative log likelihood.

It is worth noting that, in contrast with (Liu, Simonyan, and Yang 2018), we are performing NAS in a space that has different geometric architectures, instead of different local operations (e.g. check which operation between max pooling and average pooling is best). Please refer to the original work for more details (Liu, Simonyan, and Yang 2018).

6 Results

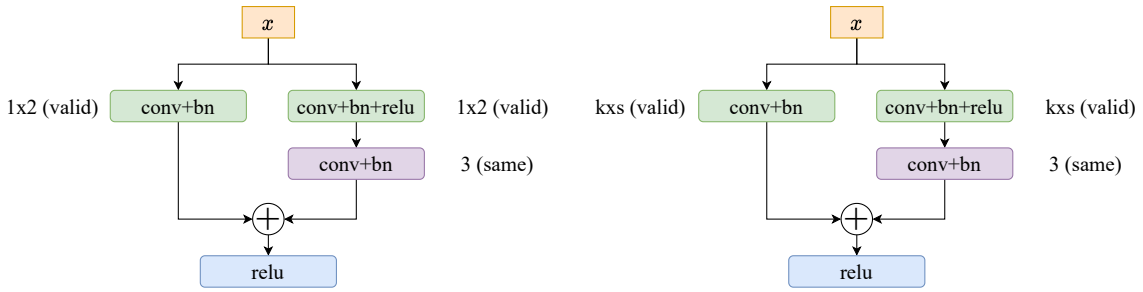
In this Section, we provide the results regarding the following experiments: the number of solutions of the formula defined in Section 3; quality of the CES of *Limited-M* and *Limited_Uniform-M* according to the metric described in Equation 2; a performance comparison between *Limited-M* and *Limited_Uniform-M* in a classification task setting; and an analysis of the time performance related to the approaches defined in the beginning of Section 5.

6.1 Number solutions

In Figure 4, one can see how the number of solutions increases with the difference between the input and output, $I - O$. The generated NAs setup a search space for a NAS algorithm, and as with any algorithm, the bigger the search space the harder it is to converge, especially when evaluating a state takes a long time, which is the case in NAS. As a consequence, the full solution space, $\mathcal{S}$, can not be fully explored, but a subset, $\mathcal{A}$, with M solutions can.

6.2 Quality

Adding to the need of selecting a limited number of solutions from F, the chosen subset of solutions, $\mathcal{A}$, should be representative of $\mathcal{S}$. In this Section, we show the results on two approaches that select a limited number of solutions: *Limited-M* and *Limited_Uniform-M*. The first follows the standard bias of the Z3-Solver and the second uses XOR constraints to promote a uniform sampling of solutions. We



(a) Original Resnet block.

(b) Redefined block to fit our framework.

Figure 3: Comparison between the original Resnet block and the redefined block. All of the architectures submitted to evaluation have the redefined block of Figure 3b. It is considered as the original Resnet, a neural network that integrates the redefined block with $k = 1 \wedge s = 2$ in the downsampling blocks.

Table 1: Adj_T metric for both limited approaches. Entries are formatted as *Limited-M*/*Limited_Uniform-M* for a better comparison. For all settings, *Limited_Uniform-M* had the bigger Adj_T, producing a uniform like CES.

Number Layers	1D	2D	3D
2	3.375/**4.125**	3.895/**7.632**	5.684/**10.789**
3	3.579/**5.000**	3.632/**10.053**	3.211/**13.842**

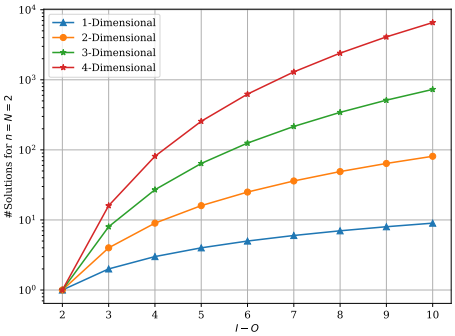


Figure 4: Number Solutions *All*. $I - O$ refers to the difference between the input and output. All input, I, dimensions had value of 30, however the exact value of I does not impact the number of solutions, but the $I - O$ does.

compare the two with the evaluation metric described in Equation 2, that is capable of evaluating how distant are CES. Results are reported in Table 1.

Results were gathered for 1D, 2D and 3D dimensional settings with $I = 30$ and $O = 20$. The number of layers considered was $N = \{2, 3\}$. More settings were not considered as the average sampling time increased, for a higher number of layers and dimensions, becoming unfeasible to run in real time, as shown in Section 6.4. In Table 2 is shown the same comparison of Table 1, but with the addition of pooling layers after each convolution, i.e. after each layer, a layer with $k = 2$ and $s = 1$ is added, emulating a pooling layer. This setting is widely used in computer vision and inserting already setup layers in the middle of the convolutional layers, decreased significantly the solving time. This made feasible in real time the generation of neural networks with as much as 5 convolutional layers, each one of them followed by a pooling layer (giving a total of 10 layers).

We observe that *Limited-M* had a slighty bigger distance of CES than *Limited_Uniform-M* in the 1D case for 2 and 3 layers. In the other cases, 2D and 3D for 2, 3 and 4 layers the *Limited_Uniform-M* showed that it promotes CES with a higher degree of dissimilarity than *Limited-M*.

6.3 Classification using Resnet as a generation baseline

To address the quality of the generated architectures, the setup, introduced in Section 5.2, is used to evaluate the algorithm with the MNIST (LeCun and Cortes 2010) dataset. The experiments were ran with $M = 20$. All networks, including the weights, α, were trained with a learning rate of 0.0001, batch size of 512, 0.0001 weight decay and trained for 10 epochs. Recall that the zero order gradient propagation, defined by Liu, Simonyan, and Yang (2018), was used for optimization. Table 3 shows the results gathered from the experiments.

The original Resnet version achieved an accuracy of 0.9843. In comparison, the generated architectures, using the *Limited_Uniform-M* approach, achieved an average accuracy of 0.9856 with 0.0028 standard deviation. The best and worst architecture had accuracies 0.9903 and 0.9801, respectively, meaning a $+0.0060$ and -0.0042 difference against the original Resnet. As for the *Limited-M* approach, an average accuracy of 0.9865 with 0.0019 standard deviation. The best and worst architecture achieved 0.9909 and 0.9835, respectively, with a $+0.0086$ and $+0.0012$ difference against the original Resnet. *Limited_Uniform-M* had higher accuracy standard deviation than *Limited-M*, with 0.0028 against 0.0019. The latter indicates uniformity in performance, however the differences between *Lim-*

Table 2: Adj_T metric for both limited approaches, with pooling layers following each convolutional layer. Entries are formatted as *Limited-M*/*Limited_Uniform-M* for a better comparison. For 5 layers the formula only has one solution, $|\mathcal{S}| = 1$, consequently Adj_$T = 0$.

Number Layers	1D	2D	3D
2	**4.286**/3.286	3.750/**7.150**	4.600/**10.350**
3	3.933/**4.267**	3.500/**7.750**	4.450/**11.400**
4	**3.600**/3.200	3.800/**6.000**	5.400/**9.300**

Table 3: Comparison of the results of Resnet-18, *Limited_Uniform-M* and *Limited-M*.

Number Layers	Dataset	Accuracy	Best	Worst	α deviation
Resnet	MNIST	0.9843	NA	NA	NA
Limited_Uniform-M	MNIST	0.9856	0.9903	**0.9801**	**0.0028**
Limited-M	MNIST	**0.9865**	**0.9909**	0.9835	0.0019

ited_Uniform-M and *Limited-M*, at least in accuracy, are not statistically significant.

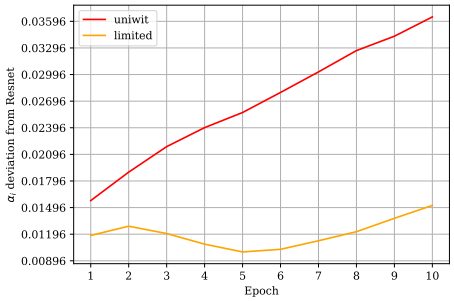


Figure 5: α_i deviation from the original Resnet-18. The plot shows that the *Limited_Uniform-M* generated architectures increase deviation with the epochs, which is not seen with the *Limited-M* architectures that stay with the same deviation against the Resnet-18.

Figure 5 shows the weight deviation along the epochs, during the training session. The weight deviation against the Resnet is defined as $\sqrt{\frac{\sum_{i=1}^{M}(\alpha_i-\alpha_0)^2}{M-1}}$, where α_0 is the weight related to the original Resnet-18 and $\forall i \in \{1,\ldots,M\} : \alpha_i$, the weights associated to the generated architectures. The *Limited_Uniform-M* deviation strictly increased along the epochs, whereas *Limited-M* stayed constant.

6.4 Time

Table 4 reports the total time it took to gather $M = 20$ solutions, with $I = 30$, $O = 20$ and pooling layers (similar to Table 2) for the *Limited_Uniform-M* amd *Limited-M* approaches.

Limited-M is much faster than *Limited_Uniform-M*, which was expected due to the solving time bottleneck for each solution sampled, whereas *Limited-M* performs atomic changes at the bottom leafs of the search tree to provide the next solution.

Solving time refers to the time spent solving a formula, F, i.e. the time the solver takes until returning the first solution that satisfies F. As seen in Figure 6, the solving time increases exponentially with the increase of dimensions.

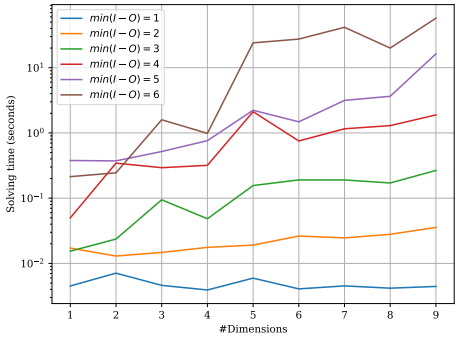


Figure 6: Solving time *All*

The higher the difference between the input and output space, the more complex the problem gets and therefore the solving time increases as well. In Figure 6, the times refer to *All*, however since all approaches solve the same formula, the solving time is the same for *Limited-M* and a sub-estimate for *Limited_Uniform-M* (the hash function introduced increases the solving time).

Finally, we analyze the *average sampling time* of the *Limited_Uniform-M* approach, which is shown in Table 5. The setting is the same that was used for Table 2.

The results show that *Limited_Uniform-M* is not able to generate deep neural networks (networks with a high number of layers) that have a high gap between the input and output space, $I - O$. However, a neural network does not have to be deep to have a good performance, in fact (Guo et al. 2019) shows that compressing neural networks not only increases the computation time, but in some cases promotes better efficacy.

7 Related Work

There are various approaches the research community has done in NAS. In this section, a description of some of the most recent works is given. Sukthanker et al. (2020) define a

Table 4: Total time (seconds) to gather $M = 20$ solutions for both limited approaches, with pooling layers following each convolutional layer. Entries are formatted as *Limited-M*/*Limited_Uniform-M* for a better comparison.

Number Layers	1D	2D	3D
2	**0.262**/0.789	**0.676**/6.293	**0.983**/13.676
3	**0.611**/5.418	**1.821**/99.775	**1.645**/1204.123
4	**1.731**/6.888	**17.338**/508.337	**394.587**/11509.547
5	123.670/**35.412**	497.815/**14.912**	1885.051/**66.540**

Table 5: Average sampling time (seconds) of the *Limited_Uniform-M* approach.

Number Layers	1D	2D	3D
2	0.0493	0.255	0.619
3	0.288	4.9219	60.128
4	0.606	25.344	575.392
5	34.252	5.526	50.457

continuous space of semi positive definite matrices, where a neural network layer/operation is represented as a point. The defined space respects defined rules in order to associate a layer to it. The chosen operation is retrieved by discretizing (taking the argmin of all operations) to the closest (defined distance) defined operation in that space. All candidate architectures are optimized using the relaxation of the softmax (Liu, Simonyan, and Yang 2018). Zhao et al. (2021) explore the use of supernets in NAS, which consists on using a neural architecture that encompasses a representation of all networks in a search space. By picking multiple supernets from a search space, one can apply a search algorithm on this subset of architectures, that automatically takes less time than exploring the entire space. Although, the methodology reduces by many orders the search, there is still a bottleneck to where/how are the architectures from/generated[3]. Nayman et al. (2021) perform search in a space configured with different convolutions, i.e., different kernels (strides are fixed). However, they do not explore all the combinations, which in consequence do not account with different dimension exploration (always apply a square shaped convolution). In terms of search algorithm, they propose a space of one hot variables, that once optimized, the optimal architecture is drawn by taking the argmax. The optimization is done using gradient descent methods on the relaxed the one hot variables, that in consequence act as probability distributions. At each step of the optimization procedure, an architecture is sampled from the distributions. The objective of Nayman et al. (2021) was to restrict the resource usage, which was done in accordance with the convolution properties and depth of the network. Kandasamy et al. (2018) approach the problem of NAS using Bayesian optimization (Snoek, Larochelle, and Adams 2012). This is feasible through the definition of an acquisition function. This function chooses the network that is most similar to the best network acquired, until a certain timestep of the optimization process. The similarity is computed through a distance proposed in (Kandasamy et al. 2018). Grathwohl et al. (2018) much like Liu, Simonyan, and Yang (2018), explore the concept of NAS with gradient based optimization. Our work differs from these, since they do not incorporate the notion of different convolutional structures (kernel and stride size).

[3]The NAS-Bench-101 was used as the neural architecture search space.

8 Conclusion

In this work, SAT and SMT techniques were used to produce a framework capable of generating NAs automatically, specifically chained neural architectures. However, it was shown one can also use it to adapt already existing networks, as was done with the Resnet-18 architecture. The dimensionality of the solution space and the enumeration bias present in the Z3-Solver restrict the use cases of the encoding formula. To tackle it, XOR constraints, that are known to restrict a solution space and perform a near uniform solution sampling, were used. In the results section, it was shown that the framework is capable of generating NAs that are relatively distant from each other, in other words a sparse and near uniform set of NAs. In terms of quality, the *Limited_Uniform-M* space of architectures was evaluated in the MNIST dataset and it showed uniformity in terms of accuracy when compared with the *Limited-M* approach. Consequently, the uniform sampling of architectures from the solution space, produces networks that are uniform in terms of performance.

9 Reproducibility

Since the evaluation process introduced in Section 5.2 is very exhaustive, the results presented in this work are not able to claim statistical significance, as only one run was made. To reproduce them please use the Tensorflow library (Abadi et al. 2016) and set the seed to 42, as was done in our experiments. A public Github repository[4] is available with the source code of the authors. We used a NVIDIA GeForce RTX 208 GPU (8GB) and the *Limited_Uniform-20* MNIST experiments took ≈45 days.

[4]**https://github.com/DCalhas/auto-nas-space**

10 Acknowledgments

This work was supported by national funds through Fundação para a Ciência e Tecnologia (FCT), under the Ph.D. Grant SFRH/BD/5762/2020 to David Calhas, ILU project DSAIPA/DS/0111/2018 and INESC-ID pluriannual UIDB/50021/2020. We want to give special thanks to Prof. Rui Henriques for the great supervision guidance given to David Calhas. This project was done in the context of Boolean Constraints and Optimization Course @ IST-UL DEIC Programme. We also want to thank Pedro Orvalho and Margarida Ferreira for the input to this work.

References

Abadi, M.; Barham, P.; Chen, J.; Chen, Z.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; Irving, G.; Isard, M.; et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, 265–283.

Chakraborty, S.; Meel, K. S.; and Vardi, M. Y. 2013. A scalable and nearly uniform generator of SAT witnesses. In *International Conference on Computer Aided Verification*, 608–623. Springer.

de Moura, L. M.; and Bjørner, N. 2008. Z3: An Efficient SMT Solver. In Ramakrishnan, C. R.; and Rehof, J., eds., *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, 337–340. Springer.

Dumoulin, V.; and Visin, F. 2016. A guide to convolution arithmetic for deep learning. *arXiv preprint arXiv:1603.07285*.

Elsken, T.; Metzen, J. H.; Hutter, F.; et al. 2019. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55): 1–21.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2007. Near-uniform sampling of combinatorial spaces using XOR constraints. In *Advances In Neural Information Processing Systems*, 481–488.

Grathwohl, W.; Creager, E.; Ghasemipour, S. K. S.; and Zemel, R. 2018. Gradient-based optimization of neural network architecture.

Guo, Y.; Zheng, Y.; Tan, M.; Chen, Q.; Chen, J.; Zhao, P.; and Huang, J. 2019. Nat: Neural architecture transformer for accurate and compact architectures. *arXiv preprint arXiv:1910.14488*.

He, K.; Zhang, X.; Ren, S.; and Sun, J. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision andpattern recognition*, 770–778.

Ioffe, S.; and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, 448–456. PMLR.

Kandasamy, K.; Neiswanger, W.; Schneider, J.; Poczos, B.; and Xing, E. P. 2018. Neural Architecture Search with Bayesian Optimisation and Optimal Transport. In Bengio, S.; Wallach, H.; Larochelle, H.; Grauman, K.; Cesa-Bianchi, N.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 31, 2016–2025. Curran Associates, Inc.

LeCun, Y.; and Cortes, C. 2010. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/.

Li, L.; and Talwalkar, A. 2020. Random search and reproducibility for neural architecture search. In *Uncertainty in Artificial Intelligence*, 367–377. PMLR.

Liu, C.; Zoph, B.; Neumann, M.; Shlens, J.; Hua, W.; Li, L.-J.; Fei-Fei, L.; Yuille, A.; Huang, J.; and Murphy, K. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, 19–34.

Liu, H.; Simonyan, K.; and Yang, Y. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055*.

Nair, V.; and Hinton, G. E. 2010. Rectified linear units improve restricted boltzmann machines. In *Icml*.

Nayman, N.; Aflalo, Y.; Noy, A.; and Zelnik-Manor, L. 2021. HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. *arXiv preprint arXiv:2102.11646*.

Roussel, O.; and Manquinho, V. M. 2009. Pseudo-Boolean and Cardinality Constraints. *Handbook of satisfiability*, 185: 695–733.

Snoek, J.; Larochelle, H.; and Adams, R. P. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25: 2951–2959.

Sukthanker, R. S.; Huang, Z.; Kumar, S.; Endsjo, E. G.; Wu, Y.; and Van Gool, L. 2020. Neural Architecture Search of SPD Manifold Networks. *arXiv preprint arXiv:2010.14535*.

Ying, C.; Klein, A.; Christiansen, E.; Real, E.; Murphy, K.; and Hutter, F. 2019. Nas-bench-101: Towards reproducible neural architecture search. In *International Conference on Machine Learning*, 7105–7114. PMLR.

Zhang, Q.; Wu, Y. N.; and Zhu, S.-C. 2018. Interpretable convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 8827–8836.

Zhao, Y.; Wang, L.; Tian, Y.; Fonseca, R.; and Guo, T. 2021. Few-shot neural architecture search. In *International Conference on Machine Learning*, 12707–12718. PMLR.

Zoph, B.; and Le, Q. V. 2016. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*.