Designing resource-constrained neural networks using neural architecture search targeting embedded devices

Available online at www.sciencedirect.com

ScienceDirect

IoT

ELSEVIER

Internet of Things; Engineering Cyber Physical Human Systems 00 (2020) 1–15

Designing Resource-Constrained Neural Networks Using Neural Architecture Search Targeting Embedded Devices

Thomas Cassimon, Simon Vanneste, Stig Bosmans, Siegfried Mercelis, Peter Hellinckx

University of Antwerp - imec, IDLab - Faculty of Applied Engineering, Sint-Pietersvliet 7, 2000 Antwerp, Belgium

Abstract

Recent advances in the field of Neural Architecture Search (NAS) have made it possible to develop state-of-the-art deep learning systems without requiring extensive human expertise and hyperparameter tuning. In most previous research, little concern was given to the resources required to run the generated systems. In this paper, we present an improvement on a recent NAS method, Efficient Neural Architecture Search (ENAS). We adapt ENAS to not only take into account the network's performance, but also various constraints that would allow these networks to be ported to embedded devices. Our results show ENAS' ability to comply with these added constraints. In order to show the efficacy of our system, we demonstrate it by designing a Recurrent Neural Network that predicts words as they are spoken, and meets the constraints set out for operation on an embedded device, along with a Convolutional Neural Network, capable of classifying 32x32 RGB images at a rate of 1 FPS on an embedded device.

Keywords: Neural Architecture Search, Resource Constraint, Embedded Device, Neural Network, Internet of Things

1. Introduction

In recent years, developing state-of-the-art neural networks has become a challenge, due to the vast complexity of these systems. Developing neural networks usually requires a substantial amount of experimentation and hyperparameter tuning, as well as domain knowledge and expertise in designing neural networks. This is a lengthy and tedious process due to the sheer size of the hyperparameter and architectural space. In order to mitigate this problem, the idea of Neural Architecture Search (NAS) [1] was introduced. In NAS, a controller is trained using a Reinforcement Learning (RL) algorithm, REINFORCE [2] in our case. The controller first generates a neural network. This network is then trained and evaluated based on its performance. The controller then uses the networks' performance to learn to generate better networks. This search process still takes a long time to converge, however. In order to remedy this, Pham et al. tested the idea of weight sharing [3]. Instead of training a network from scratch every time, weights are shared between all architectures, allowing them to converge faster. While this was a major improvement on NAS, there are still some unaddressed issues. Something that has been overlooked in most research are the resource requirements of a NAS system. Most papers just focus on optimizing the generated networks' performance [1] [3].
The resource requirements of neural networks are set to become equally important however, given the prevalence of mobile and edge devices in modern Internet of Things (IoT) networks. Some research has gone into improving networks resource consumption, showing promising results [4] [5].

Email addresses: thomas.cassimon@uantwerpen.be (Thomas Cassimon), simon.vanneste@uantwerpen.be (Simon Vanneste), stig.bosmans@uantwerpen.be (Stig Bosmans), siegfried.mercelis@uantwerpen.be (Siegfried Mercelis), peter.hellinckx@uantwerpen.be (Peter Hellinckx)

In this paper we will attempt to address some of these resource constraints by introducing hard and soft constraints on our architectures. Our research combines the short search times achieved by Pham et al. [3] with a multi-objective approach, allowing us to quickly find cells that also meet a given set of constraints. We first discuss the current state-of-the-art in NAS (Section 2), then we give an overview of the techniques we employed to improve on the current state-of-the-art (Section 3), we explain our experiments (Section 4) and also present our results (Section 5). Finally we give a brief summary of our results (Section 6) and determine a possible direction for future research (Section 7).

2. State-of-the-art

The state-of-the-art in NAS can be split into three categories: RL based approaches [1] [3] [6], Evolutionary approaches [4] [7] and Bayesian approaches [8]. For our research, we opted for a RL-based approach because of the short search times that recent RL-based NAS methods can achieve. Evolutionary approaches have the advantage of being much more straightforward than RL, requiring less parameter tuning, but they also take significantly more time to find solutions. Bayesian optimization techniques on the other hand, are more sample-efficient than RL-based approaches, allowing for even shorter search times, the problem with Bayesion Optimization methods, is that the existing toolboxes typically focus on continuous problems with low dimensionality [9].

In their initial paper, Zoph & Le [1] described a way to use a reinforcement learning controller to generate Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs). Both of these systems performed similar to state-of-the-art, human-designed, systems, while requiring little to no human interaction. In order to train their controller, they used REINFORCE, a policy gradient algorithm [2]. In an attempt to limit the amount of time needed to find competitive CNN architectures, NAS was adapted to search for cells instead of complete architectures [6], similar to human-designed architectures such as ResNet [10] and InceptionNet [11]. These cells can then be stacked to form a complete neural network. This way of designing systems allows for a minimal amount of performance tuning by increasing or reducing the number of cells.

Later, further improvements were made on this work by introducing weight sharing [3]. Weight sharing involves forcing all models to share a single set of trainable parameters. Using the idea of weight sharing in Efficient Neural Architecture Search (ENAS), Pham et al. were able to significantly reduce the amount of computational power required to traverse the search space. Results show that ENAS is capable of finding competitive cells in less than a day on a single Graphics Processing Unit (GPU), compared to 32 400 - 43 200 GPU hours for earlier NAS algorithms [6]. Recently, the work of Pham et al. has been drawn into question, with researchers investigating the correlation between the accuracy of architectures trained using weight-sharing, and that of architectures trained from scratch. [12] Yu et al. concluded that weight sharing has a negative impact on the results of NAS algorithms, and suggest the development of improved weight sharing techniques. [12] Other researchers suggest using different evaluation metrics for weight-sharing networks, such as the sparse Kendall-Tau correlation coefficient. [13]. Integrating the methods proposed in [13] into our work is unfortunately not feasible, since determining the sparse Kendall-Tau coefficient requires us to know the ground-truth ranking of the architectures we are evaluating. Some of our architectures do not fit into the search spaces discussed in [13], making it impossible to determine the sparse Kendall-Tau coefficient.

Recently, research has shown that NAS isn't limited to optimizing for a single objective. In their paper Elsken et al. [4] used Lamarckian evolution to find architectures that are not only comparable to state-of-the-art systems in terms of performance, but also offer a significant reduction in the amount of parameters the models require. LEMONADE, the algorithm by Elsken et al. [4], is able to find a large set of architectures in about 80 GPU-days. This set of architectures forms a Pareto-front, where all solutions are Pareto-optimal. A solution to a multi-objective problem is considered Pareto-optimal when it becomes impossible to improve one objective without harming the others [14]. While the time necessary to run LEMONADE is significantly more than ENAS, at the end of the experiment, approximately 300 architectures are left, all of which are Pareto-optimal, allowing the user to choose an architecture to meet their needs. Contrary to our work, LEMONADE doesn't require trade-offs between objectives to be defined beforehand, this is useful because it lets the evolutionary process discover the most optimal solutions, and allows the user to make trade-offs when all viable architectures are known.

3. Methods

For our research we intend to improve on ENAS by introducing extra constraints to the reward function of the controller. We will design these constraints to maximize the generated networks' performance, while keeping resource requirements low. In order to make our system sufficiently flexible, we decided to split our constraints into two categories. The first is a set of hard constraints, and the second is a set of soft constraints. If the system fails to meet one of the hard constraints, its reward will be zero. If the hard constraints are met, the system will be rewarded with the sum of its soft constraints.
In order to implement these constraints, we use Equation 1 to determine the reward, R given to our controller during its training.

$$R(\mu, C_h, C_s) = (C_{h,0} \cdot C_{h,1} \cdot \ldots \cdot C_{h,k-1}) \cdot \left(\sum_{i=0}^{n-1} \mu_i \cdot C_{s,i} \right) \quad (1)$$

In this equation, we have k hard constraints ($C_{h,0}$ through $C_{h,k-1}$) and n soft constraints ($C_{s,0}$ through $C_{s,n-1}$). Each soft-constraint is weighted using a pre-set weight (μ_i for the i-th constraint).

3.1. Hard Constraints

The first hard constraint we used is the amount of memory the model uses. Determining the memory usage requires us to traverse the cell's graph, and determine the size of the block's inputs based on the size of the outputs leading into this block. We don't just track the number of parameters, but we also consider the size of these parameters. We chose to take the parameter size into account, because this allows us to compare this to the available memory of real platforms, and check the feasibility of running our networks on embedded devices. We consider this constraint violated if the networks' memory usage exceeds a predetermined maximum. When setting the maximum, a certain amount of extra memory should be assumed for things that aren't part of our cells, but are still necessary, like encoders and decoders in RNNs. This constraint also does not account for run-time allocated memory for things like intermediate results, since this amount of memory is very difficult to predict, we decided not to take it into account for our research. Finally, it is important to mention that the amount of memory required for an entire model, is not the same as the amount of memory required for a single cell. We now formalize our hard memory constraints as:

$$C_{h,memory} = \begin{cases} 0 & \text{if } S_{model} + \epsilon \geq S_{device} \\ 1 & \text{if } S_{model} + \epsilon < S_{device} \end{cases} \quad (2)$$

In Equation 2 S_{model} is the size of our model, S_{device} is the amount of available memory on the target device, and ϵ is a small amount of memory to be used for other purposes beside our model. When determining the memory requirements for our CNNs, we include the memory for pointwise convolutions used to ensure all dimensions match. The second hard constraint we introduced is inference latency. Inference latency is measured using a cost model that contains the approximate latency for every available activation function and some basic operations. By accumulating the computational costs of every node in our cell, we can obtain an estimate of the amount of latency induced when making a single pass through our cell. If this latency exceeds a pre-determined maximum, we consider the constraint to be violated. This constraint can be formalized as:

$$C_{h,complexity} = \begin{cases} 0 & \text{if } Latency_{model} + \epsilon \geq Latency_{max} \\ 1 & \text{if } Latency_{model} + \epsilon < Latency_{max} \end{cases} \quad (3)$$

Just as in the previous case, a small ϵ is added to our latency to make sure the system has some spare time for computations that aren't part of our model. Analogous to the memory constraint, when building CNN cells, we include the time necessary to perform pointwise convolutions to prevent dimension mismatches.

3.2. Soft Constraints

While our design allows for multiple soft constraints, we found that not all soft constraints are useful. In this section, we will discuss the different soft constraints we considered, and our reasoning on why we did or did not include them in our final system.

3.2.1. Compression Estimation

We considered adding a soft constraint that estimates how much a generated network can be compressed. In order to achieve this, we evaluated three different methods: Shannon's source coding theorem, Wavelet compression and Bloomier Filters.

In 1948 Claude Shannon introduced his source coding theorem [15]. Shannon's theorem calculated the maximum amount a piece of information could be compressed in a lossless fashion. This theoretical limit [16] can be calculated as shown in equation 4 for an N-point signal $\vec{f}$:

$$S_{min} = N \cdot H\left(\vec{f}\right) \tag{4}$$

The problem with this approach is that it forces us to quantize our data. This means that we need access to that data, which is not available during the architecture search process. On top of that, the resulting limit is quite sensitive to the exact quantization, which can be especially troublesome when quantizing 32-bit floating point numbers. Since this assumes a lossless compression, the maximum obtainable compression ratio will be significantly lower than that of lossy algorithms. These problems make this a generally undesirable approach.

Wavelet transformations are also an effective way of compressing data. While wavelets have various uses, some are specifically designed for compression, such as those used in the JPEG-2000 [17] standard. The problem with wavelet transformations is that they do not allow us to simply estimate the compression ratio without executing the actual compression. Since some of our models can get quite large (several million parameters), executing a wavelet compression every time a reward needs to be calculated is not feasible without dramatically increasing our the search-time. Similar to Shannon's theorem, this also requires us to have access to the trained weights while performing architecture search, which is not possible in our current framework.

Bloomier Filters are a probabilistic data structure [18] that allow for high data compression ratios. Because of the inherent presence of the possibility of false-positives, we consider them to be a lossy compression technique. Using Bloomier Filters, Reagen et al. [19] were able to achieve compression ratios of up to 496x. While they are an excellent way to compress a deep learning system, their compression ratio is determined by the design parameters, and not by the data, thus making it impossible for our agent to perform specific optimizations that improve compressibility.

After evaluating these methods, we decided that compression estimation is a relatively difficult problem to solve, while also yielding a limited amount of extra information to our controller, which lead us to the decision to exclude the compression constraint from our research.

3.2.2. Cache Constraint

The second soft constraint we considered, is the cache constraint, this constraint was introduced because modern processors spend a large portion of their time waiting for data to be fetched from memory, thus a program that is able to place more of its data in the processor's cache will usually be faster. Another reason why caches are important in embedded systems, is energy use. According to Horowitz [20], the cost of an off-chip Dynamic, Random-Access Memory (DRAM) access (1-2nJ) is a couple of orders-of-magnitude larger than the cost of an internal cache memory access (10pJ). In order to encourage cache use, we constructed a constraint that penalizes our agent for networks that do not fit into cache, and rewards the agent for networks that fit in the cache, shown in Equation 5. The constraint also accounts for the various cache levels, since caches closer to the processor tend to take fewer cycles to access. It is important to note that this constraint is a guideline and not an exact performance measure. When evaluating our cell, other data will be present in the system's cache memory and reduce our ability to utilize the system's cache to the fullest.

$$\begin{aligned} C_{s,cache} = {} & \mu_{L1} \cdot (S_{\text{L1 Cache}} - S_{model}) \\ & + \mu_{L2} \cdot (S_{\text{L1 Cache}} + S_{\text{L2 Cache}} - S_{model}) \\ & + \mu_{L3} \cdot (S_{\text{L1 Cache}} + S_{\text{L2 Cache}} + S_{\text{L3 Cache}} - S_{model}) \end{aligned} \tag{5}$$

In our experiments, we used $\frac{1}{3}$ for μ_{L1}, $\frac{1}{2}$ for μ_{L2} and 1 for μ_{L3}. This choice is arbitrary and not representative of the cost of cache misses. Our choice of parameters gives a smaller penalty for models that don't fit into L1 cache, a slightly larger penalty for models that don't fit into the combination of L1 and L2 cache and an even larger penalty for

models that don't fit into the combination of L1, L2 and L3 caches. Models are penalized less, but also rewarded less, for exceeding the size of small caches. The main reason for these decisions is, that it is unlikely that our system will be able to fit a model into a L1 cache, due to their limited size (typically measured in kilobytes). In order to get more accurate estimates of caching behaviour, more advanced models of cache memories are required, which we consider out-of-scope for this research.

3.2.3. Network Performance

The final constraint for our system is the performance of our cells on a validation data set. The definition of this constraint differs between RNNs and CNNs. For RNNs, we calculate our performance reward as described in Equation 6. When designing our reward function, we noticed that Pham et al. don't mention the value of c used for their experiments. We decided to set c to 80, which is the same value Zoph & Le [1] used.

$$C_{s,performance} = \frac{c}{(ppl_{valid})^2} \tag{6}$$

When searching for CNNs, we use the top-1 accuracy of our image classifier on a batch of 128 images.

3.3. Dynamic Cell Size

Initial experiments show that ENAS is able to meet hard constraints, given a static cell size. However, our controller is still limited in how small it can make the networks it produces, since the number of blocks in a cell is fixed. In order to give the controller greater freedom in choosing the design of its cells, we allowed the controller to sample the number of blocks in a cell. This allows the controller to change the size of its cells in a significantly larger range than before. In order to generate a dynamically-sized cell, the controller first samples the number of blocks in a cell, after which the generation process proceeds normally. When sampling CNN cells, we first sample the size of the convolution and reduction cell, and then continue the normal generation process.

4. Experiments

In this section, we will discuss the experiments we conducted. First, we briefly explain the timing models we used for execution time estimation of our networks in section 4.1. Next, we explain the idea behind our RNN experiments in section 4.2.1 and provide an overview of the parameters used in these experiments (section 4.2.2). After the experiments on RNNs, we discuss the selected use-case for our CNN experiments in section 4.3.1 and give an overview of the relevant hyperparameters in section 4.3.2.

4.1. Execution Time Analysis

In order to estimate the latency of activation functions and basic operations, we used a C++ program that performed a number of experiments. Each of these experiments executes the given activation function a certain number of times. In the context of RNNs, we conducted 1000 experiments, where every experiment consisted of applying the activation function to 100 000 elements. For CNNs, we performed 10 000 experiments, and each experiment executed the activation function on a 1 x 3 x 32 x 32 feature map. If the operation generated an output with different dimensions, the output had a dimension of 1 x 16 x 32 x 32. The cost model we obtained for our device is shown in Table 1. Durations reported are for a single experiment (100 000 elements for RNN functions, 1 x 16 x 32 x 32 elements for CNN functions). We stress that these are estimates and not guarantees, the analysis of software execution time is a complex problem that requires advanced models to solve [21] [22] [23], and is considered out-of-scope for our research.

4.2. Recurrent Neural Networks

4.2.1. Use Case

In order to demonstrate our algorithm, we set our constraints based on a real-world use case. Our model needs to be able to predict words at the same pace as they would be spoken by a native speaker, in English. When giving a presentation, English native speakers tend to speak at a pace of about 100-125 words per minute, we will round this to 120 words per minute [24], resulting in an even 2 words per second. We also want to allow some extra time for

RNN Function	Mean (μs)	Std. Dev (μs)	CNN Function	Mean (μs)	Std. Dev (μs)
Identity	409.6	32.368	Identity	8.106	0.5570
ReLU	650.0	51.956	ReLU	18.442	1.3749
Addition	799.8	79.392	Addition	25.688	0.5359
Multiplication	785.6	78.867	Concatenation	26.057	0.8917
Division	1623.4	122.576	BatchNorm 2D	112.166	2.2824
Sigmoid	12 986.4	109.108	Average Pooling (3x3)	290.894	3.7392
Tanh	17 301.9	240.072	Max Pooling (3x3)	292.378	2.8640
			Pointwise Convolution	328.942	4.3596
			Softmax	354.612	3.9878
			Depthwise Separable Convolution (3x3)	763.006	63.2884
			Depthwise Separable Convolution (5x5)	1265.742	8.7211

Table 1: Latency statistics for executing operations on a Raspberry Pi 3B

other systems such as encoders and decoders and miscellaneous tasks, which results in a maximum inference time of 330ms. Since we also want our solution to be portable, we need to run it on an embedded platform. For this, we chose the Raspberry Pi 3B. The Raspberry Pi has 1GB of RAM memory, of which we will use half, leaving 500MB for the operating system and miscellaneous memory consumption.

4.2.2. Configuration

We organize our results based on whether or not our constraints were enabled. When constraints are disabled, we only take a cell's accuracy into account. We also include the cell reported by Pham et al., trained from scratch. In our experiments, our controller is allowed to choose a cell size in the range $[2-24]$. Our search space consists of four activation functions: identity, sigmoid, ReLU and tanh. When reporting cell sizes, we only consider the data needed for the actual cell, ignoring the size of the associated encoders and decoders. In order to be able calculate an estimated cell size in bytes, we assumed a parameter size of 4 bytes. When performing architecture search, the weight of our cache-constraint is 10^{-13} and the weight of the network's performance is left at 1. Our agent tends to have little trouble understanding and optimizing for its cache-constraint, and thus, we opted to put a larger emphasis on accuracy.
Our hidden states and embeddings both have a dimension of 1000 during the search process. Weight-tying is used for our encoders and decoders, as described by Inan et al. [25], alongside L2 regularization, weighted by a factor of 10^{-7} and gradient clipping, with a maximum of 0.25. We also add our controller's sample entropy to its reward, weighted by a factor of 0.0001. Our generated cells are also enhanced using highway connections [26]. We also note that, contrary to Zoph & Le [1], we do not perform a grid-search over hyperparameters after training our cell. When training our cells, we used the DARTS codebase [27] with the default seed, and train our networks for 3600 epochs, similar to [28].

4.3. Convolutional Neural Networks

4.3.1. Use Case

For our CNNs, we consider an application requiring classification of 32x32 RGB images at a framerate of 1 Frame(s) per Second (FPS). Besides this, we also consider a maximum memory size of 5MB for our CNN. We set much stricter memory constraints for our CNNs when compared to our RNNs, because there exist various techniques to reduce the memory consumption of CNNs that can be utilized by a NAS algorithm, such as depthwise-separable convolutions [29]. Besides this, 3 of the 5 activation functions in our search space do not require any trainable parameters, resulting in even more compact networks.

4.3.2. Configuration

When searching for CNN cells, we use a proxy-network trained on the CIFAR-10 dataset [30]. This proxy-network is smaller than the network used for evaluating the final accuracy of the cell architecture, to speed up the training of the shared weights. Our proxy architecture consists of 3 stacks, witch each stack having 2 convolution

cells, and 1 reduction cell, except for the final stack, which is missing the reduction cell, similar to figure 4 in [3]. The first cell in our network has 6 filters. When estimating the resource-utilization of the final architecture, we use a similar architecture with multiple (6) convolution cells per stack with a single reduction cell at the end, except for the final stack, which doesn't have a final reduction cell. In this case, our first cell has 36 filters, following [28] (Section 4.2). When estimating the resource consumption of our cells, we do not count the auxilliary towers used in training, since they are discarded at inference time [11], this results in the number of parameters we report being lower than in [3] and [28]. Our RL controller is trained with a learning rate of 0.00035. The activation functions included in our search space are Identity, ReLU, Depthwise Separable Convolution (3x3 and 5x5), Average Pooling (3x3) and Max Pooling (3x3). When a separable convolution is selected, the convolution kernel is applied twice, together with a ReLU-non-linearity and Batchnorm, following [3] and [6]. Similar to our experiments with RNNs, we set the weight of the cache constraint to 10^{-13} and the weight for the network's performance is left at 1. The shared weights are trained using a cosine annealed learning rate schedule with warm restarts [31], with $lr_{max} = 0.05$, $lr_{min} = 0.001$, $T_0 = 10$ and $T_{mul} = 2$. The shared weights are regularized using L2 weight regularization, weighted by a factor of 0.0001. While training, we alternate between training the shared weights for 1 epoch in batches of 128 images, and training the controller weights for 400 episodes. During shared weight training, we clip the norm of our gradients to 0.25, in order to prevent exploding gradient issues. Following [6], we insert pointwise convolutions wherever necessary to match up the dimensions of our feature maps. When training our final CNN cells from scratch, we used the DARTS codebase [27] with a batch size of 32, and the default seed to train our cells for 300 epochs.

5. Results

Diagrams of the different cells discovered by our NAS algorithm can be found in appendix Appendix A. When reporting cache rewards, we report the rewards without multiplying by the preset weight (μ_0) of 10^{-13}.

5.1. Recurrent Neural Networks

Table 2 provides an overview of the results we obtained during our RNN experiments.

System	Validation Perplexity	Memory Use (MB)	Inference Latency (ms)	Cache Reward
ENAS	57.46	136	382.97	-112.54×10^6
No Constraints	71.42	56	159.16	-45.87×10^6
Constraints	69.71	40	159.27	-32.54×10^6

Table 2: Overview of our results, showing validation perplexity, memory use, inference latency and cache reward.

We also provide a graphical comparison between our 3 runs in Fig. 1. This figure does not show the value of the cache constraint, since it is strongly correlated with memory use. The figure shows that, in terms of resources, our dynamically designed cells outperform ENAS by quite a large margin, while only sacrificing a relatively small amount of performance.

It is also notable that even when given no constraints, our algorithm prefers to design smaller cells. We hypothesize this is caused partially by the regularization applied in the controller, by adding the entropy of every decision to the controller's reward, we are essentially punishing it for making more decisions, resulting in smaller cells being more rewarding. This hypothesis is further confirmed by our CNN experiments, that do not show this behaviour, and do not use this regularization technique. This effect has likely not been observed before, since most previous research uses fixed cell sizes.

5.1.1. Cells

Using the DARTS codebase [27], we were able to achieve performance similar to that by Pham et al. [3] (55.8 Perplexity points on Penn Treebank (PTB) [32]) and Li et al. [28] (60.3 Perplexity points using DARTS' [27] training code) using the cell shown in Fig. A.5.a. In order to be able to provide a fair comparison, we trained the cell reported

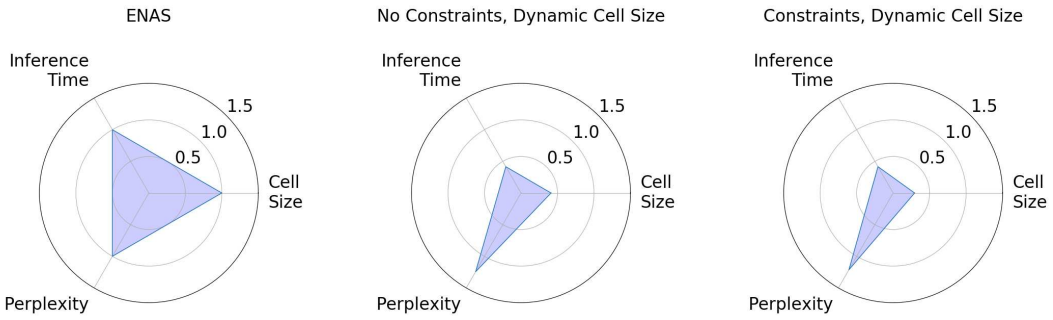


Figure 1: Graphical comparison of the different RNN cells, all numbers are normalized to ENAS' scores. A smaller surface area means a system is better.

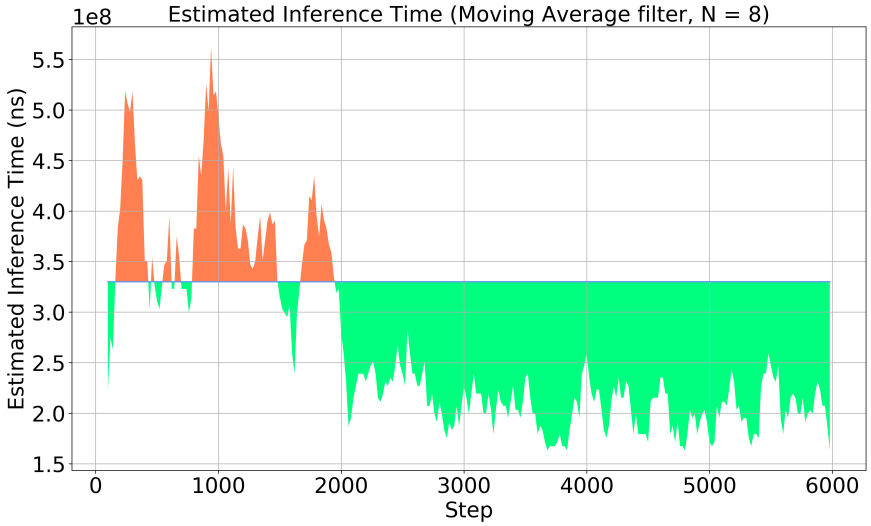


Figure 2: Estimated inference time as a function of training steps, showing that ENAS is capable of meeting and exceeding a given hard constraint in a reasonable amount of time. Red areas indicate constraint violations, green areas indicate that the constraint was met.

by Pham et al. [3] ourselves, reaching a test perplexity of 57.46.

Running our algorithm without constraints and with dynamic cell size, results in cells which are drastically smaller than those generated when the cell size is fixed. Cells generated with this combination of parameters typically contain 3-4 blocks, due to their simplicity, these cells tend to exhibit a worse validation accuracy than their fixed-size counterparts. The cell that was used to gather results is shown in Fig. A.5.b, and was able to achieve a validation perplexity of 71.42, which is significantly worse than the state-of-the-art, but still quite good given the resource constraints imposed on it.

The smallest cells are generated when ENAS is put under resource constraints and given the ability to choose the size of the cells it generates. The cell we used to gather our results consists of a single line of blocks, shown in Fig. A.5.c. This cell is similar to the cell generated without constraints, which also consisted of sigmoid and ReLU blocks. Under constraints the controller presumably creates long, snake-like cells with a single line of blocks, because this simple structure results in the least amount of connections, which in turn reduces the necessary amount of memory for the cell. It should also be noted that the generated cell contains two sigmoid activation functions, which are 19.97

times slower than ReLU activations and 31.7 times slower than identity activations, showing that ENAS still tries to keep accuracy in mind, even when designing cells in a constrained matter. It also shows a preference in the algorithm for a lower amount of complex activation functions, rather than a higher number of simple activation functions, which goes against the trend of increasingly deeper neural networks. Fig. 2 also demonstrates that ENAS is effectively capable of meeting a given hard constraint, while it initially has trouble meeting the constraint, it quickly realizes how to design cells that meet the given requirements.

5.2. Convolutional Neural Networks

System	Top-1 Accuracy	Memory Use (MB)	Inference Latency (ms)	Cache Reward
ENAS	92.55%	8.54	1675.98	-15.347×10^6
No Constraints	88.43%	8.61	2100.42	-15.503×10^6
Constraints	81.40%	4.01	943.53	-6.648×10^6

Table 3: Overview of our results, showing top-1 accuracy, memory use, inference latency and cache reward.

5.2.1. Cells

Similarly to our results with RNNs, we notice that for convolutional neural networks, enabling constraints leads to smaller cells, as demonstrated by the cells we found in figure A.6. Upon closer inspection of the generated cells, we notice that the convolution and reduction cell actually share the same architecture. When inspecting the logs of our experiments, we find that this behaviour is actually learned by the controller, rather than a result of an artifact, since at the beginning of the search process, the controller generates two distinct cells for both functions. Another noteworthy feature of our cell is the absence of convolutions with kernel size 3x3, while the cell does use 5x5 convolutions, which tend to use more resources. This would suggest, again similar to our RNN experiments, that our NAS algorithm still attempts to take accuracy into account, even when optimizing for resource utilization and prefers fewer complex activation functions over a larger amount of simple activation functions. When looking at the cells reported by Pham et al. in [3], we noticed that their cells do not contain any max-pooling operations, while our cells do contain max-pooling operations, our algorithm has also decided to ignore one of the activation functions in our search space, the 3x3 depthwise separable convolution. This effect seems to be less severe when the algorithm builds bigger cells, such as in our experiments without constraints and controller-regularization. This underlines the importance of search-space design, which was already suggested by [9]. When comparing ENAS to our cell without constraints, we see that both cells perform similarly, suggesting that the added freedom the controller has when using a dynamic cell size, only really matters when designing constrained cells.

When plotting the distribution of activation functions in each cell, depicted in figure 4, we see that normal and reduction cells typically have a similar distribution of activation functions. This is further reinforced by the fact that our constrained normal and reduction cell share the same architecture. We hypothesize that it might be beneficial to loosen the restrictions placed on our NAS algorithm, by letting it design multiple cells, and choosing the strides for every filter separately, similar to the original work of Zoph & Le [1].

6. Conclusion

In this section, we discuss the conclusions we were able to draw from our experiments, noting the effects of allowing a dynamically chosen cell size and the efficacy of applying constraints to a NAS system.

6.1. Dynamic Cell Size

When ENAS is able to choose the size of the cells it generates, it tends to generate smaller cells. We suggest that this occurs because we add the controller's sample entropy to its reward function. Since the sample entropy is summed across all decisions, making less decisions produces less entropy, thus encouraging smaller networks. When

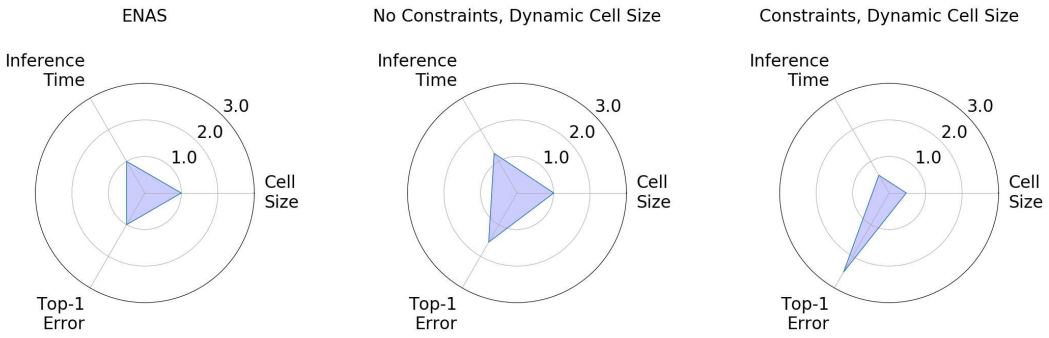


Figure 3: Graphical comparison of the different CNN cells, all numbers are normalized to ENAS' scores. A smaller surface area means a system is better.

disabling this operation we noticed that ENAS has a tendency to generate larger cells, but it also tends to ignore soft constraints, generating networks that barely comply with the given hard constraints. An alternative to this would be to use the mean rather than the sum of the sample entropies.

6.2. Constraints

We can confidently conclude that our version of ENAS is capable of meeting a given set of hard-constraints, while still also effectively optimizing for soft-constraints. We suggest that this is a promising area of research, and further enhancements could be made to make ENAS able to meet these constraints in a faster manner, while still allowing for maximum accuracy.

7. Future Work

7.1. Multi Objective Reinforcement Learning

Our current implementation uses a very simple scalarization method to solve a multi-objective problem. There are many techniques designed to allow agents to more easily solve multi-objective problems [33], some of which might be used to enhance the performance of our controller. [34] Currently, our reward is a linear combination of a set of soft constraints, multiplied by the AND-operation of all hard constraints. This has been shown to work, however, it might be worth exploring other scalarization methods such as Hypervolume-based Scalarization [35] and Chebyshev Scalarization [36]. Van Moffaert et al. have shown that these methods can outperform simple linear combinations in multi-objectivized versions of single-objective problems by a large margin, making them an interesting option to consider in future research.

Nguyen proposes a new deep Q-learning based framework consisting of both single- and multi-policy DQN [34], showing promising results on the Deep Sea Treasure problem [37]. In their research, they consider both linear and non-linear techniques to scalarize a multi-objective problem. Nguyen prevents having to re-train their Q-learning system when the weights associated with their objectives (μ_i in our case) change, by training multiple agents in parallel. While this is feasible when there are only a few parameters with a small range of possible values, it quickly becomes impossible when the amount or range of objectives changes.

Wiering et al.use a two-stage approach to learn the set of optimal policies [33] that are applicable in the Deep Sea Treasure problem. First, an agent explores the environment, attempting to explore and learn a model of the environment. After the environment has been modeled, dynamic programming is used to find the pareto-optimal solutions to the learned model. Using model-based RL in our problem would allow us to decouple NAS from the architecture it is modeling for, making it possible to re-use the NAS system while only having to re-learn the model.

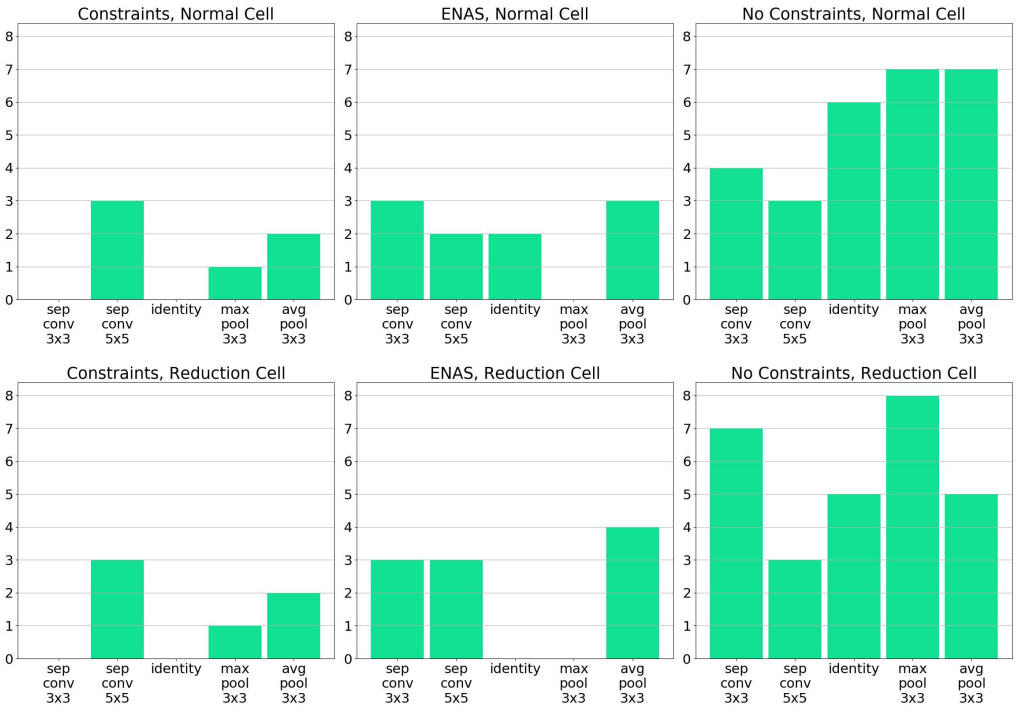


Figure 4: Distribution of activation functions, per CNN cell

7.2. Graph Embeddings

Recently, promising research has been done on deep learning systems operating on graphs [38] [39]. Our system is currently generating computational graphs node-by-node in a recurrent fashion. It could, however, be valuable to progressively optimize a single graph by performing operations on its nodes. In this respect, graph embeddings could be a very useful tool.

This is similar to the work done by Zhou et al [40]. Zhou et al.start from an existing neural network and use a set of recurrent networks to perform mutations on this network. Used operations are scaling (changing an existing parameter), insert (inserting a new layer) and remove (removing an existing layer). Using these techniques, Zhou et al. seem to have achieved reasonably good performance. Unfortunately, they fail to mention whether their test accuracy on CIFAR-10 is top-1, top-3 or top-5, as well as fail to provide comparisons to other NAS methods.

An important downside of working with Graph Embeddings is that they typically work best on large graphs, implying that their application in NAS would be best suited when operating on whole networks, rather than on single cells. This is contrary to many state-of-the-art works, which favour designing cells over entire networks [3] [6] [27], and typically results in slower search times.

7.3. Non-Stationarity

Finally, we would also like to address the non-stationarity introduced to NAS by performing shared weight training at the same time as reinforcement learning. Because the shared weight training and reinforcement learning are performed periodically, the environment our RL agent operates in is continuously changing while the agent is learning. This can can cause issues with the convergence properties of our system, and is a known problem in other disciplines such as Multi-Agent Reinforcement Learning [41]. A simple solution to solving this problem would be to perform shared-weight training beforehand, and only use the trained shared weights during the architecture search process, without updating them. This could reduce search time, but increase the overall necessary time, since the shared weights need to be trained separately. By completely separating the shared weights from the reinforcement learning agent, they could also be re-used between different NAS runs, again resulting in significant time savings. The main problem that could hinder adoption of this technique, is the lack of an established training protocol, and the

low amount of correlation between the accuracy of shared-weight architectures and their counterparts trained from scratch. Taking this approach would make weight-shared NAS more similar to the approach of [5], which uses a surrogate model to predict the accuracy of a given architecture.

Acknowledgements

We gratefully acknowledge the support of the NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.
This research received funding from the Flemish Government (AI Research Program).

References

[1] B. Zoph, Q. V. Le, Neural architecture search with reinforcement learning, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, OpenReview.net, 2017.
URL https://openreview.net/forum?id=r1Ue8Hcxg

[2] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, Machine Learning 8 (1992) 229–256.
URL https://link.springer.com/article/10.1007/BF00992696

[3] H. Pham, M. Guan, B. Zoph, Q. Le, J. Dean, Efficient neural architecture search via parameters sharing, in: J. Dy, A. Krause (Eds.), Proceedings of the 35th International Conference on Machine Learning, Vol. 80 of Proceedings of Machine Learning Research, PMLR, Stockholmsmässan, Stockholm Sweden, 2018, pp. 4095–4104.
URL http://proceedings.mlr.press/v80/pham18a.html

[4] T. Elsken, J. H. Metzen, F. Hutter, Efficient multi-objective neural architecture search via lamarckian evolution, in: International Conference on Learning Representations, 2019.
URL https://openreview.net/forum?id=ByME42AqK7

[5] J. Dong, A.-C. Cheng, D. Juan, W. Wei, M. Sun, DPP-Net: Device-aware progressive search for pareto-optimal neural architectures, in: Lecture Notes in Computer Science, Vol. 11215, 2018, pp. 540–555.
URL https://link.springer.com/chapter/10.1007/978-3-030-01252-6_32

[6] B. Zoph, V. Vasudevan, J. Shlens, Q. V. Le, Learning transferable architectures for scalable image recognition, CoRR abs/1707.07012. arXiv:1707.07012.
URL http://arxiv.org/abs/1707.07012

[7] E. Real, A. Aggarwal, Y. Huang, Q. V. Le, Regularized evolution for image classifier architecture search, CoRR abs/1802.01548. arXiv:1802.01548.
URL http://arxiv.org/abs/1802.01548

[8] K. Kandasamy, J. Schneider, B. Póczos, E. P Xing, Neural architecture search with bayesian optimisation and optimal transport, arXiv preprint arXiv:1802.07191.
URL https://arxiv.org/abs/1802.07191

[9] T. Elsken, J. H. Metzen, F. Hutter, Neural architecture search: A survey, Journal of Machine Learning Research 20 (55) (2019) 1–21.
URL http://jmlr.org/papers/v20/18-598.html

[10] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE Computer Society, Los Alamitos, CA, USA, 2016, pp. 770–778. doi:10.1109/CVPR.2016.90.
URL https://doi.ieeecomputersociety.org/10.1109/CVPR.2016.90

[11] C. Szegedy, Wei Liu, Yangqing Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015, pp. 1–9.
URL https://ieeexplore.ieee.org/document/7298594

[12] K. Yu, C. Sciuto, M. Jaggi, C. Musat, M. Salzmann, Evaluating the search phase of neural architecture search, in: International Conference on Learning Representations, 2020.
URL https://openreview.net/forum?id=H1loF2NFwr

[13] K. Yu, R. Ranftl, M. Salzmann, How to train your super-net: An analysis of training heuristics in weight-sharing nas, arXiv preprint arXiv:1802.07191.
URL https://arxiv.org/abs/2003.04276

[14] S. K. Mishra, G. P, S. Meher, R. Majhi, A fast multiobjective evolutionary algorithm for finding wellspread pareto-optimal solutions, KanGAL report 2003002.
URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.624.1212

[15] C. Shannon, A mathematical theory of communication, Bell System Technical Journal 27 (1948) 379–423, 623–656.
URL https://onlinelibrary.wiley.com/doi/10.1002/j.1538-7305.1948.tb01338.x

[16] W. Daems, Digital signal processing signal processing systems - textbook (2018).
URL https://www.digmanwaves.net/Printing.html

[17] Information technology – jpeg 2000 image coding system: Core coding system, Standard, International Organization for Standardization, Geneva, CH (October 2019).

[18] B. Chazelle, J. Kilian, R. Rubinfeld, A. Tal, The bloomier filter: An efficient data structure for static support lookup tables, in: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '04, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2004, pp. 30–39.
URL http://dl.acm.org/citation.cfm?id=982792.982797

[19] B. Reagen, U. Gupta, R. Adolf, M. M. Mitzenmacher, A. M. Rush, G. Wei, D. M. Brooks, Weightless: Lossy weight encoding for deep neural network compression, CoRR abs/1711.04686. arXiv:1711.04686.
URL http://arxiv.org/abs/1711.04686

[20] M. Horowitz, 1.1 computing's energy problem (and what we can do about it), in: 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014, pp. 10–14. doi:10.1109/ISSCC.2014.6757323.
URL https://ieeexplore.ieee.org/document/6757323

[21] M. S. Oyamada, F. Zschornack, F. R. Wanger, Accurate software performance estimation using domain classification and neural networks, in: Proceedings. SBCCI 2004. 17th Symposium on Integrated Circuits and Systems Design (IEEE Cat. No.04TH8784), 2004, pp. 175–180.
URL https://ieeexplore.ieee.org/document/1360565

[22] A. Bonenfant, D. Claraz, M. de Michiel, P. Sotin, Early WCET Prediction Using Machine Learning, in: J. Reineke (Ed.), 17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017), Vol. 57 of OpenAccess Series in Informatics (OASIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2017, pp. 1–9. doi:10.4230/OASIcs.WCET.2017.5.
URL http://drops.dagstuhl.de/opus/volltexte/2017/7307

[23] T. Huybrechts, T. Cassimon, S. Mercelis, P. Hellinckx, Introduction of deep neural network in hybrid wcet analysis, in: 3PGCIC 2018: Advances on P2P, Parallel, Grid, Cloud and Internet Computing, Vol. 24 of Lecture Notes on Data Engineering and Communications Technologies, 2019, pp. 415–425.
URL https://www.springer.com/gp/book/9783030026066

[24] B. L. Wong, Essential Study Skills, 8th Edition, no. ISBN 9781285430096 in Essential Study Skills, Cengage, 2015.

[25] H. Inan, K. Khosravi, R. Socher, Tying word vectors and word classifiers: A loss framework for language modeling, CoRR abs/1611.01462. arXiv:1611.01462.
URL http://arxiv.org/abs/1611.01462

[26] J. G. Zilly, R. K. Srivastava, J. Koutník, J. Schmidhuber, Recurrent highway networks, in: D. Precup, Y. W. Teh (Eds.), Proceedings of the 34th International Conference on Machine Learning, Vol. 70 of Proceedings of Machine Learning Research, PMLR, International Convention Centre, Sydney, Australia, 2017, pp. 4189–4198.
URL http://proceedings.mlr.press/v70/zilly17a.html

[27] H. Liu, K. Simonyan, Y. Yang, Darts: Differentiable architecture search, arXiv preprint arXiv:1806.09055.
URL https://openreview.net/forum?id=S1eYHoC5FX

[28] L. Li, A. Talwalkar, Random search and reproducibility for neural architecture search, CoRR abs/1902.07638. arXiv:1902.07638.
URL http://arxiv.org/abs/1902.07638

[29] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam, Mobilenets: Efficient convolutional neural networks for mobile vision applications, CoRR abs/1704.04861. arXiv:1704.04861.
URL http://arxiv.org/abs/1704.04861

[30] A. Krizhevsky, Learning multiple layers of features from tiny images, Tech. rep., University of Toronto (April 2009).
URL https://www.cs.toronto.edu/ kriz/learning-features-2009-TR.pdf

[31] I. Loshcilov, F. Hutter, Sgdr: Stochastic gradient descent with warm restarts, in: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings, Openreview.net, 2017.
URL https://openreview.net/forum?id=Skq89Scxx

[32] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, B. Schasberger, The penn treebank: Annotating predicate argument structure, in: Proceedings of the Workshop on Human Language Technology, HLT '94, Association for Computational Linguistics, Stroudsburg, PA, USA, 1994, pp. 114–119. doi:10.3115/1075812.1075835.
URL https://doi.org/10.3115/1075812.1075835

[33] M. Wiering, M. Withagen, M. Drugan, Model-based multi-objective reinforcement learning, in: 2014 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014, pp. 1–6. doi:10.1109/ADPRL.2014.7010622.
URL https://ieeexplore.ieee.org/document/7010622

[34] T. T. Nguyen, A multi-objective deep reinforcement learning framework, CoRR abs/1803.02965. arXiv:1803.02965.
URL http://arxiv.org/abs/1803.02965

[35] K. "Van Moffaert, M. M. Drugan, e. R. C. Nowé, Ann", P. J. Fleming, C. M. Fonseca, S. Greco, J. Shaw, Hypervolume-based multi-objective reinforcement learning, in: Evolutionary Multi-Criterion Optimization, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 352–366.
URL https://link.springer.com/chapter/10.1007/978-3-642-37140-0_28

[36] K. Van Moffaert, M. M. Drugan, A. Nowé, Scalarized multi-objective reinforcement learning: Novel design techniques, in: 2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2013, pp. 191–199. doi:10.1109/ADPRL.2013.6615007.
URL https://ieeexplore.ieee.org/document/6615007

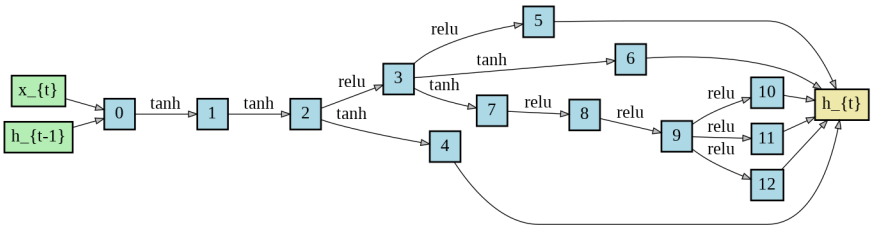
[37] P. "Vamplew, R. Dazeley, A. Berry, R. Issabekov, E. Dekker, Empirical evaluation methods for multiobjective reinforcement learning algorithms, Machine Learning 84 (1) (2011) 51–80. doi:10.1007/s10994-010-5232-5.
URL https://doi.org/10.1007/s10994-010-5232-5

[38] H. "Cai, V. W. Zheng, K. Chen-Chuan Chang, A comprehensive survey of graph embedding: Problems, techniques and applications, IEEE Transactions on Knowledge and Data Engineeringdoi:10.1109/TKDE.2018.2807452.
URL https://ieeexplore.ieee.org/document/8294302

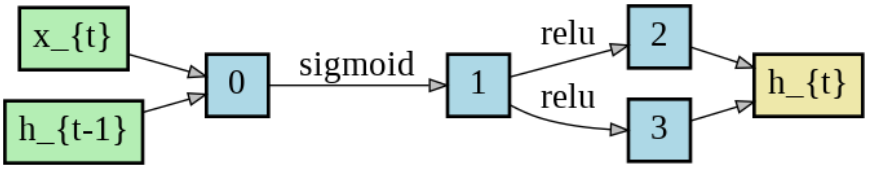
[39] B. Perozzi, R. Al-Rfou, S. Skiena, Deepwalk: Online learning of social representations, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '14, ACM, New York, NY, USA, 2014, pp. 701–710. doi:10.1145/2623330.2623732.

URL http://doi.acm.org/10.1145/2623330.2623732
[40] Y. Zhou, S. Ebrahimi, S. Ö. Arik, H. Yu, H. Liu, G. Diamos, Resource-efficient neural architect, CoRR abs/1806.07912. arXiv:1806.07912.
URL http://arxiv.org/abs/1806.07912
[41] F. A. Oliehoek, C. Amato, A Concise Introduction to Decentralized POMDPs, SpringerBriefs in Intelligent Systems, Springer International Publishing, 2016.
URL https://www.springer.com/gp/book/9783319289274

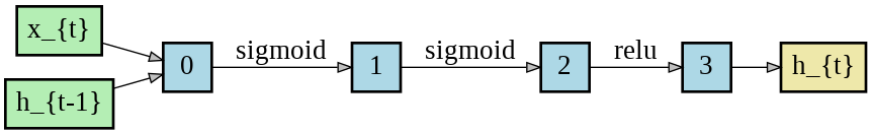
Appendix A. Found Cell Architectures



(a) ENAS

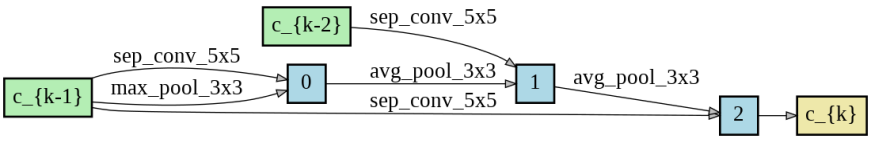


(b) No Constraints

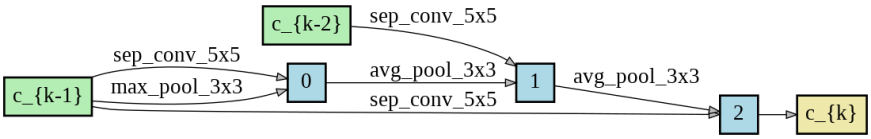


(c) Constraints

Figure A.5: Comparison of the different RNN cells generated

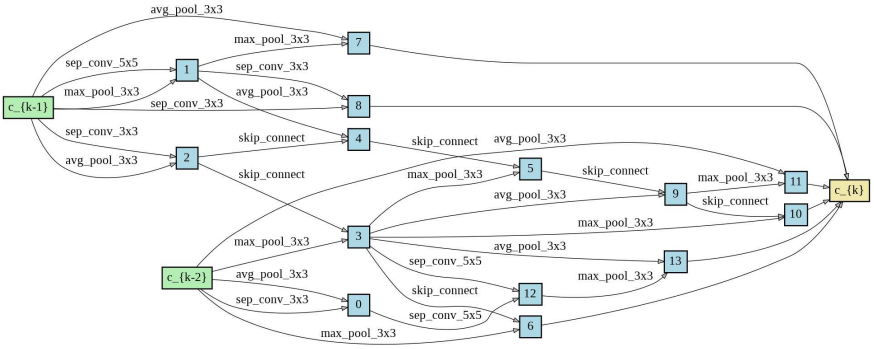


(a) Constraints, Normal Cell

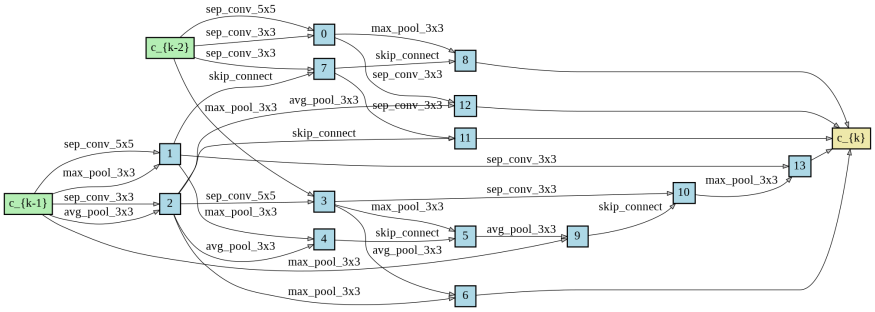


(b) Constraints, Reduction Cell

Figure A.6: Normal and Reduction CNN cell when constraints are enabled

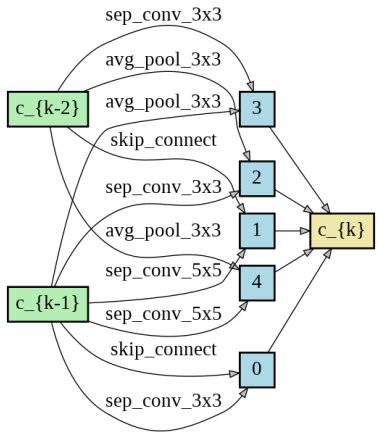


(a) No Constraints, Normal Cell

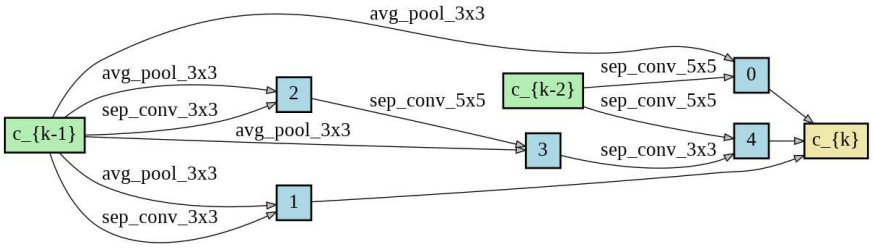


(b) No Constraints, Reduction Cell

Figure A.7: Normal and Reduction CNN cell when constraints are disabled



(a) ENAS, Normal Cell



(b) ENAS, Reduction Cell

Figure A.8: Normal and Reduction CNN cell from ENAS