Learning Time-Efficient Deep Architectures with Budgeted Super Networks

Tom Veniat and Ludovic Denoyer
Sorbonne Universités,
UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
tom.veniat@lip6.fr, ludovic.denoyer@lip6.fr

Abstract

Learning neural network architectures is a way to discover new highly predictive models. We propose to focus on this problem from a different perspective where the goal is to discover architectures efficient in terms of both prediction quality and computation cost, e.g time in milliseconds, number of operations... For instance, our approach is able to solve the following task: *find the best neural network architecture (in a very large set of possible architectures) able to predict well in less than 100 milliseconds on my mobile phone*. Our contribution is based on a new family of models called *Budgeted Super Networks* that are learned using reinforcement-learning inspired techniques applied to a budgeted learning objective function which includes the computation cost during disk/memory operations at inference. We present a set of experiments on computer vision problems and show the ability of our method to discover efficient architectures in terms of both predictive quality and computation time.

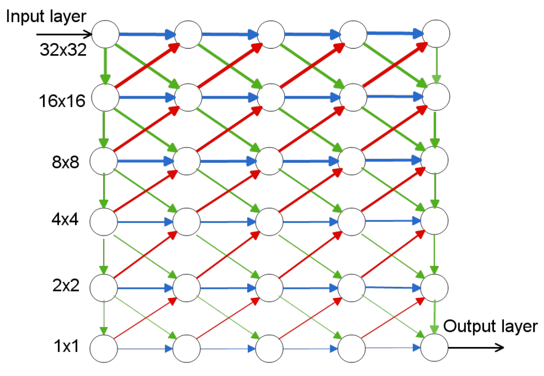
1 Introduction

In the Deep Learning community, finding the best architecture of a neural network for a given task is a key problem that is mainly addressed *by hand* or using validation techniques. For instance, in computer vision, this has lead to particularly well-known models like GoogleNet [21] or ResNet [8]. More recently, there is a surge of interest in developing techniques able to automatically discover efficient neural network architectures. Different algorithms have been proposed including evolutionary methods [20, 16, 17] or reinforcement learning-based approaches [24]. But in all cases, this selection is usually made solely based on the final predictive performance of the model such as the accuracy.

When facing real-world applications, this predictive performance is not the only measure that matters. Indeed, learning a very good predictive model with the help of a cluster of GPUs might lead to a neural network that can be incompatible with low-resources mobile devices: it would involve too many complex operations resulting in a very low computation speed at use. Another example concerns distributed models in which part of the computation is made *in the cloud* and the other part is made *on the device*. In that case, an efficient architecture would have to predict accurately while minimizing the amount of exchanged messages between the cloud and the device. One important research direction is thus to propose models that can learn to take into account the computation complexity in addition to the quality of the prediction.

We propose to tackle this issue as a problem of automatically learning a neural network architecture that is efficient in terms of both prediction and computation time at inference. This task has been recently targeted by different models [1, 15, 12], with different learning techniques. The main difference of our approach is that it can be used with any computation cost function like the time in milliseconds or the number of operations made by the network, and only requires the user to specify a maximum authorized cost. For that, we propose a *budgeted learning approach* that integrates

Submitted to 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.



(a)

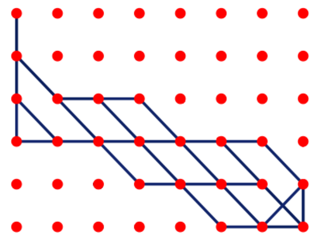

(b)

Figure 1: Examples of super Network. (a) This network corresponds to the Convolutional Neural Fabric (CNF) Architecture (represented with 5 columns instead of 8 in the experiments). Each line is a set of layers at a particular resolution level, each edge corresponds to a 2D convolutional transformation + batch normalization. The width of each edge represents the cost of each module. (b) A network whose architecture is a subgraph of the CNF.

this computation cost directly in the learning objective function. Our model called *Budgeted Super Network* (BSN) is based on the following principles: (i) the user provides a (big) *Super Network* (see Section 3) that defines a very large set of possible final network architectures as well as the maximum inference cost. (ii) Since finding the best time-efficient architecture in this set is an intractable combinatorial problem (Section 4.1), we relax this optimization problem and propose a stochastic model (called *Stochastic Super Networks* – Section 4.2) that can be optimized using reinforcement learning-inspired methods (Section 4.3). We show that the optimal solution of this new problem corresponds to the optimal network architecture (Proposition 1). At last, we evaluate our approach on both image classification and image segmentation problems on which we show that our model is able to reduce the computation time while keeping a very high accuracy (Section 5). The related work is presented in Section 2.

2 Related Work

Different authors have proposed to provide networks with the ability to learn to select the computations that will be applied. This is the case for example for classification in [2, 24] based on Reinforcement learning techniques, in [19] based on gating mechanisms, in [17] based on evolutionary algorithms or even in [4] based on both RL and evolutionary techniques. One strong difference with our approach is that these models are only guided by the final predictive performance of the network. For example in [24], a controller neural net can propose a “child” model architecture, which can then be trained and evaluated before trying a new architecture. The process is repeated iteratively until a good architecture is obtained. Moreover, in this approach each generated architecture is learned to convergence on the training set, resulting in a very low training speed while our model learns the parameters of the modules and the architecture simultaneously.

When the objective is to learn time or memory efficient models, different methods have been proposed. The most common approach is to simplify the learned network typically by removing some connections between neurons. The oldest approach is certainly the Optimal Brain Surgeon [7] which removes weights in a classical neural network architecture. The problem of network compression can also be seen as a way to speedup a particular architecture, for example by using quantization of the weights of the network [22]. Pruning and quantization techniques can also be combined [6] to obtain impressive performances.

Other algorithms include the use of hardware efficient operations that allow a high speedup. For example [3] combines algorithmic optimizations and architecture improvements to speed up the response time of a network while keeping high accuracy. This technique is effective but requires a strong expertise to find the best optimization and is also highly specific to each architecture and/or hardware. Note that the computation speed can be reduced by using parallelized inference approaches which is not the topic of this paper.

In some other works, authors learn time-efficient models in an end-to-end manner. For example, the model proposed in [1] can be seen as an extension of dropout where the probability of sampling neurons is conditioned on currently processed input (conditional computation). In the same vein, the model proposed in [15] aims at learning routes in Super networks. It is certainly the most similar approach to ours, while keeping several important differences. First, the model is based on an explicit computation cost formulation while our approach can be used for any type of cost. Moreover, while our model converges to a static architecture, allowing to keep only the blocks used by the trained model, their model learns dynamic routes, causing an overhead in the computation cost and forcing to keep the whole network to do further inferences. At last, the number of possible architectures explored is smaller in this approach than in ours.

3 Super Networks

We consider the classical supervised learning problem defined by an input space $\mathcal{X}$ and an output space $\mathcal{Y}$. In the following, input and output spaces correspond to multi-dimensional real-valued spaces (vectors, matrices or larger tensors). The training set is denoted as $\mathcal{D} = \{(x^1, y^1), ..., (x^\ell, y^\ell)\}$ where $x^i \in \mathcal{X}$, $y^i \in \mathcal{Y}$ and ℓ is the number of supervised examples. At last, we consider a model $f : \mathcal{X} \rightarrow \mathcal{Y}$ that predicts an output given a particular input.

We first describe a family of models called *Super Networks* (S-network)[1] since our contribution presented in Section 4 will be a (stochastic) extension of this model. Note that the principle of Super Networks is not new and the same ideas have been already proposed in the literature under different names, e.g Deep Sequential Neural Networks [2], Neural Fabrics [18], or even PathNet [4] – and for different tasks.

A Super Network is composed of a set of layers connected together in a direct acyclic graph structure. Each edge is a (small) neural network, and the Super Network corresponds to a particular combination of these neural networks and defines a computation graph. Examples of S-networks are given in Figure 1. More formally, let us denote $l_1, ..., l_N$ a set of layers, N being the number of layers, such that each layer l_i is associated with a particular representation space $\mathcal{X}_i$ which is a multi-dimensional real-valued space. l_1 will be the *input layer* while l_N will be the *output layer*. We also consider a set of (differentiable) functions $f_{i,j}$ associated to each possible pair of layers such that $f_{i,j} : \mathcal{X}_i \rightarrow \mathcal{X}_j$. Each function $f_{i,j}$ will be referred as a *module* in the following: it takes data from $\mathcal{X}_i$ as inputs and transforms these data to $\mathcal{X}_j$. Note that each $f_{i,j}$ will make complex disk/memory/network operations and their computation will have consequences on the inference speed of the S-network. Each $f_{i,j}$ module is associated with parameters in θ, θ being implicit in the notation for sake of clarity.

On top of this structure, a particular architecture $E = \{e_{i,j}\}_{(i,j) \in [1;N]^2}$ is a binary adjacency matrix over the N layers such that E defines a directed acyclic graph (DAG) with a single source node is l_1 and a single sink node l_N. Different matrices E will thus correspond to different super network architectures – see Figure 1 that shows different possible architectures within the same super network. A S-network will be denoted (E, θ) in the following, θ being the parameters of the different *modules*, and E being the architecture of the super network.

Predicting with S-networks: The computation of the output $f(x, E, \theta)$ given an input x and a super network (E, θ) is made through a classic forward algorithm, the main idea being that the output of modules $f_{i,j}$ and $f_{k,j}$ leading to the same layer l_j will be added in order to compute the value of l_j. Let us denote $l_i(x, E, \theta)$ the value of layer l_i for input x, the computation is recursively defined as:

$$\text{Input:} l_1(x, E, \theta) = x, \text{Layer Computation: } l_i(x, E, \theta) = \sum_k e_{k,i} f_{k,i}(l_k(x, E, \theta)) \tag{1}$$

Note that learning the parameters θ can be made using classical back-propagation and gradient-descent techniques.

[1]The name *Super Network* comes from [4] which presents an architecture close to ours.

4 Learning Time-efficient architectures

Our main idea is the following: we now consider that the structure E of the S-network (E, θ) describes not a single neural network architecture but a set of possible architectures. Indeed, each sub-graph of E (subset of edges) corresponds to a S-network. A sub-graph of edges will be denoted $H \odot E$ where H corresponds to a binary matrix in charge of selecting the edges in E, $\odot$ denoting the Hadamard product between H and E. Our objective will thus be to identify the best matrix H such that the corresponding S-network $(H \odot E, \theta)$ will be a network efficient both in terms of predictive quality and in terms of computation speed.

The next sections are organized as follows: (i) First, we formalize this problem as a combinatorial problem where one wants to discover the best matrix H in the set of all possible binary matrices of size $N \times N$. Since this optimization problem is intractable, we propose a new family of models called *Stochastic Super Networks* where the edges of E are sampled following a parametrized distribution Γ before each prediction. We then show that the resulting budgeted learning problem is continuous and that its solution corresponds to the optimal solution of the initial budgeted problem (Proposition 1). We then propose a practical learning algorithm based on reinforcement learning techniques to learn Γ and θ simultaneously.

4.1 Budgeted Architectures Learning

Let us consider H a binary matrix of size $N \times N$. Let us denote $C(H \odot E) \in \mathbb{R}^+$ the computation cost[2] associated to the computation of the S-network $(H \odot E, \theta)$ which can be of different nature (see Section 5). Let us also define $\mathbf{C}$ the maximum cost the user would allow to the wanted model. For instance, when solving the problem of *learning a model with a computation time lower than 200 ms* then $\mathbf{C}$ is equal to $200ms$. We aim at solving the following budgeted learning problem:

$$H^*, \theta^* = \arg\min_{H,\theta} \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) \quad \text{under constraints: } C(H \odot E) \leq \mathbf{C} \tag{2}$$

In this article, we propose to focus on a soft version of the problem written as:

$$H^*, \theta^* = \arg\min_{H,\theta} \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) \quad + \lambda \max(0, C(H \odot E) - \mathbf{C}) \tag{3}$$

where λ corresponds to the importance of the cost penalty. Note that the proposed learning model directly includes the computational cost which will be specific to the particular infrastructure on which the model is used. For instance, if C is the cost in milliseconds, the value of $\mathbf{C}(H \odot E)$ will not be the same depending on the device on which the model is used. While previous works make assumptions on the structure of the cost function [15], our approach does not rely on any constraint concerning $C(H \odot E)$ except the fact that this cost can be measured during training. In other words, solving this problem on a mobile device will produce a different structure that the one obtained when solving this problem on a cluster of GPUs which is an important property of our model.

Finding a solution to this learning problem is not trivial since it involves the computation of all possible architectures which is prohibitive ($\mathcal{O}(2^N)$ in the worst case). We explain in the next section how this problem can be solved using *Stochastic Super Networks*.

4.2 Stochastic Super Networks

Now, given a particular architecture E, we consider the following stochastic model – called **Stochastic Super Network** (SS-network) – that computes a prediction in two steps: (i) first a sub-graph H is sampled based on a distribution with parameters Γ. This operation will be denoted $H \sim \Gamma$ (ii) The final prediction is made using this sub-graph i.e by computing $f(x, H \odot E, \theta)$. This inference algorithm is described in Algorithm 1. A SS-network is thus defined by a triplet (E, Γ, θ), Γ and θ being both learned on a training set.

We can rewrite the budgeted learning objective of Equation 3 as:

$$\Gamma^*, \theta^* = \arg\min_{\Gamma,\theta} \frac{1}{\ell} \sum_i \mathbb{E}_{H \sim \Gamma} \left[\Delta(f(x^i, H \odot E, \theta), y^i) + \lambda \max(0, C(H \odot E) - \mathbf{C})\right] \tag{4}$$

[2]Note that we consider that the cost only depends on the network architecture. The model could easily be extended to costs that depend on the input x to process

Algorithm 1 Stochastic Super Network forward algorithm

```
1: procedure SSN-FORWARD($x, E, \Gamma, \theta$)
2:     $H \sim \Gamma$                                              ▷ as explained in Section 4.2
3:     For $i \in [1..N]$, $l_i \leftarrow \emptyset$   EndFor                        ▷ Init layers values
4:     $l_1 \leftarrow x$                                           ▷ Set the input in the first layer
5:     For $i \in [2..N]$,$l_i \leftarrow \sum_{k<i} e_{k,i} h_{k,i} f_{k,i}(l_k)$   EndFor
6:     return $l_N$
7: end procedure
```

Proposition 1. *When The solution of Equation 4 is reached, then the models sampled following* (Γ^*) *and using parameters* θ^* *are optimal solution of the problem of Equation 3.*

The demonstration of the proposition is given in supplementary material.

Edge Sampling: For each layer l_i visited following the DAG order of E (from the first layer to the last one) and for all $k < i$: If l_k is connected to the input layer l_1 based on the previously sampled edges, then $h_{k,i}$ is sampled following a Bernoulli distribution with probability[3] $\gamma_{k,i}$. In the other cases then $h_{k,i} = 0$. This sampling procedure avoids sampling edges that would not help in the output computation i.e edges not connected to the input layer.

4.3 Learning Algorithm

We consider a generic case where the cost-function $\mathcal{C}(H \odot E)$ is unknown and can only be observed at the end of the computation of the model over an input x e.g the computation time in seconds. Note that this case also includes stochastic costs where the cost is a random variable, caused by some network latency during distributed computation for example. We now describe the case where C is deterministic, its extension to stochastic values being simple (see supplementary material).

Let us denote $\mathcal{D}(x, y, \theta, E, H)$ the quality of the model $(H \odot E, \theta)$ on a given training pair (x, y):

$$\mathcal{D}(x, y, \theta, E, H) = \Delta(f(x, H \odot E, \theta), y) + \lambda \max(0, C(H \odot E) - \mathbf{C}) \tag{5}$$

We propose to use a REINFORCE-inspired algorithm as in [2, 1] for learning. In that case, let us denote $\mathcal{L}(x, y, \Gamma, \theta)$ the expectation of $\mathcal{D}$ over the possible sampled matrices H:

$$\mathcal{L}(x, y, E, \Gamma, \theta) = \mathbb{E}_{H \sim \Gamma} \mathcal{D}(x, y, \theta, E, H) \tag{6}$$

The gradient of $\mathcal{L}$ can be written as[4]:

$$\begin{aligned} \nabla_{\theta,\Gamma} \mathcal{L}(x, y, E, \Gamma, \theta) = & \sum_H P(H|\Gamma) \left[(\nabla_{\theta,\Gamma} \log P(H|\Gamma)) \mathcal{D}(x, y, \theta, E, H) \right] \\ & + \sum_H P(H|\Gamma) \left[\nabla_{\theta,\Gamma} \Delta(f(x, H \odot E, \theta), y) \right] \end{aligned} \tag{7}$$

The first term corresponds to the gradient over the log-probability of the sampled structure H while the second term is the gradient of the prediction loss given the sampled structure $H \odot E$.

Learning can be made using back-propagation and stochastic-gradient descent algorithms as it is made in Deep Reinforcement Learning models. Note that in practice, in order to reduce the variance of the estimator, the update is made following:

$$\nabla_{\theta,\Gamma} \mathcal{L}(x, y, E, \Gamma, \theta) \approx (\nabla_{\theta,\Gamma} \log P(H|\Gamma))(\mathcal{D}(x, y, \theta, E, H) - \tilde{\mathcal{D}}) + \nabla_{\theta,\Gamma} \Delta(f(x, H \odot E, \theta), y) \tag{8}$$

where H is sampled following Γ, and where $\tilde{\mathcal{D}}$ is the average value of $\mathcal{D}(x, y, \theta, E, H)$ computed on a batch of learning examples.

[3]Note that $\gamma_{k,i}$ is obtained by applying a *logistic* function over a continuous parameter value, but this is made implicit in our notations.

[4]details provided in supplementary material

5 Experiments

We propose to focus on two tasks: **Image classification** using MNIST and CIFAR-10 [11, 10] and **Image Segmentation** using the Part Labels dataset [9]. Standard train/validation/test splits are used. The base Super Network used in the experimental section is a Convolutional Neural Fabric (CNF) [18] – see Figure 1 (a). In this model (details in the supplementary material section), the layers are organized in a $W \times H$ matrix where W is the width of the matrix, and H is the height. In our experiments, $W = 8$ and $H = 6$. Each level in H corresponds to a particular image resolution. Edges correspond to a simple NN composed of a BatchNorm module followed by a 2D-convolution with kernels of size 3, stride and padding depending on the position of the module. We use 64 filters per convolution for both MNIST and Part Labels and 128 filters for CIFAR-10. The input layer is the top-left layer taking images on dimension 32x32 for the MNIST and CIFAR-10 datasets and 256x256 for Part Labels. In classification the output layer is the bottom-right layer followed by a linear + log softmax layer transforming the data to a vector of size K, $K = 10$ being the number of categories. For the Image Segmentation task, the output layer is the top-right layer followed by a log softmax transformation, each pixel being associated with its (log) probability of belonging to one of the $K = 3$ possible pixel categories.

We perform experiments considering two different costs: (i) The *computational cost* (Flop) corresponding to the number of operations made by the convolution modules which is proportional to the number of pixels in the output map produced by the convolution. Intuitively, in the CNF architecture, top edges represents convolutions over high resolution images, being much more expensive than bottom edges which represents convolutions over lower resolution inputs – see Figure 1 (a). (ii) The *Time cost* (T-cost) corresponds to the time spent in milliseconds to compute the prediction. In comparison to the Flop-cost, this cost includes all the extra operations made by the model, and is evaluated in milliseconds after each prediction (measured on the CPU). For example, the Time cost includes rescaling operations (diagonal edges oriented to the top-right direction) that are expensive.

We test the *Budgeted Super Network* model (BSN) with different values of the maximum allowed cost $\mathbf{C}$ and we thus obtain different learned models evaluated in terms of both computation cost and accuracy. We compute the Pareto front of the cost/accuracy curves on the validation set, and report the corresponding results obtained on the test set (see supplementary material).

We compare our algorithm with the CNF model which is equivalent to BSN where the maximum cost $\mathbf{C}$ is unlimited – all edges are kept. Following the description of [18], we also compare our approach to pruned versions of CNF (denoted P-CNF). The pruned algorithm consists in (i) removing the edges of a learned CNF ordered increasingly by their average absolute value of the weights in the convolution kernels – the convolution kernel weights acting as a measure of the importance of each edge – and (ii) in fine-tuning the obtained pruned architecture on the dataset. This pruning procedure has many drawbacks: it is specific to convolution filters and cannot be applied to any S-network architecture while our model is generic to any S-network, it is time-consuming since a fine-tuning must be made after each pruning, and it may produce S-network architectures that are inconsistent since it might remove all the paths between the input layer and the output layer.

5.1 Image Classification

MNIST: Table 1 presents the results obtained on MNIST with the Flop cost and the Time cost compared to (pruned) CNF. On the Flop cost, our model is able to gain 97.8 % of the maximum cost while having 0.48 % of error rate. In terms of time, BSN is able to find a model which computation time is 5.17 ms (in comparison to 68.12 ms for the full CNF) while keeping an error rate of 0.43 %. These two results show the ability of BNF to discover time-efficient architectures, and to handle different cost functions. The pruned version of CNF has a lower performance when the pruning becomes more important. This is mainly due to the fact that the pruning does not consider the overall cost, but also because this heuristic pruning procedure tends to produce architectures that are inconsistent, i.e architectures where there is no path between the input layer and the output layer due to a too strong pruning.

CIFAR-10: Figure 2 (a) shows the performance on CIFAR for the Flop cost. One can see that, in comparison with Pruned CNF, our model obtains a better accuracy at any given cost level. The error only moves from 9.21 % to 13.66% while computing 98% less Flops. On the CIFAR-10 dataset,

Model	Flop gain	Error
CNF/BNF	0 %	0.39 %
CNF [18]	0 %	0.39 %
CKN [14]	-	0.39 %
MaxOut [5]	-	0.45 %
BSN	80.0 %	0.39 %
	90.0 %	0.49 %
	97.8 %	0.48 %
	100 %	89.72 %
P-CNF	20.0 %	0.45 %
	50.0 %	0.49 %
	79.4 %	2.59 %
	100 %	89.72 %

(a)

Model	Time (ms)	Error
CNF/BNF	68.12	0.39 %
CNF [18]	68.12	0.39 %
CKN [14]	-	0.39 %
MaxOut [5]	-	0.45 %
BSN	54.50	0.45 %
	34.06	0.42 %
	13.62	0.43 %
	6.81	0.45 %
	5.17	0.43 %
	0.0	89.72 %
P-CNF	54.50	0.54 %
	34.06	1.08 %
	23.23	1.97 %
	0.0	89.72 %

(b)

Table 1: (a) Performance using the Flop cost. Cost gain is the relative cost gain w.r.t the maximum possible cost (all edges) which value is 22 172 Flop on MNIST. Note that we report only the results of the most representative models (see supplementary material) (b) Performance using the Time cost. Time corresponds to the time (in ms) on CPU for one prediction.

the CNF pruning method tends to early remove unexpected edges, thus producing inconsistent architectures. To avoid this effect, we also compare the model with a 'Smart' pruning algorithm of our invention where the edges are removed by their inverse order of importance (the average value of the convolution absolute weights) as in [18], but if and only if the remaining architecture stays consistent. In that case, the Smart P-CNF model obtains an error of 12.8 % while BSN has a 11.03% error rate at the same cost level (cost gain = 90%). Here again, our algorithm outperforms this baseline. Note that like P-CNF, Smart P-CNF is also fine-tuned at each cost level, resulting in a very slow training procedure, BSN learning an order of magnitude faster. The BNF best learned architecture is illustrated in Figure 2 (b1) (for Flop gain = 98%).The majority of the expensive edges (top edges) are 'naturally' removed by our model since they correspond to a too low accuracy/cost trade-off.

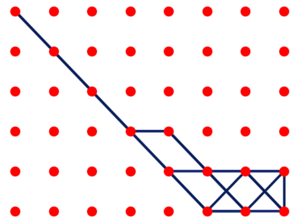
Other costs: To measure the ability of BNF to learn with complex costs, we train our model on architectures where one specific cost is associated to each edge, the overall cost being the sum of the costs of the edges used for prediction. Figure 3 (b) illustrates two examples of costs (left part), and the resulting learned architectures (Figure 3 (b) left) on MNIST. The first example (Figure 3-b1) is particularly interesting since it simulates an architecture where some edges are very expensive, it can for instance corresponds to a decentralized model that needs to exchange information on the Internet i.e expensive edges being communication edges. In that case, our model is able to find an architecture that minimizes the number of used expensive edges. In the random cost case (Figure 3-b2), our model is able to discover a complex structure also avoiding expensive edges.

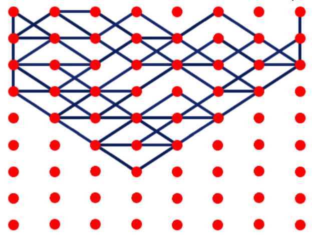
5.2 Image Segmentation

We also perform experiments on the image segmentation task. Note that, in that case, the model computes a map of pixel probabilities, the output layer being now located at the top-right position of the CNF matrix. It is thus more difficult to reduce the overall cost. On the Part Labels dataset, we are able to learn a BSN model with a Flop gain of 40%. In that task, forcing the model to increase the Flop gain by decreasing the value of **C** results in inconsistent models. At a Flop gain level of 40%, BSN obtains an error rate of 4.57%, which can be compared with the error of 4.94% for the full model. Due to its pruning procedure, the P-CNF model is unable to produce consistent architectures if the desired Flop gain is greater than 20%. Even at 20% its error rate is higher than ours (5.5%). The Smart P-CNF achieves higher Flop gains, but at the price of a clearly higher error rate. On this task, BNF also outperforms the baselines. The BNF best learned architecture is illustrated in Figure 2 (b2) and shows that BNF is able to learn an architecture adapted to the task.

Model	Flop gain	Error
CNF/BNF	0 %	7.99 %
CNF [18]	0 %	9.42 %
MaxOut [5]	0 %	9.38 %
Drop Con. [23]	0 %	9.32 %
BSN	20.0 %	9.21 %
	50.0 %	9.20 %
	80.0 %	10.18 %
	90.0 %	11.03 %
	97.0 %	11.58 %
	98.0 %	13.66 %
	>98 %	90 %
P-CNF	17.5 %	10.29 %
	28.1 %	19.90 %
	>28.1 %	90.00 %
Smart P-CNF	0.0 %	8.74 %
	50.0 %	11.41 %
	80.0 %	12.45 %
	96.8 %	13.03 %
	>96.8 %	90.00 %

(a)


(b1): Best CIFAR-10 Architecture (Flop cost)

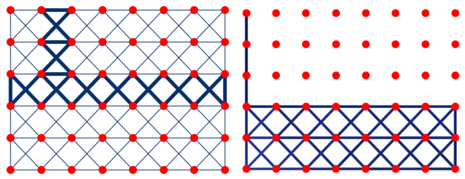

(b2): Best Part Labels Architecture (Flop cost)

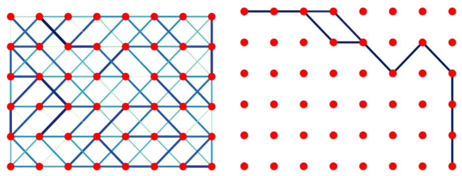
(b)

Figure 2: (a) Results on CIFAR-10 for the Flop cost. (b1) Best CIFAR-10 architecture. (b2) Best Part Labels architecture (the output layer is the top-right layer in that case)

Model	Flop gain	Error
CNF	0 %	4.94 %
CNF [18]	0 %	4.61 %
Liu et. al [13]	0 %	4.76 %
BSN	0.0 %	4.94 %
	20.0 %	4.79 %
	40.0 %	4.57 %
	>40 %	30.43 %
Smart P-CNF	20.0 %	5.50 %
	79.0 %	12.50 %
P-CNF	20.0 %	5.50 %
	>20.0 %	28.01 %

(a)


(b1) Simulating a distributed environment


(b2) Randomly sampled edges costs

(b)

Figure 3: (a) Results on Part Labels for the Flop cost. (b1) *left* represent the cost of each edge. *right* represents the learned architecture. (b2) *left* represent the cost of each edge (randomly sampled). *right* represents the learned architecture.

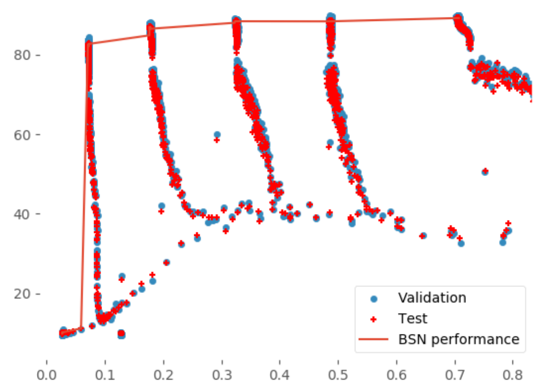
6 Conclusion and Perspectives

We proposed a new model called *Budgeted Super Network* able to automatically discover a time-efficient neural network architecture for a given problem by specifying a maximum authorized computation cost. The experiments in the computer vision domain show the effectiveness of our approach. Its main advantage is that BSN can be used for any computation costs (computation time, number of operations, extra-time occurred by some network latency,..). It can also be used with costs of another nature like memory consumption for example which has not been studied in this paper. This work opens many research directions. One direction is to evaluate the ability of this model to support inherent cost over a large distributed cluster, and thus to discover architectures that are efficient on large clusters. A second long-term direction is to study whether this model could be adapted in order to reduce the training time (instead of the test computation time). This could for example be done using meta-learning approaches.

Supplementary Material

Model Selection Protocol

As explained in the article, the selection of the model is obtained by learning many different models, by computing the Pareto front of the accuracy/cost curve on the validation set, and then by reporting the performance obtained on the test set. This is illustrated in the following figure where many different models are reported (blue circles) and the corresponding performance on test (red crosses). The line corresponds to the front obtained on the test models, this front being used to build the table of results presented in the article.



Demonstration of Proposition 1

Let us consider the stochastic optimization problem defined in Equation 4. The schema of the proof is the following:

- First, we lower bound the value of Equation 4 by the optimal value of Equation 3
- Then we show that this lower bound can be reached by some particular values of Γ and θ in Equation 4. Said otherwise, the solution of Equation 4 is equivalent to the solution of 3.

Let us denote:

$$B(H \odot E, \theta, \lambda) = \frac{1}{\ell} \sum_i \Delta(f(x^i, H \odot E, \theta), y^i) + \lambda \max(0, C(H \odot E) - C) \tag{9}$$

Given a value of Γ, let us denote $supp(\Gamma)$ all the H matrices that can be sampled following Γ. The objective function of Equation 4 can be written as:

$$\begin{aligned} E_{H\sim\Gamma}[B(H, E, \theta, \lambda)] &= \sum_{H \in supp(\Gamma)} B(H \odot E, \theta, \lambda) P(H|\Gamma) \\ &\geq \sum_{H \in supp(\Gamma)} B((H \odot E)^*, \theta^*, \lambda) P(H|\Gamma) \\ &= B((H \odot E)^*, \theta^*, \lambda) \end{aligned} \tag{10}$$

where $(H \odot E)^*$ and θ^* correspond to the solution of:

$$(H \odot E)^*, \theta^* = \arg\min_{H,\theta} B(H \odot E, \theta, \lambda) \tag{11}$$

Now, it is easy to show that this lower bound can be reached by considering a value of Γ^* such that $\forall H \in supp(\Gamma), H \odot E = (H \odot E)^*$. This corresponds to a value of Γ where all the probabilities associated to edges in E are equal to 0 or to 1.

Considering stochastic costs in the REINFORCE algorithm:

As explained previously, the proposed algorithm can also be used when the cost $C(H \odot E)$ is a stochastic function that depends of the environment e.g the network latency, (or even on the input data x). Our algorithm is still able to learn with such stochastic costs since the only change in the learning objective is that the expectation is now made on both H and C (and x if needed). This property is interesting since it allows to discover efficient architecture on stochastic operational infrastructure.

Additional Architecture Details

The network we use in our experiments is based on the dense Convolutional Neural Fabrics, which can be seen as a multi-layer and multi-scale convolutional neural network. As shown in Figure 1, this architecture spans on 2 axis: The first axis represents the different layers L of the network while the second axis corresponds to different scales S of output feature maps, the first scale being the size of the input images, each subsequent scale being of a size reduced by a factor of 2 up to the last scale corresponding to a single scalar.

Each layer (l, s) in this fabric takes its input from three different layer of the preceding column: One with a finer scale $(l - 1, s - 1)$, one with the same scale $(l - 1, s)$ and one with a coarser scale $(l - 1, s + 1)$.

The first and last columns are the only ones which have vertical connections within scales of the same layer (as can be seen in Figure 1). This is made to allow the propagation of the information to all nodes in the first column and to aggregate the activations of the last column to compute the final prediction. A more detailed description of this architecture can be found in the CNF original article.

Additional Learning Details

Datasets

MNIST. The MNIST dataset is composed of 70k images of 28x28 pixels representing a single handwritten digit between 0 and 9. The training set contains 60k training samples and the testing set contains 10k samples. We use the standard split of 50k training samples and 10k validation samples from the training set. Each image is first padded with zeros to a 36x36 images before applying a random 32x32 crop to obtain the final resolution. Images are normalized in the range [-1,1].

CIFAR10. The CIFAR10 dataset consists of 50k training images and 10k testing images with 10 classes. We split the training set following the standard, i.e 45k training samples and 5k validation samples. We use two data augmentation techniques : padding the image to 36x36 pixels before extracting a random crop of size 32x32 and horizontally flipping. Images are then normalized in the range [-1,1].

Part Labels. The Part Labels dataset is a subset of the LFW dataset composed of 2927 face images in which each pixel is labeled as one of the Hair/Skin/Background classes. The standard split contains 1500 training samples, 500 validation samples and 927 test samples. Images are zero padded from 250x250 to 256x256 and horizontally flipped before being normalized in the range [-1,1].

References

[1] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297, 2015.

[2] Ludovic Detnoyer and Patrick Gallinari. Deep sequential neural network. *CoRR*, abs/1410.0510, 2014.

[3] Jacob Devlin. Sharp Models on Dull Hardware: Fast and Accurate Neural Machine Translation Decoding on the CPU, 2017.

[4] Chrisantha Fernando, Dylan Banarse, Charles Blundell, Yori Zwols, David Ha, Andrei A. Rusu, Alexander Pritzel, and Daan Wierstra. Pathnet: Evolution channels gradient descent in super neural networks. *CoRR*, abs/1701.08734, 2017.

[5] Ian J. Goodfellow, David Warde-farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. In *In ICML*, 2013.

[6] Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. In *International Conference on Learning Representations (ICLR'16 best paper award)*, 2015.

[7] Babak Hassibi and David G. Stork. Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in Neural Information Processing Systems 5, [NIPS Conference]*, pages 164–171, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[8] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[9] Andrew Kae, Kihyuk Sohn, Honglak Lee, and Erik Learned-Miller. Augmenting CRFs with Boltzmann machine shape priors for image labeling. 2013.

[10] Alex Krizhevsky. Learning multiple layers of features from tiny images. Technical report, 2009.

[11] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, pages 306–351. IEEE Press, 2001.

[12] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. *CoRR*, abs/1701.00299, 2017.

[13] Sifei Liu, Jimei Yang, Chang Huang, and Ming-Hsuan Yang. Multi-objective convolutional learning for face labeling. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[14] Julien Mairal, Piotr Koniusz, Zaïd Harchaoui, and Cordelia Schmid. Convolutional kernel networks. *CoRR*, abs/1406.3332, 2014.

[15] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. *CoRR*, abs/1703.06217, 2017.

[16] Risto Miikkulainen, Jason Zhi Liang, Elliot Meyerson, Aditya Rawal, Dan Fink, Olivier Francon, Bala Raju, Hormoz Shahrzad, Arshak Navruzyan, Nigel Duffy, and Babak Hodjat. Evolving deep neural networks. *CoRR*, abs/1703.00548, 2017.

[17] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc V. Le, and Alex Kurakin. Large-scale evolution of image classifiers. *CoRR*, abs/1703.01041, 2017.

[18] Shreyas Saxena and Jakob Verbeek. Convolutional neural fabrics. *CoRR*, abs/1606.02492, 2016.

[19] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *CoRR*, abs/1505.00387, 2015.

[20] Kenneth O. Stanley and Risto Miikkulainen. Efficient reinforcement learning through evolving neural network topologies. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, New York, USA, 9-13 July 2002*, pages 569–577, 2002.

[21] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[22] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on cpus. In *Deep Learning and Unsupervised Feature Learning Workshop, NIPS 2011*, 2011.

[23] Li Wan, Matthew D. Zeiler, Sixin Zhang, Yann LeCun, and Rob Fergus. Regularization of neural networks using dropconnect. In *ICML (3)*, volume 28 of *JMLR Proceedings*, pages 1058–1066. JMLR.org, 2013.

[24] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.