HardCoRe-NAS: Hard Constrained Differentiable Neural Architecture Search

Niv Nayman, Yonathan Aflalo, Asaf Noy, Lihi Zelnik-Manor

Abstract

Realistic use of neural networks often requires adhering to multiple constraints on latency, energy and memory among others. A popular approach to find fitting networks is through constrained Neural Architecture Search (NAS), however, previous methods enforce the constraint only softly. Therefore, the resulting networks do not exactly adhere to the resource constraint and their accuracy is harmed. In this work we resolve this by introducing Hard Constrained Differentiable NAS (HardCoRe-NAS), that is based on an accurate formulation of the expected resource requirement and a scalable search method that satisfies the hard constraint throughout the search. Our experiments show that HardCoRe-NAS generates state-of-the-art architectures, surpassing other NAS methods, while strictly satisfying the hard resource constraints without any tuning required.

1. Introduction

With the rise in popularity of Convolutional Neural Networks (CNN), the need for neural networks with fast inference speed and high accuracy, has been growing continuously. At first, manually designed architectures, such as VGG (Simonyan & Zisserman, 2015) or ResNet (He et al., 2015), targeted powerful GPUs as those were the common computing platform for deep CNNs. Many variants of those architectures were the golden standard until the need for deployment on edge devices and standard CPUs emerged. These are more limited computing platforms, requiring lighter architectures that for practical scenarios have to comply with hard constraints on the real time latency or power consumption. This has spawned a line of research aimed at finding architectures with both high performance and bounded resource demand.

The main approaches to solve this evolved from Neural Architecture Search (NAS) (Zoph & Le, 2016; Liu et al., 2018; Cai et al., 2018), while adding a constraint on the target latency over various platforms, e.g., TPU, CPU, Edge-GPU, FPGA, etc. The constrained-NAS methods can be grouped into two categories: (i) Reward based methods such as Reinforcement-Learning (RL) or Evolutionary Algorithm (EA) (Cai et al., 2019; Tan et al., 2019; Tan & Le, 2019; Howard et al., 2019), where the search is performed by sampling networks and predicting their final accuracy and resource demands by evaluation over some validation set. The predictors are expensive to acquire and oftentimes inaccurate. (ii) Resource-aware gradient based methods (Hu et al., 2020; Wu et al., 2019) formulate a differentiable loss function consisting of a trade-off between an accuracy term and a proxy soft penalty term. Therefore, the architecture can be directly optimized using stochastic gradient descent (SGD) (Bottou, 1998), however, it is hard to tune the trade-off between accuracy and resources, which deteriorates the network accuracy and fails to fully meet the resource re-
quirements. The hard constraints over the resources are further violated due to a final discretization step projecting the architecture over the differentiable search space into the discrete space of architectures.

In this paper, we propose a search algorithm that produces architectures with high accuracy (Figure 1) that strictly satisfy any given hard latency constraint (Figure 3). The search algorithm is fast and scalable to a large number of platforms. The proposed algorithm is based on several key ideas, starting from formulating the NAS problem more accurately, accounting for hard constraints over resources, and solving every aspect of it rigorously. For clarity we focus in this paper on latency constraints, however, our approach can be generalized to other types of resources.

At the heart of our approach lies a suggested differentiable search space that induces a one-shot model (Bender et al., 2018; Chu et al., 2019; Guo et al., 2020; Cai et al., 2019) that is easy to train via a simple, yet effective, technique for sampling multiple sub-networks from the one-shot model, such that each one is properly pretrained. We further suggest an accurate formula for the expected latency of every architecture residing in that space. Then, we search the space for sub-networks by solving a hard constrained optimization problem while keeping the one-shot model pretrained weights frozen. We show that the constrained optimization can be solved via the block coordinate stochastic Frank-Wolfe (BC-SFW) algorithm (Hazan & Luo, 2016a; Lacoste-Julien et al., 2013a). Our algorithm converges faster than SGD, while tightly satisfying the hard latency constraint continuously throughout the search, including during the final discretization step.

The approach we propose has several advantages. First, the outcome networks provide high accuracy and closely comply to the latency constraint. In addition, our solution is scalable to multiple target devices and latency demands. This scalability is due to the efficient pretraining of the one-shot model as well as the fast search method that involves a relatively small number of parameters, governing only the structure of the architecture. We hope that our formulation of NAS as a constrained optimization problem, equipped with an efficient algorithm that solves it, could give rise to followup work incorporating a variety of resource and structural constraints over the search space.

2. Related Work

Efficient Neural Networks are designed to meet the rising demand of deep learning models for numerous tasks per hardware constraints. Manually-crafted architectures such as MobileNets (Howard et al., 2017; Sandler et al., 2018b) and ShuffleNet (Zhang et al., 2018) were designed for mobile devices, while TResNet (Ridnik et al., 2020) and ResNesT (Zhang et al., 2020a) are tailor-made for GPUs. Techniques for improving efficiency include pruning of redundant channels (Dong & Yang, 2019; Aflalo et al., 2020) and layers (Han et al., 2015b), model compression (Han et al., 2015a; He et al., 2018) and weight quantization methods (Hubara et al., 2016; Umuroglu et al., 2017). Dynamic neural networks adjust models based on their inputs to accelerate the inference, via gating modules (Wang et al., 2018), graph branching (Huang et al., 2017) or dynamic channel selection (Lin et al., 2017). These techniques are applied on predefined architectures, hence cannot utilize or satisfy specific hardware constraints.

Neural Architecture Search methods automate models’ design per provided constraints. Early methods like NASNet (Zoph & Le, 2016) and AmoebaNet (Real et al., 2019) focused solely on accuracy, producing SotA classification models (Huang et al., 2019) at the cost of GPU-years per search, with relatively large inference times. DARTS (Liu et al., 2018) introduced a differential space for efficient search and reduced the training duration to days, followed by XNAS (Nayman et al., 2019) and ASAP (Noy et al., 2020) that applied pruning-during-search techniques to further reduce it to hours. Recent methods (Lee et al., 2021) also attempt at rapidly generating architectures for unseen datasets without considering resource constraints. Hardware-aware methods such as ProxylessNAS (Cai et al., 2018), MnasNet (Tan et al., 2019), FBNet (Wu et al., 2019), SPNAS-Net (Stamoulis et al., 2019) and TFNAS (Hu et al., 2020) produce architectures that satisfy the required constraints by applying simple heuristics such as soft penalties on the loss function. OFA (Cai et al., 2019) proposed a scalable approach across multiple devices by training an one-shot model (Brock et al., 2017; Bender et al., 2018) for 1200 GPU hours. This provides a strong pretrained super-network being highly predictive for the accuracy ranking of extracted sub-networks (Guo et al., 2020; Chu et al., 2019; Yu et al., 2020). OFA searches under latency constraints by conducting evolutionary search (Real et al., 2019) over an accuracy predictor learnt by sampling 16K sub-networks and measuring their accuracy on 10K validation images, hence, training the predictor costs 40 GPU hours. This work relies on such one-shot model acquired within only 400 GPU hours in a much simpler manner, without training an accuracy predictor and satisfies hard constraints tightly with less heuristics.

Frank-Wolfe (FW) algorithm (Frank et al., 1956) is commonly used by machine learning applications (Sun et al., 2019) thanks to its projection-free property (Combettes et al., 2020; Hazan & Minasyan, 2020) and ability to handle structured constraints (Jaggi, 2013). Modern adaptations aimed at deep neural networks (DNNs) optimization include more efficient variants (Zhang et al., 2020b; Combettes et al., 2020), task-specific variants (Chen et al., 2020; Tsiligkaridis & Roberts, 2020), as well as improved
convergence guarantees (Lacoste-Julien & Jaggi, 2015; d’Aspremont & Pilanci, 2020). Two prominent variants are the stochastic FW (Hazan & Luo, 2016b) and Block-coordinate FW (Lacoste-Julien et al., 2013b). While FW excels as an optimizer for DNNs (Berrada et al., 2018; Pokutta et al., 2020), this work is the first to utilize it for NAS.

3. Method

In the scope of this paper, we focus on latency-constrained NAS, searching for an architecture with the highest validation accuracy under a predefined latency constraint, denoted by \( T \). Our architecture search space \( S \) is parametrized by \( \zeta \in \mathcal{S} \), governing the architecture structure in a fully differentiable manner, and \( w \), the convolution weights. A latency-constrained NAS can be expressed as the following constrained bilevel optimization problem:

\[
\min_{\zeta \in \mathcal{S}} \mathbb{E} (x, y \sim D_{\text{val}}) \left[ L_{CE}(x, y \mid w^*, \tilde{\zeta}) \right] \\
\text{s.t.} \; \text{LAT}(\zeta) \leq T \\
\quad w^* = \arg\min_w \mathbb{E} (x, y \sim D_{\text{train}}) \left[ L_{CE}(x, y \mid w, \tilde{\zeta}) \right] \\
\quad \tilde{\zeta} \sim \mathcal{P}_\zeta(\mathcal{S})
\]

(1)

where \( D_{\text{train}} \) and \( D_{\text{val}} \) are the train and validation sets’ distributions respectively, \( \mathcal{P}_\zeta(\mathcal{S}) \) is some probability measure over the search space parameterized by \( \zeta \), \( L_{CE} \) is the cross entropy loss as a differentiable proxy for the negative accuracy and \( \text{LAT}(\zeta) \) is the estimated latency of the model.

To solve problem (1), we construct a fully differentiable search space parameterized by \( \zeta = (\alpha, \beta) \) (Section 3.1), that enables the formulation of a differentiable closed form formula expressing the estimated latency \( \text{LAT}(\alpha, \beta) \) (Section 3.2) and efficient acquisition of \( w^* \) (Section 3.3). Finally, we introduce rigorous constrained optimization techniques for solving the outer level problem (Section 3.4).

3.1. Search Space

Aiming at the most accurate latency, a flexible search space is composed of a micro search space that controls the internal structures of each block \( b \in \{1, \ldots, d = 4\} \), together with a macro search space that specifies the way those are connected to one another in every stage \( s \in \{1, \ldots, S = 5\} \).

3.1.1. Micro-Search

Every block is an elastic version of the MBInvRes block, introduced in (Sandler et al., 2018a), with expansion ratio \( er \in \mathcal{A}_{er} = \{3, 4, 6\} \) of the point-wise convolution, kernel size \( k \in \mathcal{A}_{k} = \{3 \times 3, 5 \times 5\} \) of the depth-wise separable convolution (DWS), and Squeeze-and-Excitation (SE) layer (Hu et al., 2018) \( se \in \mathcal{A}_{se} = \{\text{on, off}\} \). The blocks are configurable, as illustrated at the bottom of Figure 2, using a parametrization \( (\alpha^s_{b,er}, \alpha^s_{b,k}, \alpha^s_{b,se}) \), defined for every block \( b \) of stage \( s \):

\[
\alpha^s_{b,er} \in \{0, 1\} \quad \forall er \in \mathcal{A}_{er}, k \in \mathcal{A}_{k}, se \in \mathcal{A}_{se} \\
\Sigma_{er \in \mathcal{A}_{er}} \alpha^s_{b,er} = 1 ; \Sigma_{k \in \mathcal{A}_{k}} \alpha^s_{b,k} = 1 ; \Sigma_{se \in \mathcal{A}_{se}} \alpha^s_{b,se} = 1
\]

Each triplet \((er, k, se)\) induces a block configuration \( c \) that resides within a micro-search space \( \mathcal{C} = \mathcal{A}_{er} \otimes \mathcal{A}_{k} \otimes \mathcal{A}_{se} \), parameterized by \( \alpha \in \mathcal{A} = \otimes_{s=1}^S \otimes_{b=1}^d \otimes_{c=0}^{\mathcal{C}} \alpha^s_{b,c} \), where \( \otimes \) denotes the Cartesian product. Hence, for each block \( b \) of stage \( s \) we have:

\[
\alpha^s_{b,c} \in \{0, 1\}^{\mathcal{C}} ; \sum_{c \in \mathcal{C}} \alpha^s_{b,c} = 1
\]

An input feature map \( x^s_b \) to block \( b \) of stage \( s \) is processed as follows:

\[
x^s_{b+1} = \sum_{c \in \mathcal{C}} \alpha^s_{b,c} \cdot O^s_{b,c}(x^s_b)
\]

where \( O^s_{b,c}(\cdot) \) is the operation performed by the elastic MBInvRes block configured according to \( c = (er, k, se) \).

Having several configurations \( O^s_{b,c}(\cdot) \) share the same values of \( \alpha^s_{b,er} \) or \( \alpha^s_{b,k} \) or \( \alpha^s_{b,se} \) induces weight sharing between the common operations of the associated architectures. This weight sharing is beneficial for solving the inner problem (1) effectively, as will be discussed in Section 3.3.

3.1.2. Macro-Search

Inspired by (Hu et al., 2020; Cai et al., 2019), the output of each block of every stage is also directed to the end of the stage as illustrated in the middle of Figure 2. Thus, the depth of each stage \( s \) is controlled by the parameters \( \beta \in \mathcal{B} = \otimes_{s=1}^S \otimes_{b=1}^d \beta^s_{b} \), such that:

\[
\beta^s_{b} \in \{0, 1\}^d ; \sum_{b=1}^S \beta^s_{b} = 1
\]
The depth is \( b^* \in \{ b \mid \beta^*_b = 1, b \in \{ 1, \ldots, d \} \} \), since:

\[
x_{t+1}^{b} = \sum_{b=1}^{d} \beta^*_b \cdot x_{b+1}
\]

### 3.1.3. The Composed Search Space

The overall search space is composed of both the micro and macro search spaces parameterized by \( \alpha \in \mathcal{A} \) and \( \beta \in \mathcal{B} \), respectively, such that:

\[
S = \left\{ (\alpha, \beta) \mid \begin{align*}
\alpha^*_b \in \{0, 1\}^{|C|} ; & \sum_{c \in C} \alpha^*_b = 1 \\
\beta^*_b \in \{0, 1\}^d ; & \sum_{b=1}^{d} \beta^*_b = 1 \\
& \forall s \in \{1, \ldots, S\}, b \in \{1, \ldots, d\}, c \in C
\end{align*} \right\}
\]

A continuous probability distribution is induced over the space, by relaxing \( \alpha^*_b, c \in \{0, 1\}^{|C|} \rightarrow \alpha^*_b, c \in \mathbb{R}^{|C|} \) and \( \beta^*_b \in \{0, 1\}^d \rightarrow \beta^*_b \in \mathbb{R}^d_+ \) to be continuous rather than discrete. A sample sub-network is drawn \( \hat{\zeta} = (\hat{\alpha}, \hat{\beta}) \sim \mathcal{P}_c(S) \) using the Gumbel-Softmax Trick (Jang et al., 2016) such that \( \hat{\zeta} \in S \), as specified in (4) and (5). In summary, one can view the parametrization \( (\alpha, \beta) \) as a composition of probabilities in \( \mathcal{P}_c(S) \) or as degenerated one-hot vectors in \( S \).

Effectively we include at least a couple of blocks in each stage by setting \( \beta^*_1 = 0 \), hence, the overall size of the search space is:

\[
|S| = \left( \sum_{b=2}^{d} |C|^b \right)^S = \left( \sum_{b=2}^{d} |A_{x=b}^b| \cdot |A_k^b| \cdot |A_{c=b}^b| \right)^S
\]

\[
= ((3 \times 2)^2 + (3 \times 2 \times 3)^2 + (3 \times 2 \times 2)^4)^5 \approx 10^{27}
\]

### 3.2. Formulating the Latency Constraint

Aiming at tightly satisfying latency constraints, we propose an accurate formula for the expected latency of a sub-network. The expected latency of a block \( 0 \) can be computed by summing over the latency \( t_{b,c} \) of every possible configuration \( c \in A \):

\[
\bar{\ell}^b = \sum_{c \in A} \alpha^*_b, c \cdot t_{b,c}
\]

Thus the expected latency of a stage of depth \( b' \) is

\[
\bar{\ell}^{b'} = \sum_{b=1}^{b'} \bar{\ell}^b
\]

Taking the expectation over all possible depths yields

\[
\bar{\ell}^b = \sum_{b'=1}^{d} \beta^*_b, b' \cdot \bar{\ell}^{b'}
\]

and summing over all the stages results in the following formula for the overall latency:

\[
LAT(\alpha, \beta) = \sum_{s=1}^{S} \sum_{b'=1}^{d} \sum_{b=1}^{b'} \alpha^*_b, c \cdot t_{b,c} \cdot \beta^*_b, b'
\]

### 3.3. Solution to the Inner Problem \( w^* \)

Previous work proposed approximated solutions to the following unconstrained problem:

\[
\min_{\zeta \in S} \mathbb{E} \left[ \mathcal{L}_{CE} (x, y \mid w^*, \zeta) \right]
\]

s.t. \( w^* = \arg\min_w \mathbb{E}_{\zeta \sim \mathcal{P}_c(S)} \left[ \mathcal{L}_{CE} (x, y \mid w, \zeta) \right] \)

typically by alternating or simultaneous updates of \( w \) and \( \zeta \) (Liu et al., 2018; Xie et al., 2018; Cai et al., 2018; Wu et al., 2019; Hu et al., 2020). This approach has several limitations. First, obtaining a designated \( w^*_\zeta \) with respect to every update of \( \zeta \) involves a heavy training of a neural network until convergence. Instead a very rough approximation is obtained by just a few update steps for \( w \). In turn, this approximation creates a bias towards strengthening networks with few parameters since those learn faster, hence, get sampled even more frequently, further increasing the chance to learn in a positive feedback manner. Eventually, often overly simple architectures are generated, e.g., consisting of many skip-connections (Chen et al., 2019; Liang et al., 2019). Several remedies have been proposed, e.g.,

![Figure 3. Experimental results showing that the latency calculated with formula (3) is very close to the latency measured in practice, on both CPU and GPU.](image-url)
temperature annealing, adding uniform sampling, modified gradients and updating only $w$ for a while before the joint optimization begins (Noy et al., 2020; Wu et al., 2019; Hu et al., 2020; Nayman et al., 2019). While those mitigate the bias problem, they do not solve it.

We avoid this approximation. Instead we obtain $w^*$ of the inner problem of (1) only once, with respect to a uniformly distributed architecture, sampling $\zeta$ from $P_\zeta(S) = U(S)$.

This is done by sampling multiple distinctive paths (sub-networks of the one-shot model) for every image in the batch in an efficient way (just a few lines of code), using the Gumbel-Softmax Trick, as opposed to the common sampling of a single path per batch. For every feature map $x$ that goes through block $b$ of stage $s$, distinctive uniform random variables $U_{b,c,x}^s; U_{b,c,x}^s \sim U(0, 1)$ are sampled, governing the path undertaken by this feature map:

$$\alpha_{b,c,x}^s = e^{\log(\alpha_{b,c,x}^s) - \log(\log(U_{b,c,x}^s))}$$

$$\beta_{b,c,x}^s = \frac{\sum_{c \in C} e^{\log(\beta_{b,c,x}^s) - \log(\log(U_{b,c,x}^s))}}{\sum_{c \in C} e^{\log(\alpha_{b,c,x}^s) - \log(\log(U_{b,c,x}^s))}}$$

Based on the observation that the accuracy of a sub-network with $w^*$ should be predictive for its accuracy when optimized as a stand-alone model from scratch, we aim at an accurate prediction. Our simple approach implies that, with high probability, the number of paths sampled at each update step is as high as the number of images in the batch. This is two orders of magnitude larger than previous methods that sample a single path per update step (Guo et al., 2020; Cai et al., 2019), while avoiding the need to keep track of all the sampled paths (Chu et al., 2019). Using multiple paths reduces the variance of the gradients with respect to the paths sampled by an order of magnitude. Furthermore, leveraging the weight sharing implied by the structure of the elastic MBInvRes block (Section 3.1.1), the number of gradients observed by each operation is increased by a factor of at least $\frac{|C|}{\max(|A_{cell}|, |A_b|, |A_{cell} |)} = \frac{3 \times 2 \times 2}{\max(3, 2, 2)} \approx 4$. This further reduces the variance by half.

Figure 4 shows that we obtain a one-shot model $w^*$ with high correlation between the ranking of sub-networks directly extracted from it and the corresponding stand-alone counterpart trained from scratch. See more details in Section 4.2.2. This implies that $w^*$ captures well the quality of sub-structures in the search space.

Having defined the formula for the latency $\text{LAT}^*(\zeta)$ and obtained a solution for $w^*$, we can now continue to solve the outer problem (1).

### 3.4. Solving the Outer Problem

#### 3.4.1. Searching Under Latency Constraints

Most differentiable resource-aware NAS methods account for the resources through shaping the loss function with soft penalties (Wu et al., 2019; Hu et al., 2020). This approach solely does not meet the constraints tightly. Experiments illustrating this are described in Section 4.2.3.

Our approach directly solves the constrained outer problem (1), hence, it enables the strict satisfaction of resource constraints by further restricting the search space, i.e., $\mathcal{S}_{LAT} = \{ \zeta \mid \zeta \in \mathcal{P}(\zeta), \text{LAT}(\zeta) \leq T \}$.

As commonly done for gradient based approaches, e.g., (Liu et al., 2018), we relax the discrete search space $\mathcal{S}$ to be continuous by searching for $\zeta \in \mathcal{S}_{LAT}$. As long as $\mathcal{S}_{LAT}$ is convex, it could be leveraged for applying the stochastic Frank-Wolfe (SFW) algorithm (Hazan & Luo, 2016a) to directly solve the constrained outer problem:

$$\min_{\zeta \in \mathcal{S}_{LAT}} \mathbb{E}_{x,y \sim \mathcal{D}_{val}} [\mathcal{L}_{CE}(x, y \mid w^*, \zeta)]$$

following the update step:

$$\zeta_{t+1} = (1 - \gamma_t) \cdot \zeta_t + \gamma_t \cdot \xi_t$$

$$\xi_t = \arg\min_{\zeta \in \mathcal{S}_{LAT}} \mathbb{E}_{x \sim \mathcal{D}_{train}} [\mathcal{L}_{CE}(x, y \mid w^*, \zeta)]$$

where $(x_t, y_t)$ and $\gamma_t$ are the sampled data and the learning rate at step $t$, respectively. For $\mathcal{S}_{LAT}$ of linear constraints, the linear program (8) can be solved efficiently, using the Simplex algorithm (Nash, 2000).
A convex $S_{LAT}$ together with $\gamma_t \in [0, 1]$ satisfy $\zeta_t \in S_{LAT}$ anytime, as long as $\zeta_0 \in S_{LAT}$. We provide a method for satisfying the latter in the supplementary materials.

The benefits of such optimization are demonstrated in Figure 5 through a toy problem, described in Section 4.2.3. While SGD is sensitive to the trade-off involved with a soft penalty, SFW converges faster to the optimum with zero penalty.

![Figure 5. Objective loss and soft penalty for BCSFW and SGD for different values of the penalty coefficient $\lambda$. Solid and dashed curves represent the objective loss (left y-axis) and soft penalty (right y-axis), respectively. BCSFW converges to the optimum much faster than SGD.](image)

All this requires $S_{LAT}$ to be convex. While $P_\zeta(S)$ is obviously convex, formed by linear constraints, the latency constraint $LAT(\zeta) \leq T$ is not necessarily so. The latency formula (3) can be expressed as a quadratic constraint by constructing a matrix $\Theta \in \mathbb{R}_+^{S \times d} \times |A| \times S \times d$ from $t_{b,c}$, such that,

$$LAT(\zeta) = LAT(\alpha, \beta) = \alpha^T \Theta \beta \quad ; \quad \zeta \in P_\zeta(S) \quad (9)$$

Since $\Theta$ is constructed from measured latency, it is not guaranteed to be positive semi-definite, hence, the induced quadratic constraint could make $S_{LAT}$ non-convex.

To overcome this, we introduce the Block Coordinate Stochastic Frank-Wolfe (BCSFW) Algorithm 1, that combines Stochastic Frank-Wolfe with Block Coordinate Frank-Wolfe (Lacoste-Julien et al., 2013a). This is done by forming separated convex feasible sets at each step, induced by linear constraints only:

$$\alpha_t \in S_{\alpha}^* = \{\alpha \mid \alpha \in A, \beta^T \Theta \alpha \leq T\} \quad (10)$$

$$\beta_t \in S_{\beta}^* = \{\beta \mid \beta \in B, \alpha^T \Theta \beta \leq T\} \quad (11)$$

This implies that $\zeta_t = (\alpha_t, \beta_t) \in S_{LAT}$ for all $t$. Moving inside the feasible domain at anytime avoids irrelevant infeasible structures from being promoted and hiding feasible structures.

3.4.2. Projection Back to the Discrete Space

As differentiable NAS methods are inherently associated with a continuous search space, a final discretization step $P$:

**Algorithm 1 Block Coordinate SFW (BCSFW)**

input $(\alpha_0, \beta_0) \in S_{LAT}$

for $t = 0, \ldots, K$ do

- Pick $\delta := \alpha$ or $\delta := \beta$ at random
- Sample an i.i.d validation batch $(x_t, y_t) \sim D_{val}$
- $\xi_t = \arg\min_{\xi \in S_{\xi}} \xi^T \cdot \nabla_{\delta_t} \mathcal{L}_{CE}(x_t, y_t \mid w^*, \delta_t)$
- Update $\delta_{t+1} = (1 - \gamma_t) \cdot \delta_t + \gamma_t \cdot \xi_t$ with $\gamma_t = \frac{4}{t+4}$

end for

$P_\zeta(S) \to S$ is required for extracting a single architecture. Most methods use the argmax operator:

$$\hat{\alpha}_{t,c}^* := 1 \{ c = \arg\max_{c \in C} \alpha_{t,c}^* \} \quad (12)$$

$$\hat{\beta}_{b}^* := 1 \{ b = \arg\max_{b \in \{2, \ldots, d\}} \beta_{b}^* \}$$

for all $s \in \{1, \ldots, S\}, b \in \{1, \ldots, d\}, c \in C$, where $(\hat{\alpha}, \hat{\beta})$ is the solution to the outer problem of (1).

For resource-aware NAS methods, applying such projection results in possible violation of the resource constraints, due to the shift from the converged solution in the continuous space. Experiments showing that latency constraints are violated due to (12) are provided in Section 4.2.4.

While several methods mitigate this violation by promoting sparse probabilities during the search, e.g., (Noy et al., 2020; Nayman et al., 2019), our approach completely eliminates it by introducing an alternative projection step, described next.

Viewing the solution of the outer problem $(\hat{\alpha}^*, \hat{\beta}^*)$ as the credit assigned to each configuration, we introduce a projection step that maximizes the overall credit while strictly satisfying the latency constraints. It is based on solving the following two linear programs:

$$\max_{\alpha \in S_{\alpha}^*} \alpha^T \cdot \alpha^* \quad ; \quad \max_{\beta \in S_{\beta}^*} \beta^T \cdot \beta^* \quad (13)$$

Note, that when there is no latency constraint, e.g., $T \to \infty$, (13) coincides with (12).

We next provide a theorem that guarantees that the projection (13) yields a sparse solution, representing a valid sub-network of the one-shot model. Specifically, a single probability vector from those composing $\alpha$ and $\beta$ contains up to two non-zero entries each, as all the rest are one-hot vectors.

**Theorem 3.1.** The solution $(\alpha, \beta)$ of (13) admits:

$$\sum_{c \in C \backslash d} |\alpha_{b,c}^*|^0 = 1 \forall (s, b) \in \{1, \ldots, S\} \times \{1, \ldots, d\} \setminus \{(s_{\alpha}, b_{\alpha})\}$$

$$\sum_{b=1}^{d} |\beta_{b}^*|^0 = 1 \quad \forall s \in \{1, \ldots, S\} \setminus \{s_{\beta}\}$$

where $|\cdot|^0 = 1\{\cdot > 0\}$ and $(s_{\alpha}, b_{\alpha}), s_{\beta}$ are single block
and stage respectively, satisfying:

$$\sum_{c \in C} |\alpha^{a,b}_{c, b}|^2 \leq 2 \quad ; \quad \sum_{b=1}^{d} |\beta^{a,b}_b|^2 \leq 2 \quad \text{(14)}$$

Refer to the supplementary materials for the proof.

**Remark:** A negligible deviation is associated with taking the argmax (12) over the only two couples referred to in (14). Experiments supporting this are described in Section 4.2.4.

## 4. Experimental Results

### 4.1. Search for State-of-the-Art Architectures

#### 4.1.1. Dataset and Setting

For all of our experiments, we train our networks using SGD with a learning rate of 0.1 with cosine annealing, Nesterov momentum of 0.9, weight decay of $10^{-4}$, applying label smoothing (Szegedy et al., 2016) of 0.1, cutout, Autoaugment (Cubuk et al., 2018), mixed precision and EMA-smoothing.

We obtain the solution of the inner problem $w^*$ as specified in sections 3.3 and 4.2.2 over 80% of a random 80-20 split of the ImageNet train set. We utilize the remaining 20% as a validation set and search for architectures with latencies of 40, 45, 50, 55, 60 and 25, 30, 40 milliseconds running with a batch size of 1 and 64 on an Intel Xeon CPU and and NVIDIA P100 GPU, respectively. The search is performed according to section 3.4 for only 2 epochs of the validation set, lasting for 8 GPU hours.$^5$

#### 4.1.2. Comparisons with Other Methods

We compare our generated architectures to other state-of-the-art NAS methods in Table 1 and Figure 1. For the purpose of comparing the generated architectures alone, excluding the contribution of evolved pretraining techniques, for each model in Table 1, the official PyTorch implementation (Paszke et al., 2019) is trained from a random initialization (but OFA $^6$) using the exact same techniques and code, as specified in section 4.1.1. We report the maximum accuracy between the original paper and our training. We emphasize that all latency values presented are measured without any formula but through actual time measurements of the models, running on a single thread with the exact same settings and on the same hardware. We excluded further optimizations, such as Intel MKL-DNN (Intel, R), therefore, the latency we report may differ from the one originally reported. It can be seen that networks generated by our method meet the latency target closely, while at the same time surpassing all the others methods on the top-1 accuracy over ImageNet with a reduced scalable search time.

---

$^5$Finetuning a model obtained by 1200 GPU hours.

---

### 4.2. Empirical Analysis of Key Components

#### 4.2.1. Validation of the Latency Formula

One of our goals is to provide a practical method to accurately meet the given resource requirements. Hence, we validate empirically the accuracy of the latency formula (3), by comparing its estimation with the measured latency. Experiments were performed on two platforms: Intel Xeon CPU and NVIDIA P100 GPU, and applied to multiple networks. Results are shown in Figure 3, which confirms a linear relation between estimated and measured latency, with a ratio of 1.003 and a coefficient of determination of $R^2 = 0.99$. This supports the accuracy of the proposed formula.

#### 4.2.2. Evaluating the Solution of the Inner Problem $w^*$

The ultimate quality measure for a generated architecture is arguably its accuracy over a test set when trained as a stand-alone model from randomly initialized weights. To evaluate the quality of our one-shot model $w^*$ we compare...
the accuracy of networks extracted from it with the accuracy of the corresponding architectures when trained from scratch. Naturally, when training from scratch the accuracy could increase. However, a desired behavior is that the ranking of the accuracy of the networks will remain the same with and without training from scratch. The correlation can be calculated via the Kendall-Tau (Maurice, 1938) and Spearman’s (Spearman, 1961) rank correlation coefficients, denoted as $\tau$ and $\rho$, respectively.

To this end, we first train for 250 epochs a one-shot model $\tilde{w}$ using the heaviest possible configuration, i.e., a depth of 4 for all stages, with $er = 6, k = 5 \times 5, se = on$ for all the blocks. Next, to obtain $w^*$, we apply the multi-path training of Section 3.3 for additional 100 epochs of fine-tuning $\tilde{w}$ over 80% of a 80-20 random split of the ImageNet train set (Deng et al., 2009). The training settings are specified in Section 4.1.1. The first 250 epochs took 280 GPU hours\(^4\) and the additional 100 fine-tuning epochs took 120 GPU hours\(^5\), summing to a total of 400 hours on NVIDIA V100 GPU to obtain $w^*$. To further demonstrate the effectiveness of our multi-path technique, we repeat this procedure also without it, sampling a single path for each batch.

For the evaluation of the ranking correlations, we extract 18 sub-networks of common configurations for all stages of depths in $\{2, 3, 4\}$ and blocks with an expansion ratio in $A_{cr} = \{3, 4, 6\}$, a kernel size in $A_k = \{3 \times 3, 5 \times 5\}$ and Squeeze and Excitation being applied. We train each of those as stand-alone from random initialized weights for 200 epochs over the full ImageNet train set, and extract their final top-1 accuracy over the validation set of ImageNet.

Figure 4 shows for each extracted sub-network its accuracy within a small marginal cost of 15 GPU hours. The total cost for generating $N$ trained models is $400 + 15N$, much lower than the $1200 + 25N$ reported by OFA (Cai et al., 2019). See Table 1. This makes our method scalable for many devices and latency requirements. Note, that allowing for longer training further improves the accuracy significantly.

4.2.3. OUTER PROBLEM: HARD vs SOFT

Next, we evaluate our method’s ability to satisfy accurately a given latency constraint. We compare our hard-constrained formulation (1) with the common approach of adding soft penalties to the loss function (Hu et al., 2020; Wu et al., 2019). The experiments were performed over a simple and intuitive stochastic toy problem:

$$\min_x \Sigma^d_{i=1} (1 + \epsilon_i) \cdot x^2_i \quad \text{s.t.} \quad \Sigma^d_{i=1} x_i = 1 ; \quad x, \epsilon \in \mathbb{R}^d$$

(15)

where $\epsilon \sim \mathcal{N}(0, 0.01I)$. This problem can be solved using the Frank-Wolfe (FW) (Frank et al., 1956) update rule:

$$x_{t+1} = (1 - \gamma_t) \cdot x_t + \gamma_t \cdot \xi_t$$

(16)

starting from an arbitrary random feasible point, e.g. sample a random vector and normalize it. The soft-constraint approach minimizes $\Sigma^d_{i=1} (1 + \epsilon_i) \cdot x^2_i + \lambda \left(\Sigma^d_{i=1} x_i - 1\right)^2$ using stochastic gradient descent (SGD), where $\lambda$ is a coefficient that controls the trade-off between the objective function and the soft penalty representing the constraint. Instead of using FW, we implement a version of Block Coordinate Stochastic Frank-Wolfe (BCSFW) by randomly sampling one of two predefined blocks of coordinates at every iteration and applying (16) for the sampled block only.

Figure 5, shows the objective value for $d = 10$ and the corresponding soft penalty value along the optimization for both BCSFW and SGD with several values of $\lambda$. It can be seen that SGD is very sensitive to the trade-off tuned by $\lambda$, often violating the constraint or converging to a sub-optimal objective value. On the contrary, BCSFW converges faster to the optimal solution while strictly satisfying the constraint throughout the optimization.

4.2.4. EVALUATING THE DISCRETIZING PROJECTION

Table 2 evaluates the projection of architectures to the discrete space, as proposed in Section 3.4.2. While the commonly used argmax projection violates the constraints by up to 10%, those are strictly satisfied by our proposed projection.

<table>
<thead>
<tr>
<th>Constraint</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
</tr>
</thead>
<tbody>
<tr>
<td>argmax</td>
<td>36</td>
<td>42</td>
<td>50</td>
<td>54</td>
<td>58</td>
<td>66</td>
</tr>
<tr>
<td>Our Projection</td>
<td>35</td>
<td>40</td>
<td>45</td>
<td>49</td>
<td>54</td>
<td>60</td>
</tr>
</tbody>
</table>

Table 2. The effect of discretization methods
5. Conclusion

The problem of resource-aware differentiable NAS is formulated as a bilevel optimization problem with hard constraints. Each level of the problem is addressed rigorously for efficiently generating well performing architectures that strictly satisfy the hard resource constraints. HardCoRe-NAS turns to be a fast search method, scalable to many devices and requirements, while the resulted architectures perform better than architectures generated by other state-of-the-art NAS methods. We hope that the proposed methodologies will give rise to more research and applications utilizing constrained search for inducing unique structures over a variety of search spaces and resource specifications.

References


