An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks

Yuke Wang[†], Boyuan Feng[†], Xueqiao Peng[*], Yufei Ding[†]
[†]{yuke_wang, boyuan, yufeiding}@cs.ucsb.edu, [*]{peng.969}@osu.edu
† University of California, Santa Barbara
[*]The Ohio State University

ABSTRACT

With the increasing popularity of deep learning, Convolutional Neural Networks (CNNs) have been widely applied in various domains, such as image classification and object detection, and achieve stunning success in terms of their high accuracy over the traditional statistical methods. To exploit potentials of CNN models, a huge amount of research and industry efforts have been devoted to optimizing CNNs. Among these endeavors, CNN architecture design has attracted tremendous attention because of its great potential of improving model accuracy or reducing model complexity. However, existing work either introduces repeated training overhead in the search process or lacks an interpretable metric to guide the design.

To clear these hurdles, we propose *3D-Receptive Field (3DRF)*, an explainable and easy-to-compute metric, to estimate the quality of a CNN architecture and guide the search process of designs. To validate the effectiveness of *3DRF*, we build a static optimizer to improve the CNN architectures at both the stage level and the kernel level. Our optimizer not only provides a clear and reproducible procedure but also mitigates unnecessary training efforts in the architecture search process. Extensive experiments and studies show that the models generated by our optimizer achieve up to 5.47% accuracy improvement and up to 65.38% parameters deduction, compared with state-of-the-art CNN model structures like MobileNet and ResNet.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; **Supervised learning by classification**.

KEYWORDS

Deep Learning, Neural Network Optimization, Image Classification.

ACM Reference Format:
Yuke Wang[†], Boyuan Feng[†], Xueqiao Peng[*], Yufei Ding[†]. 2021. An Efficient Quantitative Approach for Optimizing Convolutional Neural Networks. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management (CIKM '21), November 1–5, 2021, Virtual Event, QLD, Australia.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3459637.3482230

CIKM '21, November 1–5, 2021, Virtual Event, QLD, Australia
2021. ACM ISBN 978-1-4503-8446-9/21/11…$15.00
https://doi.org/10.1145/3459637.3482230

1 INTRODUCTION

Deep convolutional neural networks (CNNs) have achieved significant successes in a broad collection of fields, including object-detection [6], video classification [10], object tracking [29], image segmentation [17] and human pose estimation [28]. Such unparalleled successes attract many interests in CNN architecture design to *improve accuracy* or *reduce complexity*. Examples include an array of efficient models that have been crafted manually (*e.g.,* VGG [26], MobileNet [8], ShuffleNet [18]) and those generated automatically by the neural architecture search (NAS) tools [1, 14, 16, 22, 35]. Yet, two challenges of CNN architecture design remain far from well resolved: 1) missing an interpretable metric, and 2) huge training efforts. The former indicates that some direct and easy-to-interpret metric is still missing to guide the design, while the latter means that the repeated training cost is huge for evaluating different architectures in the search process.

To address these challenges, we propose *3D-Receptive Field (3DRF)*, an interpretable metric, for efficient CNN architecture designs. Particularly, we focus on two levels: the *stage level*[1] and the *kernel level*. At the stage level, we decide the number of convolution kernels in different stages, while at the kernel level we choose the type of the convolution kernel to use (*i.e.,* standard convolution kernels or efficient factorized kernels [25, 33]). We build up *3DRF* to uniformly conduct the optimization at both levels. The key insight is that the portion of the input tensor that can flow into each output neuron, which we name as *3DRF*, often determines the learning potential of that given stage or kernel. A stage or kernel with larger *3DRF* will have more input elements passing through, leading to a higher potential for extracting useful features and improving the classification accuracy. Therefore, we use *3DRF* to estimate the quality of architecture design in the search process, rather than repeated training.

To validate and showcase the effectiveness of *3DRF*, we propose an *architecture optimizer* to examine CNN architecture designs at stage and kernel level. At stage level, we provide an *organizer* to improve the accuracy of a CNN model while using the same or fewer convolution kernels. The organizer, in effect, removes the convolution kernels that cannot contribute to *3DRF* enough or move the kernels from the positions with marginal contributions to *3DRF* in one stage to another stage with larger contributions. The optimization is based on two key observations: 1) the contributions from the latter kernels in a stage are diminishing since the newly observed input elements are on the marginal positions, which have

[1]Following many works [7, 32, 33], we define a stage in a CNN as a collection of consecutive convolution layers with *input tensors* of the same spatial dimensions (*i.e.,* pooling or convolution kernel with stride ≥ 2 will generate a new stage).

less impact compared with the central input already observed; 2) when the spatial size of the input tensor to a stage is small, piling more layers can barely learn more features. On the other side, moving some layers to another stage with larger input tensor would promote *3DRF* and better learning capacity.

At the kernel level, we propose a *decomposer* to reduce model complexity without substantively affecting accuracy. The decomposer, in effect, replaces standard convolution kernels [2] with convolution blocks composed of efficient factorized kernels (*e.g.,* Depthwise Convolution [25], and Pointwise Convolution [27]). The key guidance behind such replacement is to maintain the same *3DRF* (*i.e.,* the efficient convolution block should observe the same amount of *3DRF* as standard convolutions in order to maintain accuracy). We name this rule as *Rule for Kernel Replacement*. This rule not only allows us to unify all existing convolution blocks used in MobileNet, ShuffleNet, clcNet [32], and Xception [2], but also inspires the discovery of one new basic factorized convolution kernel, as we named *Rolling Pointwise Convolution (RPW)*, and a new convolution block (Depthwise (DW) + RPW). This new *convolution block* turns out to be more efficient than existing factorized kernel designs, like that in MobileNet model.

To facilitate the end-to-end CNN model design, we introduce our design prototype. As shown in the Listing 1, we start with importing our 3DRF-based optimization libraries, including a stage optimizer (`stage_opt`) and a kernel optimizer (`kernel_opt`). We will then build a CNN models as we normally do in the regular Pytorch. Here, convolutional layers in the CNN models can be grouped into different stages, where each stages consists of convolutions linearly stacked together. Different stages are sequentially connected. At the end of those stages, we put the linear (fully-connected) layer and a softmax layer layer to generate logits for classification.

In summary, the major contributions of our work are:

- We propose a brand-new interpretable metric *3D-Receptive Field (3DRF)* for guiding CNN architecture designs efficiently. Whereas previous CNN model architecture exploration techniques (*e.g.*, NAS) require huge training and searching efforts.
- We build an end-to-end CNN stage-level organizer for improving the accuracy performance of CNN models at the model architectural level. This can largely ease the manual efforts in arduous CNN model optimization process.
- We introduce an new type of convolution kernel – Rolling-Pointwise Convolution to reduce the model parameters and the computation FLOPs.

Rigorous evaluations on real-world image datasets (*e.g.*, CIFAR-10/100 [11], and ImageNet [3]), demonstrate the strength of our architecture optimizer in terms of model accuracy, FLOPs and parameters. At the stage level, the organizer improves the accuracy (up to 5.47%) of the manually crafted CNN structures (*e.g.*, MobileNet) by maximizing the contribution to *3DRF*. For instance, the optimized MobileNet achieves 3.7% higher accuracy with 74% fewer parameters and 16% fewer FLOPs compared with the original structure. At the kernel level, the newly discovered convolution block achieves higher accuracy (up to 0.58%) with much fewer computations (up to 40.0% reduction) and parameters (up to 90.4% reduction) compared with the existing design. For example, one kernel designed by us has 2.54% higher accuracy and 29.04% fewer FLOPs in comparison with the MobileNet.

[2]In this paper, we refer to the standard convolution kernel as the one with 3 * 3 * C filters, where C is the number of input channels.

Listing 1: Illustration of 3DRF-based Optimizer Prototype.

```
from 3DRF_optimizer import stage_opt, kernel_opt
# import other libraries, such as Pytorch...

# Create an stage of CNN model.
def make_stage(stage_depth):
    layers = nn.sequential()
    for i in range(stage_depth):
        layers.append(nn.conv2D(inChannel, outChannel))
    return layers

# Create a CNN model.
class CNN(nn.module):
   def __init__(self, stageDepth=[2,2,2,2], outClass=10):
        self.stages = torch.nn.moduleList()
        for depth in stageDepth:
            self.stages.append(make_stage(depth))
        self.classifier = nn.Linear(flatDim, outClass)
        self.softmax = nn.softmax()

    def forward(self, X):
        out = X
        for stg in self.stages:
            out = stg(out)
        out = self.classifier(out)
        out = self.softmax(out)
        return out
# Define a simple CNN Model.
model = CNN([2,2,2,2], 10)
# Compute the delta 3DRF for a input model.
info_3DRF = stage_opt.comp_Delta3DRF(model)
# Optimize the model structure with delta 3DRF.
model_opt = stage_opt.optimze_arch(model, info_3DRF)
# Optimize the kernel.
model_final = kernel_opt(model_opt)
# Do regular model training and inference.
```

2 RELATED WORK

2.1 Neural Architecture Search (NAS)

NAS methods have been widely studied to automatically construct efficient CNN architectures. NAS frameworks generally come with three major components, 1) *Search space:* The NAS search space is composed of several types of operations (*e.g.*, convolution, fully-connected, and pooling) and the inter-connection among these operators. The design of search space demands domain expertise from both the deep learning and the specific application settings; 2) *Search algorithm:* A NAS search algorithm samples a population of network architecture candidates. It receives the model performance evaluation result (*e.g.,* result) as rewards and optimizes to generate high-performance architecture candidates. 3) *Evaluation strategy:* This step will measure the performance of candidate models in order to improve the search algorithm.

The most significant part of NAS research has been devoted to the neural architecture search algorithm. And a array of techniques and strategies have been proposed, such as evolutional algorithms [21, 22], hill climbing [4]; multi-objective search [5, 34], and reinforcement learning (RL) [16, 22]. To accelerate the NAS search, ENAS [20] represents the search space using a directed

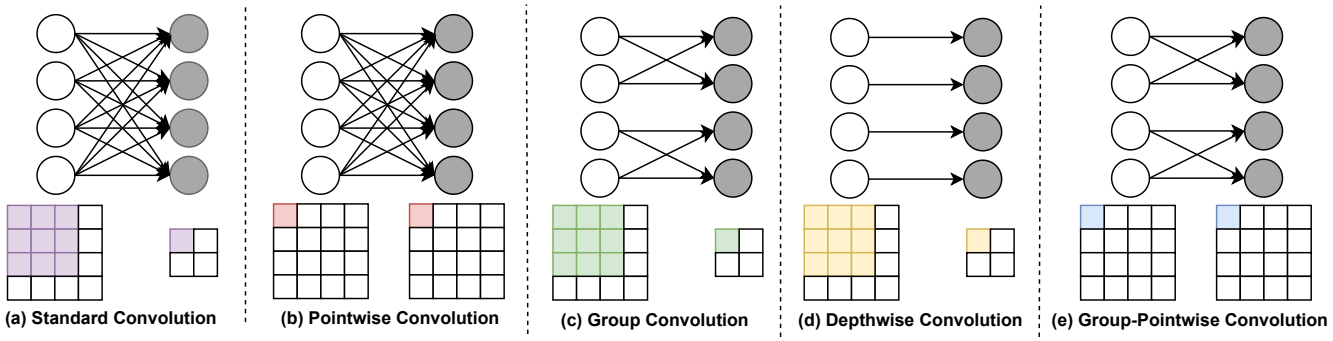

Figure 1: Channel mapping (top) and Spatial mapping (bottom) of the standard convolution and factorized convolution kernel.

acyclic graph (DAG) and targeting at optimizing the subgraph structure within the large supergraph. Meanwhile, it also introduces a training strategy of parameter sharing among subgraphs to significantly boost the searching efficiency. Work from [9, 15] also follow the similar idea of hierarchical computation graph optimization. Work from [31] further share the parameters of different paths within a block using super-kernel representation. [19] proposes a fine-grained search space comprised of atomic blocks that is much smaller than the ones used in recent NAS algorithms.

Although NAS methods can build high-quality CNN architecture, they have two major drawbacks. First, they require prohibitively expensive computing power and add significant overhead to the design time. For instance, the RL-based method in [35] requires 500 NVIDIA P100 GPUs for more than 4 days to evaluate 20000 candidate neural networks, even after adopting many proxy tasks techniques including early stopping with few epochs, running on a small dataset, and limiting the kernel numbers. Second, the NAS method can identify the design, but it does not explain the general rule behind to obtain such a design, which limits its applicability. Once the task changes, one has to run NAS again. In contrast, our static architecture optimizer gives an alternative solution, offering a clear and reproducible design procedure without training in the architecture search process. Other works still requires non-trivial overhead of CNN runtime profiling for optimization.

2.2 Standard Convolution

The widely applied deep learning application demands effective ways to capture the characters of the inputs (*e.g.*, images). Among those techniques, the standard convolution is most widely used in many CNNs [23, 26, 30]. In general, we annotate the input image (I), output feature map (O), and filter (F). The dimension of an image is $[I_w, I_w, C_{in}]$, where I_W is the size of an image while C_{in} is the number of input channels (*e.g.*, the RGB image has 3 input channel). The standard convolution (Figure 1a) leverages C_{out} standard convolutional filters with the shape of $[K, K, C_{in}]$, where the K is the filter size, C_{in} is the number of input channels, and C_{out} is the number filters. After applying the standard convolution on the input (with the shape of $[I_w, I_w, C_{in}]$), we will get the output feature map O, which has the shape of $[O_w, O_w, C_{out}]$., where the O_w is size of the output feature map. Note that the mainstream CNNs [8, 26, 30] generally maintain the same feature map spatial dimension at different convolutional layers while only changing the number of the channels across different layers.

Formally, for standard convolution, we have

$$O_{m,n,c} = \sum_{i,j,a}^{K,K,C_{in}} F_{i,j,a,c} * I_{m+i-1,n+j-1,a} \quad (1)$$

where $O_{m,n,c}$ is one pixel point in the output feature map; m and n are the spatial indexes in the output feature map ($m \in Z : m \in [0, O_w)$ and $n \in Z : n \in [0, O_w)$); a is the channel index in the input feature map ($a \in [0, C_{in})$); c is the channel index in the output feature map ($c \in Z : c \in [0, C_{out})$); i, j, and a are the index used to accumulated the elementwise multiplication values between input feature map and one filter. The standard convolution will not only extract the spatial information by traversing a $K \times K$ 2D sliding window within each channel but also effectively fuses the information across different channels (Figure 1a), where each kernel filter will gather the information from all input channels.

2.3 Kernel Factorization

Besides the standard convolution kernel, recent deep-learning research introduces several factorized kernels [12, 25, 27, 33] and combine them into a convolution block. This can offer another way to improve the computation efficiency of CNN architecture designs while maintaining the prediction power. Existing factorized kernels can be divided into four categories. Specifically, the first type is the Pointwise Convolution (PW) [27]) (Figure 1b), which is a standard convolution with 1×1 spatial size. The second type is Group Convolution (GC) [12] (Figure 1c) that divides input channels into several groups and performs standard convolution within each group. The third type is Depthwise Convolution (DW) [25] (Figure 1d) which calculates spatial convolution per channel or can be regarded as an extreme case of GC when the group number equals the number of the input channels. The last one is Group Pointwise Convolution (GPW) [33] (Figure 1e), that further splits PW into groups. Previously, researchers combine some of the factorized kernels into convolution blocks.

Xception [2] and MobileNet [8] demonstrate the successful application of convolutional kernel factorization in the popular CNN models. It breaks the original standard convolution into two parts: **depthwise** (DW) convolution and **pointwise** (PW) convolution. The first step (DW) applies C_{in} different $[W, W, 1]$ filters to each

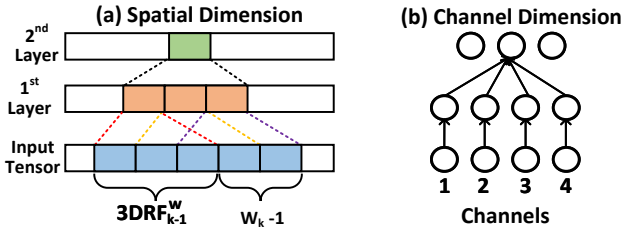

Figure 2: Illustration of 3D-Receptive Field (*3DRF*) for convolutions of a single stage.

of the C_{in} input channels independently, which can be formalized as Equation 2

$$\hat{O}_{m,n,a} = \sum_{i,j}^{K,K} F_{i,j,a}^{(dw)} * I_{m+i-1,n+j-1,a} \tag{2}$$

The second step (PW) applies a filter with 1×1 spatial dimension. As shown in Equation 3.

$$O_{m,n,c} = \sum_{a}^{C_{in}} F_{a,c}^{(pw)} * I_{m-1,n-1,a} \tag{3}$$

In this paper, we use the idea of *3DRF* to unify these previous convolution blocks. In addition, we create a new type of factorized convolution kernel, named *Rolling Pointwise Convolution (RPW)*, and a new convolution block (DW+RPW) that can outperform the previous designs.

3 3D-RECEPTIVE FIELD

In this section, we present *3D-Receptive Field* (*3DRF*) for measuring the representation ability of each neuron in a convolution layer. Then, we derive the *3D-Receptive Field Gain* (*3DRF* Gain) for quantifying the representation ability change when an additional convolution layer is inserted. This *3DRF* Gain is sensitive to the location, type, and combination of the inserted convolution layer, thus guiding the CNN design. We demonstrate the effectiveness of *3DRF* Gain in quantifying representation ability, in terms of its impact on accuracy.

Our 3D-Receptive field is inspired by an existing metric, *receptive field* [24], which quantifies the spatial area of neurons for evaluating a single neuron in the next convolution layer. This receptive field serves well for quantifying the local representation ability in a single traditional convolution layer, where a larger receptive field leads to higher accuracy. However, the receptive field fails to quantify the global representation ability across layers, when a large number of convolution layers with diverse receptive fields stacked in a CNN stage. Moreover, the receptive field fails to consider the channel number, which becomes critical in modern convolution layers (*e.g.*, Depthwise convolution and Channel-wise convolution). By contrast, our *3DRF* provides the first global metric for quantifying the global representation ability across layers, considering extensively the location, type, and combination of convolution layers. By quantifying the global representation ability, *3DRF* serves as an effective and efficient tool for guiding the CNN design without tediously enumerating and training NN architectures.

3.1 Definition of 3D-Receptive Field

For a CNN stage with a sequence of layers, we define the 3D-Receptive Field (*3DRF*) for the k^{th} convolution layer in the current stage as $3DRF_k$. This $3DRF_k$ captures the number of neurons in the initial input tensor to the CNN stage that contributes to computing individual neurons in this layer k. This initial input tensor is the $w_0 \times w_0 \times 3$ input tensor (*e.g.*, input image) in the first stage of a CNN, and a $w_0 \times w_0 \times c_0$ input tensor in later stages. Here, w_0 is the spatial width of the input tensor and c_0 is the channel number of the input tensor. To cater convolution layers with diverse kernel sizes and types, *3DRF* considers two factors of the spatial width $3DRF_k^w$ for the kernel size and the channel number $3DRF_k^c$ for the convolution type:

$$3DRF_k = (3DRF_k^w)^d * 3DRF_k^c \tag{4}$$

where $d = 1$ for 1D convolution (Figure 2) and $d = 2$ for 2D convolution. We recursively compute the spatial width $3DRF_k^w$ in layer k based on the spatial width $3DRF_{k-1}^w$ in the preceding layer $k-1$ and the kernel width w_k in the current layer k:

$$3DRF_k^w = \min(3DRF_{k-1}^w + w_k - 1, w_0) \tag{5}$$

A $min()$ is applied for ensuring that the spatial width $3DRF_k^w$ does not exceed the spatial width w_0 of the input tensor.

We compute recursively the channel number $3DRF_k^c$ in layer k with a property function $g(\cdot,\cdot)$, that captures the channel number $3DRF_{k-1}^c$ in the preceding layer $k-1$ and the convolution type T_k in the current layer k:

$$3DRF_k^c = \min(g(3DRF_{k-1}^c, T_k), c_0) \tag{6}$$

A $min()$ is applied for ensuring that the channel number $3DRF_k^c$ does not exceed the channel number c_0 of the input tensor. The property function $g(\cdot,\cdot)$ captures the information flow from the perspective of channel numbers and is designed for individual convolution types. For example, as illustrated in Figure 2, we set the property function $g(3DRF_{k-1}^c, PW) = c_0$ for Pointwise (PW) Convolution, since the output neuron of PW observes all input channels. Similarly, we set $g(3DRF_{k-1}^c, DW) = 3DRF_{k-1}^c$ for Depthwise Convolution (DW), since only one channel from the preceding layer $k-1$ contributes to the neuron in the current layer k. This property function $g(\cdot,\cdot)$ is designed only once for a small set of convolution types. While modern CNNs may have hundreds of convolution layers, these layers often use the same convolution type repeatedly. Thus, the property function can be written once and applied repeatedly for a large number of convolution layers.

3.2 Definition of *3DRF* Gain

We derive the *3DRF* Gain ($\Delta 3DRF$) to measure the impact of a convolution layer k over the model representation ability, in terms of the impact over the 3DRF. While *3DRF* quantifies the information flow in a convolution unit as a whole, *3DRF Gain*—denoted by $\Delta 3DRF$— targets at measuring the contribution of a single convolution kernel k in the unit. The goal of introducing $\Delta 3DRF$ is to create a direct indicator that could match the learning power (*i.e.,* prediction accuracy) of a CNN model in the granularity of a single convolution, laying a foundation for static architecture optimization. Specifically, we define $\Delta 3DRF_k$ as the difference in

Table 1: Illustration of computing *3DRF* Gain on Variant-3.

k	Layer Type	$3DRF_k^w$	$3DRF_k^c$	$3DRF_k$	$\Delta 3DRF_k$
1	conv3-256	3	128	1152	-
2	conv3-256	5	128	3200	1.17
3	conv3-256	7	128	6272	0.29

Table 2: Impact of *3DRF* Gain ($\Delta 3DRF$) over Accuracy.

Network	$\Delta 3DRF$	Accuracy (%)	ΔAccuracy (%)
VGG-11	0	92.68	0
Variant-1	1.73	93.56	0.88
Variant-2	1.60	93.46	0.78
Variant-3	0.29	92.75	0.07
Variant-4	0.0	92.58	-0.10
Variant-5	0.0	92.41	-0.27

the receptive field with and without the layer k, adjusted with an exponential decay term:

$$\Delta 3DRF_k = \frac{3DRF_k - 3DRF_{k-1}}{3DRF_{k-1}} * e^{-\alpha * \frac{3DRF_{k-1}}{V_0}} \quad (7)$$

where $V_0 = w_0 \times w_0 \times c_0$ is the volume of the input tensor. The exponential decay term rescales the impact of the k^{th} layer with regards to the information already observed by 1^{th} to $(k-1)^{th}$ layers. which composes of two major terms: the former calculates the relative increase in *3DRF* incurred by kernel k; the latter introduces an exponentially decay term to rescale the impact of the k^{th} layer with regards to the information already observed by 1^{th} to $(k-1)^{th}$ layers. This decay term is inspired by the observation that the elements in the central region of the input tensor usually have a larger impact than the newly observed elements on the margin: the central input elements have more paths to propagate their values into the output in the forward pass and larger gradient in the backward pass. Note that α is a hyperparameter that should be set larger than 0. In our empirical study, we tried multiple choices and observed no substantial difference in architecture optimization, and we set it to 3 for the rest of this paper.

3.3 Case Study: Accuracy Impact of *3DRF* Gain

We demonstrate the impact of diverse ($\Delta 3DRF$) over the accuracy. Here we generate diverse ($\Delta 3DRF$) by sticking to the same base model and inserting an additional convolution layer at diverse location. More study on the ($\Delta 3DRF$) from varying the type and combination of convolution layers will be conducted later in the evaluation section. As shown in Figure 2, we take VGG11 [26] as the baseline structure and run it on CIFAR-10 dataset [11]. Specifically we generate five VGG-variants by inserting a single standard convolution before each max pooling. The inserted convolution layer has the same kernel width and channel number as its preceding layer. For example, we insert a conv3-64 before the first max pooling as the Variant-1, and a conv3-512 before the fifth max pooling as the Variant-5. Specifically, we train these models on the CIFAR-10 training dataset and report the accuracy on the CIFAR-10 testing dataset. We repeat this procedure for ten times and present the average accuracy here. We also present the $\Delta 3DRF$ of each variants for demonstrating the impact of $\Delta 3DRF$ over accuracy. $\Delta 3DRF$ is calculated by leveraging our proposed Equation 7 for the newly inserted layer.

As shown in Table 1, the procedure of computing $\Delta 3DRF$ on Variant-3, which inserts an additional layer to the third stage in VGG-11. Originally, the third stage in VGG-11 contains two convolution layers (*i.e.*, the 1^{st} layer and the 2^{nd} layer in Table 1). We insert the 3^{rd} convolution layer with the same kernel width and channel number as the first two layers. The input tensor to this third stage is of shape $8 \times 8 \times 128$, leading to a V_0 of 8192. Following Equation 4 - 6, we can compute $3DRF_k^w$, $3DRF_k^c$, and $3DRF_k$ recursively. The derived $3DRF_k$ can be exploited for computing $\Delta 3DRF$ following Equation 7. This procedure can be applied for other VGG-11 model variants, leading to the $\Delta 3DRF$ in Table 2.

As shown in Table 2, we can clearly figure out the the impact of $\Delta 3DRF$ on CNN model accuracy. Large $\Delta 3DRF$ of the newly inserted layer agrees with notable accuracy gain, as is the case for *Variant-1* and *Variant-2*. For the *Variant-3*, small $\Delta 3DRF$ indicates close-to-saturation information coverage, yielding negligible accuracy improvement from the original model. Variant-4 and Variant-5 has a low $\Delta 3DRF$ of 0, indicates that inserting convolution layers does not improve its *3DRF*. The insight is that, for an input tensor with a small spatial width w_0 of 2 (after 4 times of max pooling from an input image of shape $32 \times 32 \times 3$), a single convolution layer of kernel width 3 is sufficient for capturing all neurons. In fact, Variant-4 and Variant-5 show an accuracy degradation of -0.10% and -0.27% respectively. This degradation shows that a $\Delta 3DRF$ of 0 signals overfitting since all input elements have already been observed by other kernels at such stage. Comparing across variants, Variant-1 has a larger $\Delta 3DRF$ of 1.73 and a larger $\Delta Accuracy$ of 0.88%, compared with Variant-5 with $\Delta 3DRF$ of 0.0 and $\Delta Accuracy$ of -0.27%. This trend demonstrates a strong correlation between the $\Delta 3DRF$ and the $\Delta Accuracy$, thus guiding the NN design in terms of the insertion location.

To sum up, $\Delta 3DRF$ effectively probes the potential of accuracy improvement, and we leverage such an easy-to-compute metric to build our architecture optimizer in Section 4.

4 ARCHITECTURE OPTIMIZER VIA 3DRF

We build a static *Architecture Optimizer* based on *3DRF* and $\Delta 3DRF$. It examines the structure inefficiency in a given CNN architecture and optimizes it at the *stage level* and *kernel level*.

4.1 Stage-Level Organizer

Stage-level organizer (Figure 3) manages to improve the prediction accuracy of a CNN design by iteratively removing a convolution kernel from a saturated stage or moving it to another stage with more room to absorb new information (*i.e.,* learn from more marginal elements introduced by the kernel).

Three sub-steps are conducted in each iteration. The first step is to find the convolution kernel with minimum $\Delta 3DRF$, which has the lowest contribution to the 3DRF. In consideration of the decaying property of $\Delta 3DRF$ within a stage, this step can be simplified to compute the $\Delta 3DRF$ of the last convolution kernel in each stage.

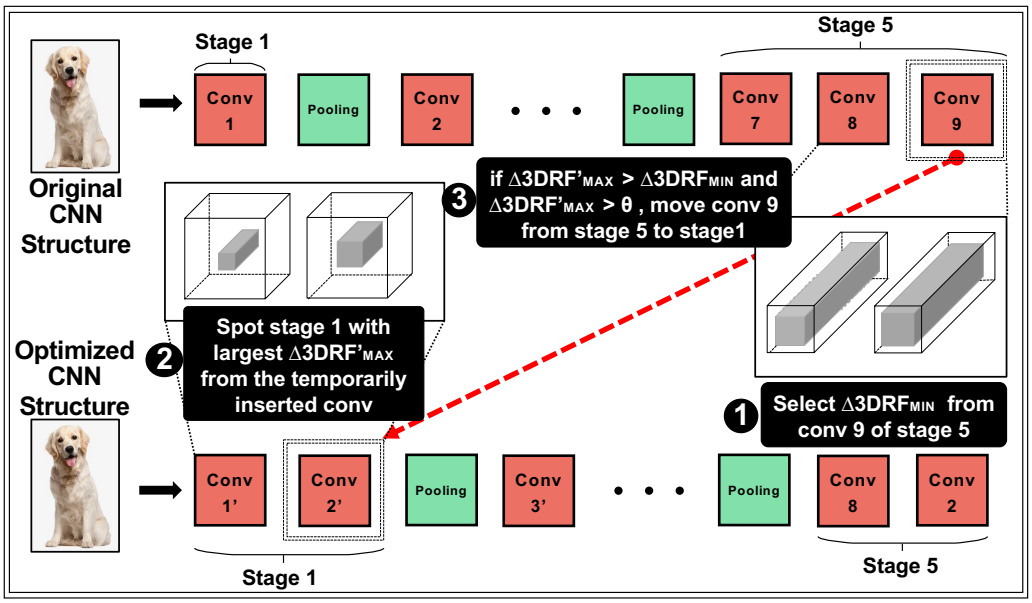

Figure 3: Illustration of the Stage-level Organizer.

Comparing across stages, we select the convolution layer with the minimum *3DRF* Gain, denoted as $\Delta 3DRF_{MIN}$ in Figure 3, and identify the corresponding stage as the *source stage*. This identified convolution layer will be either deleted or moved from the source stage to another stage, in the following steps.

The second step is to spot the stage with the largest room for improving 3DRF. This step follows the insight from our case study that a larger $\Delta 3DRF$ often leads to higher accuracy. We tentatively append the convolution kernel identified in the first step to each stage and compute the corresponding $\Delta 3DRF$. When appending the convolution layer, the input and output channel number will be adjusted for catering to the preceding layers in the source stage and the following layer in the next stage if available. Comparing across stages, we can find the one, called *target stage*, with maximum $\Delta 3DRF$ for the appended layer ($\Delta 3DRF'_{MAX}$ in Figure 3). This step follows the insights obtained from our case study that a strong correlation exists between $\Delta 3DRF$ and $\Delta Accuracy$. to conduct architecture optimization.

The third step decides whether moving the last convolution layer from the source stage to the target stage or simply removing this layer. When moving the convolution layer, we adjust the input channel number and the output channel number with the same strategy in the second step. This step follows the insights obtained from our case study to conduct architecture optimization. There are three key choices: 1) If $\Delta 3DRF'_{MAX} > \Delta 3DRF_{MIN}$ *and* $\Delta 3DRF'_{MAX} > \theta$, we move the last kernel from the *source stage* and append it to the *target stage*; 2) If $\Delta 3DRF'_{MAX} < \theta$ *and* $\Delta 3DRF_{MIN} < \theta$, we just remove the last kernel from the *source stage* (no appending); 3) If $\Delta 3DRF_{MIN} > \Delta 3DRF'_{MAX}$ and $\Delta 3DRF_{MIN} > \theta$, we keep the original structure and terminate our optimization procedure. Here the hyperparameter θ is the border we draw empirically to distinguish underfitting from overfitting. For example, θ is set to 0 for VGG. Following this iterative optimization procedure, our organizer manages to mitigate the structure-level inefficiency in a CNN design via static architecture optimization. The experimental results of the organizer can be found in our evaluation.

4.2 Kernel-Level Decomposer

At the kernel level, our *decomposer* reduces the computational cost of a CNN architecture design, by substituting its *standard convolution kernels* with less computational expensive *convolution blocks*. The key challenge here is to construct such an efficient and effective *convolution block* with multiple factorized kernels. Previous manual efforts by domain experts have made some progress [23, 32, 33], but the underlying design principle remains unclear. In this paper, we provide the first easy-to-follow design principle, *Rule of Kernel Replacement*, to guide the design of efficient convolution blocks.

Rule of Kernel Replacement To avoid significant accuracy degradation and achieve computation efficiency, a convolution block N can replace the standard convolution kernels S only if two conditions are satisfied: 1) *Quality Condition*: $3DRF(N) = 3DRF(S)$ for the same input tensor; 2) *Compact Condition*: $3DRF(N - x) < 3DRF(S)$ if we remove a factorized kernel x from N. The former ensures the effectiveness of N with regards to its learning capacity, while the latter guarantees its optimality in terms of computation efficiency. The rule helps us unify the previous construction of the convolution block, as well as inspires us to build a new convolution blocks and one efficient factorized kernel.

Unifying Existing Convolution Blocks This section shows that the previous four convolution blocks follow the *Rule of Kernel Replacement*: they have the same *3DRF* as the standard convolutions and they are already in the compact form that cannot be further simplified. Figure 4 depicts the *3DRF* for a standard convolution block (S) and four previously explored convolution blocks (A-D), in their spatial and channel dimensions. As shown in Figure 4 (S), the *3DRF* spatial size $3DRF_1^w$ for S is 3 for one standard convolution and $3DRF_2^w$ is 5 when two standard convolutions are packed together

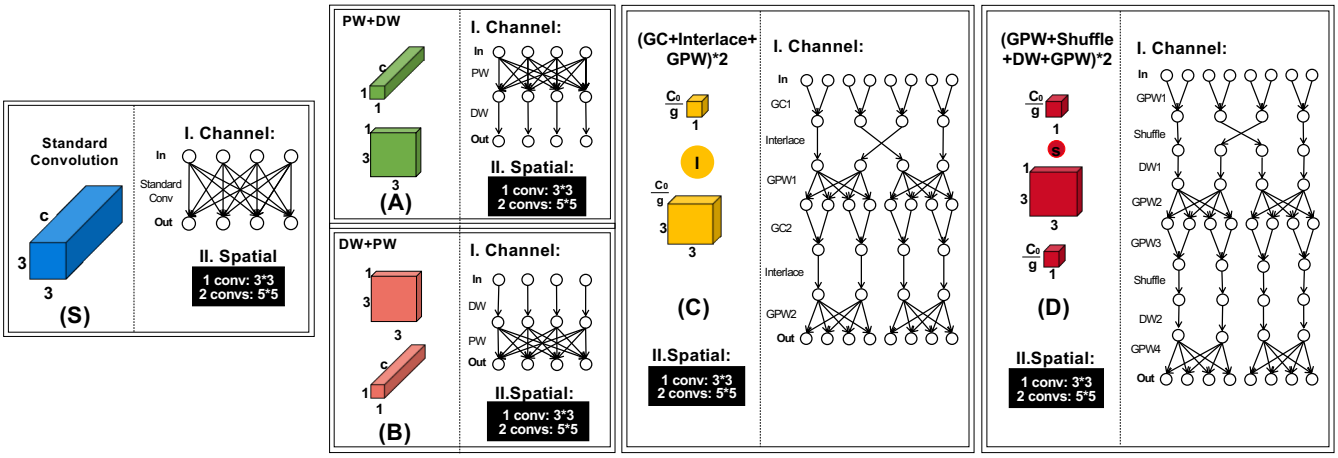

Figure 4: Illustration of the *3DRF*, both in the channel (I) and spatial (II) dimension, for the standard kernels (*S*) and previous convolution blocks (*A-D*). g is the number of groups for GC and GPW. The arrow denotes the flow from inputs to outputs in the channel dimension, and the number of input channels that could flow into an output neuron would be the channel dimension of *3DRF* for that block. We omit the process of computing the spatial size of *3DRF*, while only giving the computed result based on Equation 4 in the figure.

in the block. The *3DRF* channel dimension $3DRF_k^c$ for *S* equals the number of the input channels to the block.

Convolution block *A* (adopted by Xception [2]) and *B* (applied in MobileNet [8]) follow a similar structure. Both *A* and *B* successfully maintain the same *3DRF* with that of *S* with one standard kernel. Specifically, the spatial coverage is managed by DW [3] and channel coverage is taken care of by PW, which communicates the information among all input channels. Convolution block *C* (used in clcNet [32]) and *D* (utilized by ShuffleNet [33]), on the other hand, achieve the same *3DRF* with that of *S* with two standard kernels. Take block *C* (shown in Figure 4 (*C*)) as an example, one combination of GC, Interlace, and GPW, can perceive the same spatial region but only half of the entire input channels, compared to a standard convolution kernel. But with one extra GC+Interlace+GPW, the channel dimension gets full coverage. Thus, the *3DRF* is the same for the block with (GC+Interlace+GPW) * 2 and two standard convolutions. The proof of the compactness for four convolution blocks is omitted, but it is clear from the plot that if we remove any of the factorized kernels, the *3DRF* cannot be maintained.

New Kernel Design Inspired by the *Rule of Kernel Replacement*, we discover an unexplored convolution block and a new type of factorized kernel, shown in Figure 5. The first block includes a DW, a channel shuffle, and a GWC. The *key insight* of the design is choosing a DW to capture information in the spatial dimension and using a GPW with a shuffle operation to observe full channel information. Since the PW contributes to the majority of the computations in the previous factorized design (more than 95% FLOPs in MobileNet [8]), the usage of GPW to replace PW can largely reduce the computation cost, compared to blocks like (A) and (B).

The convolution block we come up with composes of a DW and a Rolling-Pointwise Convolution (RPW), as shown in the left side of Figure 5 (model *F*). The comparison between RPW and

[3]Definitions of factorized kernels like DW can be found in the Related Work Section.

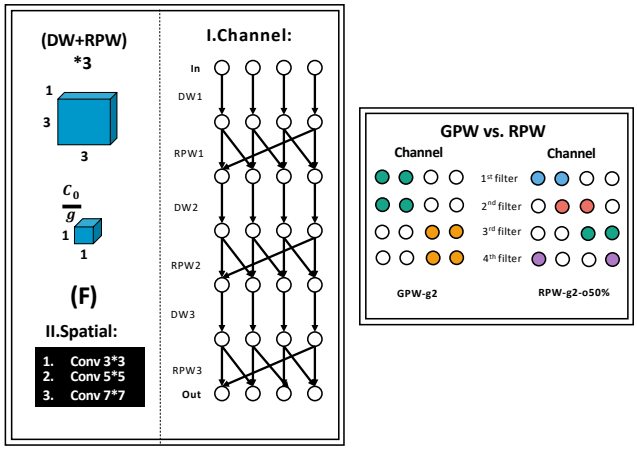

Figure 5: Left: DW+RPW convolution block design. Right: Comparison of RPW kernel with GPW kernel. Note that in RPW, adjacent filters overlap in channel dimensions.

GPW is presented in the right side of Figure 5. Different from GPW, RPW is the new factorized convolution kernel we invented, where adjacent convolution filters partially overlap in the channel dimension. The overlapped part serves as a bridge to communicate the different channel information and allows the later kernel to observe different channels without channel shuffle. Specifically, there are two parameters that come with RPW: group number g and overlap ratio o. For instance, RPW-gX-oY% denotes each filter in the convolution kernel takes $\frac{1}{X}$ number of input channels, while adjacent filters in RPW have y% overlap in their consumed channels. The newly designed block outperforms previous designs in accuracy, memory and computation efficiency, which are detailed in our evaluation.

Listing 2: Compositing RPW via PyTorch Operators.

```
width = int(input_channel/num_groups)
start, end, start_v, end_v= 0, width, 0, width
item_set, slice_li = set(), []
# input channel range for each kernel filter.
for fid in range(output_channel):
    item_set.add((start,end)); slice_li.append((start,end))
    start_v = end_v - int(overlap * width)
    end_v = start_v + width
    start, end= start_v%input_channel, end_v%input_channel
# define a groupwise convolution.
conv2D = nn.Conv2d(width*len(item_set), len(item_set),
                   kernel_size=1, groups=len(item_set))
# forward computation.
def forward(input):
    comb_unit = []
    for idx in range(len(item_set)):
        item = slice_li[idx]
        start, end = item[0], item[1]
        if start > end and start < input_channel:
            tmp = input[:, start:, :, :]
            tmp_1 = input[:, :end, :, :]
            new_tmp = torch.cat([tmp, tmp_1], dim=1)
            comb_unit.append(new_tmp)
        else:
            comb_unit.append(input[:, start:end, :, :])
    comb_tensor = torch.cat(combined_unit, dim=1)
    return conv2D(comb_tensor)
```

Implementation of New Kernel Design To implement the new rolling-pointwise convolution, we introduce two kind of implementation by compositing the existing Pytorch Operators. First, we can first extract the corresponding channels and concatenate them together. We will leverage the existing Pytorch operators, such as tensor slicing, concatenation, and standard group convolution. There are several steps, as shown in Listing 2. The second type of design is to let the convolution iterate through the input channel. The second implementation circumvents the "huge" concatenated tensor in the above implementation by applying convolution operation before concatenating. One major key insight is that the computation on the large concatenated tensor can be decomposed into the more effective computation on a set of small tensors. Instead of simply combining all the extracted features maps, we can pre-build a set of lightweight convolutions, each of which will generate the feature map for only one kernel filter. Finally, we concatenate these output feature map together. While this solution can largely overcome the third problem of the above channel-stack implementation, it is still hindered by the excessive inefficient Pytorch operations and lack of parallelization.

5 EVALUATION

To validate the effectiveness of the architecture optimizer, we run comprehensive experiments on the state-of-the-art CNN models (VGG16 and VGG19 [26], MobileNet [8] and ResNet50 [7]. The major reason of choosing these CNN models are 1) VGG16 and VGG19 are two most classic CNNs with linearly stacked layers; 2) MobileNet is the representative lightweight model with DW+PW convolution block; 3) ResNet50 is the representative model with the non-linearly stacked layers (residual connections).

Dataset: We use CIFAR-10 (CIFAR-100) [11] and ImageNet [3] dataset for evaluation. CIFAR-10 consists of 60,000 32×32 colour images in 10 classes, with 6,000 images per class. CIFAR-100 dataset is just like the CIFAR-10, except it has 100 classes containing 600 images each. ImageNet is a large dataset of over 14 million images with up to 1,000 output classes, and it is mainly used for computer vision research, such as image classification.

Training Settings: We follow the conventional settings [13] for training and testing on CIFAR-10 and CIFAR-100: learning rate starts from 0.1 and decays by the factor of 0.1 after 150 and 250 epochs, with 350 epochs in total. We adopt SGD with 0.9 momentum and 5e-4 for the weight decay. We apply normalization for the input image with (0.491, 0.482, 0.446) for each RGB channel as the mean and (0.247, 0.243, 0.261) for standard deviation, respectively. And we select two state-of-the-art Pytorch CNNs implementations on CIFAR-10 and CIFAR-100 , respectively. For ImageNet, we use the official Pytorch implementations [4] and choose learning rate starts with 0.1 with total 120 epochs. We adopt SGD with 0.9 momentum and 1e-4 weight decay. We also apply normalization for the input image with (0.485, 0.456, 0.406) for each RGB channel as the mean and (0.229, 0.224, 0.225) for standard deviation. We select the pre-trained model as the baseline from Pytorch official website.

5.1 Stage-Level Organizer

This experiment aims to demonstrate the effectiveness of our stage-level organizer. Specifically, we first use CIFAR-10 and CIFAR-100 for detailed analysis, and further leverage ImageNet to show our design applicability and scalability towards the challenging state-of-the-art large dataset. Table 3 exhibits the performance of various CNNs optimized by the stage-level organizer, including computation complexity (MFLOPs), parameter size, and accuracy. It is clear that the stage-level organizer can improve the accuracy of various state-of-the-art CNN models. On CIFAR-10 and CIFAR-100, stage-level organizer improves the accuracy of four evaluated models by 1.18% and 1.90% on average, while reducing model parameters by 54.15% and 32.33% on average, respectively. We also notice on the more complicated model, such as ResNet50, the accuracy improvement is notable (2.04% on CIFAR-10 and 0.86% on CIFAR-100). The original ResNet50 model has 4 stages. Each stage contains {3, 4, 6, 3} bottleneck blocks respectively. Following the iterative optimization steps, the organizer moves the last two blocks from the third stage to the first stage and the last block from the last stage to the second stage to generate an optimized ResNet50 containing {5, 5, 4, 2} blocks in each stage. By improving the total $\Delta 3DRF$, this optimized architecture gets both higher accuracy and fewer model parameters. In addition, on the lightweight MobileNet model, which has factorized kernel designs (DW+PW) with the smallest number of parameters, our stage-level organizer also achieves a notable performance improvements (1.38% on CIFAR-10, and 5.47% on CIFAR-100). This is because our organizer finds five convolutions—four from the fourth stage and one from the last stage—which suffer from small $\Delta 3DRF$. By moving these convolutions to the first and second stage, we get a new architecture contains {4, 4, 2, 2, 1} convolutions in each stage, which offers a more efficient architecture in terms of less model parameters and higher accuracy. On the challenging ImageNet, our stage-level organizer can still effectively reduce the number of model parameters (up to 16.7%), meanwhile improving the testing accuracy (up to 0.58%) compared with baseline models.

[4] github.com/pytorch/examples/tree/master/imagenet

Table 3: Performance comparison (CIFAR-10) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	Δ3*DRF*
VGG16	310	14.73M	92.64	-
VGG16-opt	370	**5.10M**	**92.95**	2.30
VGG19	400	20.04M	91.91	-
VGG19-opt	490	**8.09M**	**92.89**	3.13
MobileNet	50	3.22M	90.67	-
MobileNet-opt	50	**1.13M**	**92.05**	3.94
ResNet50	1,300	23.52M	93.75	-
ResNet50-opt	1,310	**17.24M**	**95.79**	0.76

Table 4: Performance comparison (CIFAR-100) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	Δ3*DRF*
VGG16	330	34.02M	72.93	-
VGG16-opt	390	**24.39M**	**74.64**	2.30
VGG19	420	39.33M	72.23	-
VGG19-opt	500	**27.38M**	**74.00**	3.13
MobileNet	50	3.32M	65.98	-
MobileNet-opt	50	**1.23M**	**71.45**	3.94
ResNet50	1,310	23.71M	77.39	-
ResNet50-opt	1,380	**21.89M**	**78.25**	0.76

Table 5: Performance comparison (ImageNet) between original CNNs and reorganized structures.

Network	MFLOPs	Param.	Acc. (%)	Δ3*DRF*
VGG16	15,500	138.36M	71.59	-
VGG16-opt	16,900	**133.82M**	**72.17**	0.39
VGG19	19,670	143.67M	72.38	-
VGG19-opt	21,060	**141.34M**	**72.61**	1.09
MobileNet	580	4.23M	70.60	-
MobileNet-opt	570	**3.52M**	**71.05**	2.59
ResNet50	4,120	25.56M	76.15	-
ResNet50-opt	4,130	**23.67M**	**76.56**	0.47

5.2 Kernel-Level Decomposer

This experiment aims to demonstrate the benefits of our brand-new kernel design. We first use VGG16-opt (with stage-level optimization) on CIFAR-10 for a detailed study. We further highlight our new kernel scalability by applying it towards the complicated ResNet50-opt model on ImageNet. Table 6 shows that our new convolution block based on rolling-channel design achieve a better balance between the model efficiency and the prediction accuracy on VGG16-opt on CIFAR10, in contrast to DW+PW factorized kernel design. We tried three different group numbers g (2, 4, 8), as well as two overlapping ratios o (33%, 50%). Our model with DW+RPW-g2-o50% achieves a better accuracy compared to the high-performance

Table 6: Kernel-level design (CIFAR-10) on VGG16-opt.

Network	MFLOPs	Param.	Acc.(%)
Baseline	370	9.64M	92.95
DW+PW	50	1.11M	92.12
DW+GPW-g2	30	0.67M	92.35
DW+GPW-g4	20	0.36M	88.05
DW+GPW-g8	10	0.20M	86.41
DW+RPW-g2-o33%	**30**	**0.66M**	**92.52**
DW+RPW-g2-o50%	**30**	**0.66M**	**92.70**
DW+RPW-g4-o33%	**20**	**0.36M**	**91.61**
DW+RPW-g4-o50%	**20**	**0.36M**	**91.59**
DW+RPW-g8-o33%	**10**	**0.20M**	**89.86**
DW+RPW-g8-o50%	**10**	**0.20M**	**90.19**

DW+PW model while saving about 40.0% FLOPs and 40.5% parameters. With an increase in the group number, we observe a significant reduction in both computational cost and parameter usage, along with a slight degradation in prediction accuracy. This aligns well with our expectation that the group number g determines the number of input channels that GPW/RPW would take, and thus also decides the number of computations and parameters of the model.

We also notice that our new convolution block design consistently outperforms with the ones without overlap (o) under the same number of groups (g). For example, our new design (DW+RPW-g4-o33%) outperform DW+RPW-g4 with 3.56% better accuracy. Under the settings with same number of group in RPW, such as DW+RPW-g2-o33% vs. DW+RPW-g2-o50%, the latter with higher overlap ratio offers higher accuracy, indicating the effectiveness of overlapping channels to improve model accuracy.

6 CONCLUSION

In this paper, we propose *3D-Receptive Field (3DRF)*, an interpretable and easy-to-compute metric to guide the search of CNN designs. To illustrate the usefulness of *3DRF*, We build an optimizer and improve the CNN structure at the stage and kernel level. The stage-level optimization target at reducing the model structural redundancy by improving the kernel organization, while the kernel-level optimization improve the individual kernel design by reducing the number of parameters without much compromising the model accuracy. Experiments show models generated by our optimizer achieve higher efficiency and accuracy compared with state-of-the-art CNNs.

7 ACKNOWLEDGMENT

This work was supported in part by NSF 1925717. Use was made of computational facilities purchased with funds from the National Science Foundation (OAC-1925717) and administered by the Center for Scientific Computing (CSC). The CSC is supported by the California NanoSystems Institute and the Materials Research Science and Engineering Center (MRSEC; NSF DMR 1720256) at the University of California, Santa Barbara.

REFERENCES

[1] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. 2017. Designing neural network architectures using reinforcement learning. *ICLR* (2017).

[2] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[3] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition (CVPR)*.

[4] Thomas Elsken, Jan-Hendrik Metzen, and Frank Hutter. 2017. Simple and efficient architecture search for convolutional neural networks. *arXiv* (2017).

[5] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Efficient Multi-objective Neural Architecture Search via Lamarckian Evolution. *ICLR* (2019).

[6] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[8] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *arXiv e-prints* (2017).

[9] Xiaojie Jin, Jiang Wang, Joshua Slocum, Ming-Hsuan Yang, Shengyang Dai, Shuicheng Yan, and Jiashi Feng. 2019. Rc-darts: Resource constrained differentiable architecture search. *arXiv preprint arXiv:1912.12814* (2019).

[10] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale Video Classification with Convolutional Neural Networks. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[11] Alex Krizhevsky and Geoffrey Hinton. 2009. *Learning multiple layers of features from tiny images*. Technical Report. Citeseer.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.).

[13] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2017. Pruning filters for efficient convnets. *ICLR* (2017).

[14] Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan Yuille, Jonathan Huang, and Kevin Murphy. 2018. Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[15] Hanxiao Liu, Karen Simonyan, and Yiming Yang. 2018. Darts: Differentiable architecture search. *arXiv preprint arXiv:1806.09055* (2018).

[16] Zhuang Liu, Jianguo Li, Zhiqiang Shen, Gao Huang, Shoumeng Yan, and Changshui Zhang. 2017. Learning Efficient Convolutional Networks Through Network Slimming. In *The IEEE International Conference on Computer Vision (ICCV)*.

[17] Jonathan Long, Evan Shelhamer, and Trevor Darrell. 2015. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[18] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, and Jian Sun. 2018. Shufflenet v2: Practical guidelines for efficient cnn architecture design. In *Proceedings of the European Conference on Computer Vision (ECCV)*.

[19] Jieru Mei, Yingwei Li, Xiaochen Lian, Xiaojie Jin, Linjie Yang, Alan Yuille, and Jianchao Yang. 2020. AtomNAS: Fine-Grained End-to-End Neural Architecture Search. In *International Conference on Learning Representations (ICLR)*.

[20] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, and Jeff Dean. 2018. Efficient neural architecture search via parameters sharing. In *International Conference on Machine Learning*. PMLR, 4095–4104.

[21] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V Le. 2019. Regularized evolution for image classifier architecture search. *AAAI* (2019).

[22] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, and Alexey Kurakin. 2017. Large-scale evolution of image classifiers. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*.

[23] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[24] Charles Scott Sherrington. 1906. Observations on the scratch-reflex in the spinal dog. *The Journal of physiology* (1906).

[25] Laurent Sifre and Stéphane Mallat. 2014. *Rigid-motion scattering for image classification*. Ph.D. Dissertation. Citeseer.

[26] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. *ICLR* (2015).

[27] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence (AAAI)*.

[28] Alexander Toshev and Christian Szegedy. 2014. Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.

[29] Naiyan Wang and Dit-Yan Yeung. 2013. Learning a Deep Compact Image Representation for Visual Tracking. In *Advances in Neural Information Processing Systems (NeurIPS)*, C. J. C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K. Q. Weinberger (Eds.).

[30] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[31] Jiahui Yu, Pengchong Jin, Hanxiao Liu, Gabriel Bender, Pieter-Jan Kindermans, Mingxing Tan, Thomas Huang, Xiaodan Song, and Quoc Le. 2019. Scaling Up Neural Architecture Search with Big Single-Stage Models. *arXiv preprint* (2019).

[32] Dongqing Zhang. 2018. clcNet: Improving the Efficiency of Convolutional Neural Network using Channel Local Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[33] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. 2018. ShuffleNet: An Extremely Efficient Convolutional Neural Network for Mobile Devices. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[34] Yanqi Zhou and Gregory Diamos. 2018. Neural architect: A multi-objective neural architecture search with performance prediction. In *SysML*.

[35] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V Le. 2018. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*.