Universiteit Leiden

Master Computer Science

Learning Transferable Architectures for Image Denoising

Name: Siyuan Dong
Student ID: S2576597
Date: 30/09/2021

Specialisation: Data Science

1st supervisor: Jan van Rijn
2nd supervisor: Holger Hoos

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Abstract

Current deep learning methods rely on neural network architectures and training hyperparameters. For researchers and engineers without computer vision skills, finding a good architecture is not easy. In this work, we apply differentiable architecture search methods to conduct affordable architecture searches for image denoising. We search for cell architecture by gradient-based optimization and network architecture by AutoML methods such as grid search, random search and successive halving search. We further perform BOHB hyperparameter optimization on the searched models. The searches for overall architectures and training hyperparameters are fully automatic. The models we searched achieved promising results on image processing datasets BSD68 and Set12.

Acknowledgements

Time flies when not paying attention. When I write down the acknowledgements, it also marks the end of my master study in the Netherlands. I am grateful to have the opportunity of studying abroad and embracing a larger world. I want to thank the support and helps from my supervisors. My first supervisor Jan van Rijn held the weekly meetings with me and provided a lot of guidance for my thesis. He also offered me warm suggestions and assistance to my personal development. My second supervisor Holger Hoos also worked closely with me and provided insightful ideas in the meetings. Without them, I would not be able to finish my thesis smoothly. I would also like to thank other teachers and teaching assistants for their dedication to the high-quality courses I took. Lastly, I want to thank my family for their unconditional love and consistent support.

Siyuan Dong
Leiden University
October 12, 2021

Contents

Chapter 1

Introduction

1.1 Neural Architecture Search

The success of deep learning over the last years further promotes the exploration of well-designed deep neural networks. However, the capability of a model is influenced by many factors such as architectural innovations, hyperparameter optimization, activation and loss functions. Deep learning researchers utilize their knowledge to design dominant architectures with suitable hyperparameters, like the extraction of spatial and channel information, multi-path information fusion, and the depth and width of the network, which requires advanced expert knowledge and sufficient computing resources. At the same time, a model without hyperparameter optimization can not reflect its actual capability. Researchers and engineers anticipate carrying out these processes automatically. Therefore, Neural Architecture Search (NAS), the strategy of automating the design of neural networks, is a promising solution for automated machine learning.

Currently, NAS architectures have outperformed manually designed architectures on perceptual tasks such as image classification [26, 37], object detection [37] or semantic segmentation [6]. Common search space includes chain-structured neural networks [13], multi-branch networks [13] incorporating skip connections to flow information from multiple branches, and recently, cell-based search space [21, 23, 37] that allowing to search for the micro-architecture and stack the searched micro-architecture to form the resulting network. Among them, cell-based search space has the advantages of reducing the search cost and the flexibility of transferring to different datasets. Different searching strategies have been applied for NAS, like Bayesian optimization, evolutionary algorithms (EA), reinforcement learning (RL), and gradient-based optimization methods. Compared to other search strategies, gradient-based methods require less computing resources and running time, which receive considerable attention and have become a research hotspot in recent years.

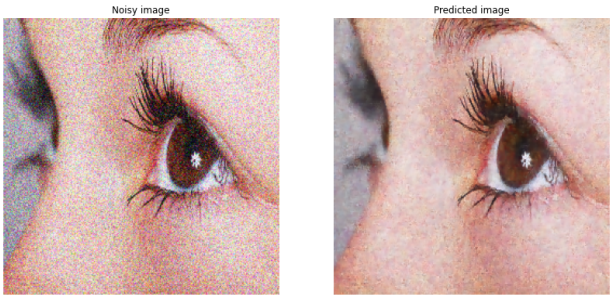

Figure 1.1: Examples of a noisy face image and the predicted clean image from PRIDNet [36].

1.2 Image Denoising

Image denoising is a classic low-level vision task with multiple applications in the real world. The aim is to recover the clean images from corrupted images with noises brought from the environment or in the process of signal transmission, which is essential for image processing, video analysis and provides support for high-level tasks such as object recognition and autopilot. It is still a challenging question since edges and textures may also be destroyed when removing the noise. Therefore the balance between denoising and preserving details should be maintained in this task. Fig. 1.1 shows examples of a corrupted face picture and its denoising prediction via PRIDNet [36]. In these examples, face details like eyelashes and eye canthus are well preserved after denoising.

In the past few decades, prior-based models have achieved outstanding performance, for instance, sparse models [24, 12, 11], Markov random field models [19, 17], nonlocal self-similarity models [1, 2, 10, 24], while model-based methods have drawbacks of involving complex optimization and many manually selected parameters [33]. At the same time, convolutional neural networks have been exploited in many vision tasks owing to the development of hardware and won success in many tasks, including image denoising. Chapter 2 will introduce many representative deep learning algorithms for image denoising tasks.

1.3 Motivation and Contribution

As one of the earliest works to apply NAS for image denoising, we would like to know if NAS methods can also achieve favourable performance like those in classification and segmentation tasks. Based on a differentiable architecture search algorithm, DARTS [23], with relatively low computational cost and flexibility of transferring between different datasets, we would like to search for an excellent neural network automatically that meets the following conditions:

- Fit the requirement of the image denoising task

- Experiment with affordable computational cost
- Automatic design of architectures and hyperparameters

We will use the differential architecture search method as the basis, integrated with machine learning methods including successive halving search, random search and grid search to search for repetitive cell architecture and exterior network architecture. Further, we will optimize the hyperparameters of searched architectures by taking advantage of BOHB algorithm [14]. The process of finding architectures and tuning hyperparameters are completed automatically and has shown satisfactory results on image processing datasets.

Chapter 2

Background

Most NAS routines are like this: first, define the search space, then use search strategies to find network structures, evaluate them, and conduct the following search round based on feedback. The design of search space corresponds to the development of deep neural networks. With the emergence of novel multi-path neural networks (e.g., ResNet, DenseNet), NAS researchers also began to consider more diverse structures and connections in the search space. Meanwhile, many successful networks contain repeating substructures, called cells or blocks; therefore, these structures have been considered. Cell-based search has been proposed; only the cell architecture is searched, and the overall network is formed by overlapping these cells. In this way, to reduce the search space and the searched architecture is well-performed in migrating between data sets.

Since most NAS methods have large search space, for some search strategies like evolutionary algorithms and reinforcement learning with discrete search space, the model evaluation typically involves the training of each sampled model, which leads to a lengthy search procedure. Compared with these search strategies, differentiable architecture search has a lower computational cost.

2.1 Differentiable Architecture Search

We know that if the search space is continuous and the objective function is differentiable, the gradient-based information could be applied for efficient search. Liu *et al.* proposed DARTS [23] to search a cell with optimal structure. They treated a cell as a directed acyclic graph containing several ordered nodes, and each node represents implicit representations (feature maps), and each directed edge represents an operator. The key trick of DARTS is to mix the candidate operators with the softmax function. So that the search space becomes continuous and the objective loss function becomes differentiable. They use a set of architecture weights to control the contribution of each operator and apply gradient-based optimization methods to update the weights and determine the optimal architecture. After searching, the mixed operations will be replaced by the operations with the largest weights and the resulting network is obtained. This method finds state-of-the-art models for image classification, language modelling

tasks and inspires many following works for further improvements and various applications.

Including DARTS, most differentiable NAS works focus on the cell-level architecture search that manages hierarchical computing but manually design the outer network structure that controls the spatial resolution change. For the image segmentation task that is sensitive to the change of spatial resolution, Zhang *et al.* proposed Auto-Deeplab [20] that searched the cell-level architecture as well as network-level architecture. In addition to similar cell architecture search in [20], they also constructed a hierarchical network-level search space and further softened the weights of network-level architecture and applied the Viterbi algorithm to determine the network architecture. On the other hand, differentiable NAS suffers from significant GPU memory cost issues when searching on large-scale datasets. Many alternative methods (e.g., use proxy datasets to search, designing smaller cells, training fewer iterations) have been widely applied. However, they cannot guarantee the optimal solution for the target task. To address it, Cai *et al.* developed ProxylessNAS [4], instead of using proxy datasets to search architectures for large-scale datasets, they built the search space with parallel paths and binarized the architecture parameters of each path. Therefore, they can activate only one path during the training process, allowing the direct search of architectures on the actual dataset with affordable GPU memory usage. Due to the flexibility, most NAS works choose to search cell architectures on small datasets and directly transfer them to larger datasets by stacking more cells. Regarding this, Chen *et al.* proposed P-DARTS [7] to bridge the depth gap of the network between the search and evaluation scenarios. It progressively increases the number of repetitive cells, simultaneously reduces the number of operations by search space approximation, and utilizes search space regularization to facilitate the exploitation of other operations other than skip-connect to improve the model stability. Xu *et al.* proposed PC-DARTS [31] to relieve the typical pressure of DARTS that requires significant memory and computing. It developed a channel sampling mechanism to utilize a proportion of channels for searching operations of the network, further aided with edge normalization to stabilize the performance of channel sampling.

2.2 Deep Learning for Image Denoising

Most deep learning methods for image denoising focus on the novel design of deep neural networks, although few deep learning researchers explain how they find out these architectures. There are some state-of-the-art deep learning algorithms for image denoising. Zhang *et al.* proposed DnCNN [33], which designs a deep network for learning the mapping between nosing images and residual images (i.e. noise map), with the utilization of residual learning and batch normalization to speed up the training and boost denoising performance. Zhang *et al.* proposed FFDNet [34] with the tunable noise level map and downsampled subimages as inputs to train a network, which is more flexible to denoise images with different noise levels in a network, and achieve

a good trade-off between noise removal and detail preservation. Plotz *et al.* [25] designed neural nearest neighbours (N3) block based on continuous relaxation of KNN to leverage the self-similarity information when building the neural network. Liu *et al.* [22] improved non-local operation to leverage non-local self-similarity and integrated the proposed non-local module into end-to-end training of recurrent networks. Zhang *et al.* [35] designed residual dense block to extract local features and further combined with dense feature fusion for processing these hierarchical features. Cha *et al.* proposed FC-AIDE [5], a neural network containing several filters and with adaptive receptive fields at each layer and obtained stronger and more robust adaptivity by introducing a regularization method. Ulyanov *et al.* [29] proposed to take network architecture as image prior, with random encoding as input and noisy image as a target, training the network with weights that can derive clean image in the middle iterative process. The method achieved satisfying results on different inverse problems, including image denoising, inpainting and super-resolution.

2.3 Differentiable NAS for Image Denoising

So far, the only work that applies differentiable architecture search for image denoising is proposed by Zhang *et al.* [32]. They also used cell sharing strategies and gradient-based search algorithms but supplemented with candidate searching operations with adaptive receptive fields to search the cell architecture. Unlike other networks that fixed the width of cells by hand, in this work, they built the overall network with parallel supercells with different widths at each layer, assigned a set of parameters for candidate paths and selected the best candidate path using the Viterbi decoding algorithm. So the cell architectures and their widths are both searched automatically. The main difference to our work we use machine learning methods to select both network depth and width, and we further conduct hyperparameter optimization for selected neural networks.

Chapter 3

Methods

The ultimate goal of our work is to find a well-performed neural architecture for the image denoising task. Given noisy images and clean images as inputs and targets (in Fig. 3.1), we are supposed to search for good network architecture in addition to standard network weights.

Different from traditional deep learning methods for the image denoising task that manually design the architectures of chain-structured neural networks, in this work, we search the network consisting of repeated *cells*. We apply differentiable architecture search to determine the micro-architecture, i.e., cell architecture, automatically. In the process, we integrate several AutoML methods, including grid search, random search and successive halving search, to optimize the macro-architecture hyperparameters, including the width and depth of the network. After deriving the searched networks, we perform hyperparameter optimization for learning rate, weight decay, and momentum that significantly impact network performance.

The framework of neural architecture search is illustrated in Fig. 3.2. Our pipeline consists of three primary stages: train search, train and test. In the stage of train search, we construct the supernet by stacking repeated *supercells*, i.e., cells containing all kinds of operations and connections, following gradient-based optimization methods in DARTS [23] to derive the dominant micro-architecture (subcell architecture). Different from [23] that focus on cell architecture search and only manually specify the outer network structure, in our search stage, we also search for the best combination of macro-architecture (the number and channel of cells) along with micro-architecture. We specify macro-architecture hyperparameters by grid search, random search and successive halving search and pick the best combinations of micro-and macro-architecture after searches. In the next train stage, we use the searched subcell structure and macro-architecture obtained from previous searches to build the new model and perform hyperparameter optimization for the network before training the weights from scratch. The trained model will be transferred to different testing sets to evaluate its denoising performance in the test stage.

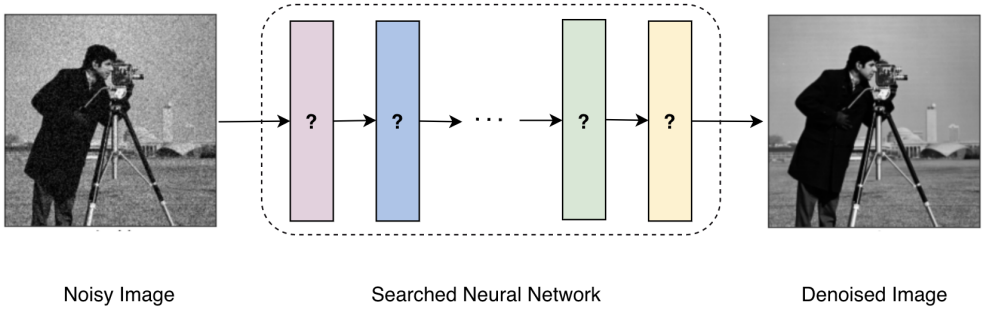

Figure 3.1: Our work aims to search for an exemplary deep neural network architecture with appropriate hyperparameters to recover clean images from noisy images. Noisy images and clean images are used as inputs and targets to search neural network architecture.

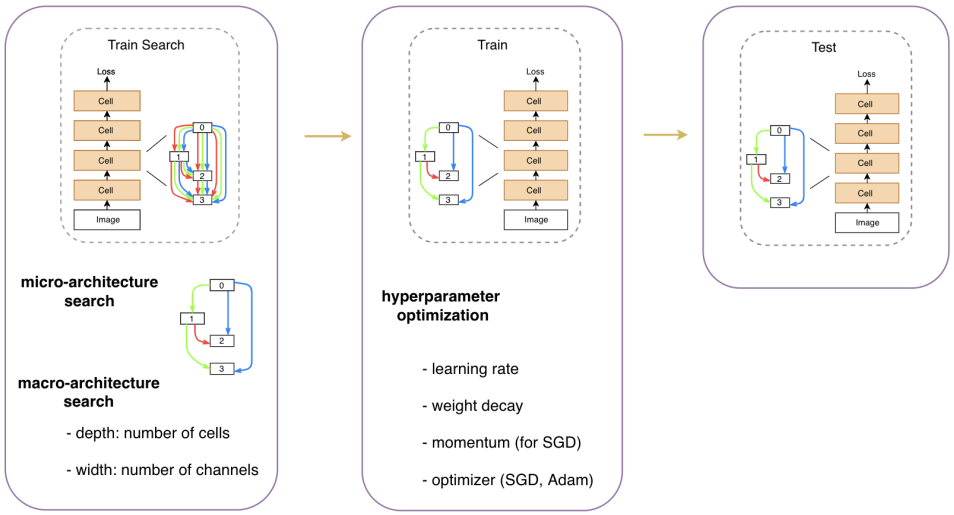

Figure 3.2: Our method consists of three primary stages. In the stage of train search, we construct the model with repeating cell structures and apply gradient-based optimization methods to derive the optimal subcell architecture. Additionally, we apply different search methods to determine the depth and width of the network. In the next train stage, we build the model upon the searched cell architecture and macro-architecture. Hyperparameter optimization for significant hyperparameters is performed before training the model weights from scratch. In the last stage, we test the model performance on several testing datasets.

3.1 Micro-architecture Search

As shown in Fig. 3.2 (train search), the network we search on consists of repeating *supercells*. The micro-architecture search is to find a good subcell architecture from the updating network. We define the search space within a supercell structure and search cell architecture using a similar strategy as in DARTS [23]. Specifically, we build a supercell that integrates all types of operations (e.g., convolution, skip-connect, none-connect) and all connections between nodes. Fig. 3.3 (left) shows an example of a supercell, which is a directed acyclic graph consisting of an ordered sequence of nodes, where each node presents a feature map and each edge represents one operation. Each edge is assigned with an architecture weight, and the architecture weights passing to the same node are softmax to make the search space continue. Within a cell, the information is transmitted between node i to node j with formula: $\bar{o}^{(i,j)}(x) = \sum_{o \in \mho} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in \mho} exp(\alpha_{o'}^{(i,j)})} o(x)$, where $\mho$ is the operation set between two nodes, x is the feature map at node i, $o(x)$ is the feature map after operation o, and $\alpha_o^{(i,j)}$ is the architecture weight associated with the operation o from node i to j. Fig. 3.3 (right) illustrates an example of feature map flow from node 0 to node 1, there are three candidate operations (i.e., skip-connect, 3 × 3 convolution, max pooling), and after separate operations, three generated feature maps are softmax to output the feature map at node 1.

Instead of sampling subnets from the supernet and evaluating each model, we only have a one-shot supernet. We are assisted by a set of architecture weights to adjust the contribution of each connection. The problem of searching neural architecture transfers into finding a set of architecture weights that minimize the validation loss. The searching dataset is split into training and validation sets. As shown in Algorithm 1, after building the supernet and a set of architecture weights, we update architecture weights on the validation set and standard network weights on the training set alternatively so that the searched network are also trained to guarantee its performance. The authors proposed an approximate scheme for evaluating architecture gradient (line 3) to get rid of expensive inner optimization, where w is the current network weight, and ξ is a learning rate for a step of internal optimization. After convergence of iterative procedure, the cell architecture is discretized by selecting operations and connections with the largest architecture weights. More details about weight optimization can be referred from DARTS [23].

Algorithm 1: DARTS – Differentiable Architecture Search

1 Build a network with architecture weights $\alpha^{(i,j)}$ for each edge (i, j);
2 **while** *not converged* **do**
3 1. Update architecture weights α by descending $\bigtriangledown_\alpha \mathcal{L}_{val}(w - \xi \bigtriangledown_w \mathcal{L}_{train}(w, \alpha), \alpha)$;
4 2. Update network weights w by descending $\bigtriangledown_w \mathcal{L}_{train}(w, \alpha)$;
5 **end**
6 Derive the cell architecture based on the updated α

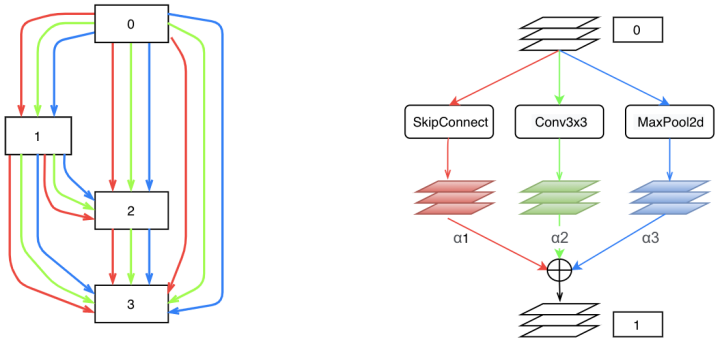

Figure 3.3: The left figure shows a cell architecture, and the right figure shows an example of information flow between nodes. Each node represents a feature map, and each edge is associated with an operation. The right figure illustrates information flow from node 0 to node 1; we generate three feature maps after applying each operation to the feature map at node 0. They are softmax with their associated architecture weights to produce the feature map at node 1.

3.1.1 Image Denoising Search Space

Different from the image classification task tackled in DARTS [23] that only have several categorical outputs, the image denoising task requires images as outputs. We did not follow the pattern of [23] by inserting *reduction cells* (cells that reduce the spatial resolution of features by two) between every two *normal cells* (cells that do not change the feature spatial resolution) to form the network. We choose not to reduce the spatial resolution of the images in order to preserve the detailed and low-level information. Therefore we only use the patterns of *normal cells* to keep the spatial resolution of images unchanged at each layer of the network and define the following candidate operations in the cell search space:

- zero operation (no connection between nodes);
- identity operation (equivalent to skip connection);
- 3×3 convolution (with padding);
- 3×3 separable convolution (with padding);
- 3×3 convolution with dilation rate of 2 (with padding);

3.2 Macro-architecture Search

Previous differentiable architecture methods always fix the macro-architecture and only search for the cell architecture. However, we argue that the hyperparameters of macro-architecture like the number of cells and channels of cells should be included in the

search stage since they are highly related to the network's capacity (number of parameters of the network). Therefore we additionally specify these two related hyperparameters when building the models for search. We use strategies including grid search, successive halving search, and random search to solve the problem.

3.2.1 Exhaustive Grid Search

We search the depth (the number of cells) and width (the channel of cells) of the overall network in the train search stage and adopt the same macro-architecture in the train and test stage. The grid of hyperparameter combinations is shown in Table 3.1, and the number of channels ranges from 8 to 64, and the number of layers(cells) ranges from 1 to 10. Considering the practical usage and affordable training for models, we exclude searching for models with 64 channels, 8 and 10 layers.

Channel \ Layer	1	2	4	6	8	10
8	✓	✓	✓	✓	✓	✓
16	✓	✓	✓	✓	✓	✓
32	✓	✓	✓	✓	✓	✓
64	✓	✓	✓	✓		

Table 3.1: Gird combinations of the number of channels and layers of candidate models.

In the train search stage, we apply grid search among these 22 candidate models. After searches, 22 corresponding cell architectures are derived will be used in the following training and testing stages.

3.2.2 Successive Halving Search

At the same time, we also employ successive halving search [16] for these models to save the running time in the train search stage. The procedure starts from training all models for several budgets (e.g., T epochs); based on the current validation loss, pick half the best models and resume training them for twice previous budgets (e.g., 2T epochs). Successively selecting half models and doubling the training time is continued until obtaining final candidate models. There is a slight deviation of our implementation from the original SH setting. We do not account for the previous running budgets when we start running selected configurations for twice budgets, so we relocate more resources for promising configurations.

3.2.3 Expanded Random Search

For previous exhaustive grid search and successive halving search, we vary the number of cells and channels of the supernet and search architectures on 22 candidate models with the same training hyperparameter setting. Considering models with different

Hyperparamter	Range
num_cell	[1, 2, 4, 6, 8, 10]
num_channel	[8, 16, 32, 64]
netw_lr	[0.001, 0.005, 0.025]
arch_lr	[1e-4, 3e-4, 6e-4]
batch_size	[24, 32, 40 , 48 , 64, 128, 256]

Table 3.2: Expanded search space involving five hyperparameters for macro-architecture search. One hundred twenty-eight configurations are sampled within the ranges to form non-repeating 128 candidates searched models.

macro-architecture, i.e., the number of cells and channels, the training hyperparameters should be varied to search for good architecture. Therefore we expand the macro-architecture search space for five selected factors: depth, width, batch_size, architecture learning rate and weight learning rate. The ranges of these categorical hyperparameters are shown in Table 3.2. Moreover, we sample unique 128 configurations for constructing 128 searched models; corresponding 128 cell architectures will be derived after the stage of train search.

3.3 Hyperparameter Optimization

After the train search stage, cell architectures with their combination of width and depth of the network will be derived. We build the new models with the searched architecture and train them on the training set. Typically, the performance of a deep neural network is sensitive to the setting of hyperparameters. Therefore tuning hyperparameters of the network is a nontrivial step. In our work, before training the searched models, we perform hyperparameter optimization for learning rate, the type of optimizer (Adam or SGD), weight decay and momentum (only for SGD optimizer).

3.3.1 BOHB

BOHB [14] is an efficient and robust hyperparameter optimization algorithm that combines the advantages of Bayesian Optimization (BO) and Hyperband (HB). It relies on HB to determine the number of configurations and the associated budgets per run. However, the speciality is to replace the random sampling of configurations at the beginning of each iteration of HB with the model-based result of BO built upon existing configurations. When the number of configurations needed for the budget is reached, a successive halving process is performed. Reversely, the performance of configurations under different budgets is used to update the BO model.

Hyperband

Hyperband [18] is an effective hyperparameter optimization method; by taking advantage of successive halving, it can be applied for picking the best configurations without

running entire budgets for each configuration. The algorithm of hyperband is given in Algorithm 2, given the minimum budget b_{min}, the maximum budget b_{max} and η which controls $1/\eta$ configurations retain in the next round, we calculate s_{max} ($s_{max}+1$ is the number of brackets) and cycle through all brackets with Hyperband. The number of configurations and the initial budget of SH for the brackets is computed in lines 5 and 6, and SH procedures are continued until discovering the best configurations.

Algorithm 2: Pseudocode for Hyperband using SuccessiveHalving (SH)

1 **Input :** budgets b_{min}, b_{max} and η
2 **Output :** selected configurations
3 $s_{max} = \log_{\eta} \frac{b_{max}}{b_{min}}$
4 **for** $s \in \{s_{max}, s_{max}-1, ..., 0\}$ **do**
5 | sample $n = \frac{s_{max}+1}{s+1} \cdot \eta^s$ configurations
6 | run SH on them with $\eta^s \cdot b_{max}$ as initial budget
7 **end**

Bayesian Optimization

The Bayesian optimization of BOHB is similar to Tree Parzen Estimator (TPE). The significant difference is that BOHB applies a multidimensional kernel density estimator (KDE) instead of one-dimensional KDEs to better deal with the interaction of hyperparameters in the search space. In TPE, they require KDE to model two densities as following:

$$l(x) = p(y < \alpha | x, D) \tag{3.1}$$

$$g(x) = p(y < \alpha | x, D) \tag{3.2}$$

In order to model effective KDEs, the minimum number of observations (N_{min}) needed for building the model is set, which is $d+1$ in experiments, where d is the dimension of the search space. After generating the first $N_{min}+2$ random selecting configurations, the number of best and worst observations are determined by:

$$N_{b,l} = max(N_{min}, q \cdot N_b) \tag{3.3}$$

$$N_{b,g} = max(N_{min}, N_b - N_{b,l}) \tag{3.4}$$

Afterwards, these two sets of observations are used for fitting density distributions $l(x)$ and $g(x)$.

BOHB Algorithm

BOHB follows the general procedure of HB, sampling several configurations, allocating initial budgets and running SH on them until reaching the maximum budgets. Instead of a random sampling of HB at the beginning of iterations, initial BOHB configuration selection is determined by the multidimensional KDE models of BO.

As shown in Algorithm 3, a small proportion of random configurations is added for more exploration and prevents the model from falling into a local minimum (line 1). Once at least $N_{min} + 2$ configurations are generated and run, the $N_{b,l}$ best and $N_{b,g}$ worst observations are divided and used for fitting KDEs models $l(x)$ and $g(x)$, respectively. According to Equation 3.2, the sets of configurations x and corresponding optimization targets y (loss or other metrics) are used to model these densities. Afterwards, N_s samples will be drawn from $l(x)$ distribution but with bandwidths multiplied by a factor of b_w to promote exploration around promising configurations (line 7). The best sample with highest ratio $l(x)/g(x)$ among N_s samples is returned for HB and further evaluation. Once reaching the maximum budget for each iteration, the configuration with the best performance is augmented to data D_b, i.e., the observations for budget b, and refit the KDEs model for each budget. BO guides hyperparameter selection of HB, while the results of HB are used for tuning the BO model.

Algorithm 3: Pseudocode for sampling in BOHB [14]

1 **Input :** observations D, fraction of random runs ρ , percentile q, number of samples N_s, minimum number of points N_{min} to build a model, and bandwith factor b_w
2 **Output :** next configuration to evaluate
3 **if** $rand() < \rho$ **then return** random configuration
4 $b = argmax \{D_b : |D_b| \geq N_{min} + 2\}$
5 **if** $b = \emptyset$ **then return** random configuration
6 fit KDEs: $l(x)$ and $g(x)$
7 draw N_s samples according to $l(x)$, but with all bandwidths multiplied by b_w
8 **return** sample with highest ratio $l(x)/g(x)$

Chapter 4

Experiments and Results

4.1 Dataset and Data Processing

Ideally, we are supposed to conduct searching and training of the architectures on the training set. We follow [8, 33] to use the Train400 dataset with 400 grayscale images of size 180 × 180 for training and clip the large image into small patches of size 40 × 40 to fit the memory of GPUs. However, direct searching on the complete training set is computationally expensive, especially for 22 candidate models and those models with large channels and more cells. Therefore we adopt MNIST and CIFAR100 as the proxy sets to search the cell architecture before training the models on the Train400 training dataset. Currently, Set12 and BSD68 from the Berkeley segmentation dataset are used as our testing sets. Fig. 4.1 provides a visualization of these datasets.

For these datasets with clean images, we generate noisy images by adding additive white gaussian noise with known noise level (e.g., σ = 25) and train the models with noisy and clean images as the inputs and the targets, respectively.

CIFAR100 is a small dataset with a more realistic scenario than MNIST and visually more similar to our training and testing sets. However, it is a colourful dataset, so we convert it into grayscale by processing RGB channels with the formula: $0.299R + 0.587G + 0.114B$. The visualization of CIFAR100 and transformed grayscale CIFAR100 can be seen from Fig. 4.2.

4.2 Evaluation Metrics

To quantify the denoising quality of images, we use two widely used metrics: Peak Signal-to-noise Ratio (PSNR) and Structural Similarity Index (SSIM). In addition, we use validation loss, i.e., Mean Square Error (MSE), as a criterion for selecting architectures.

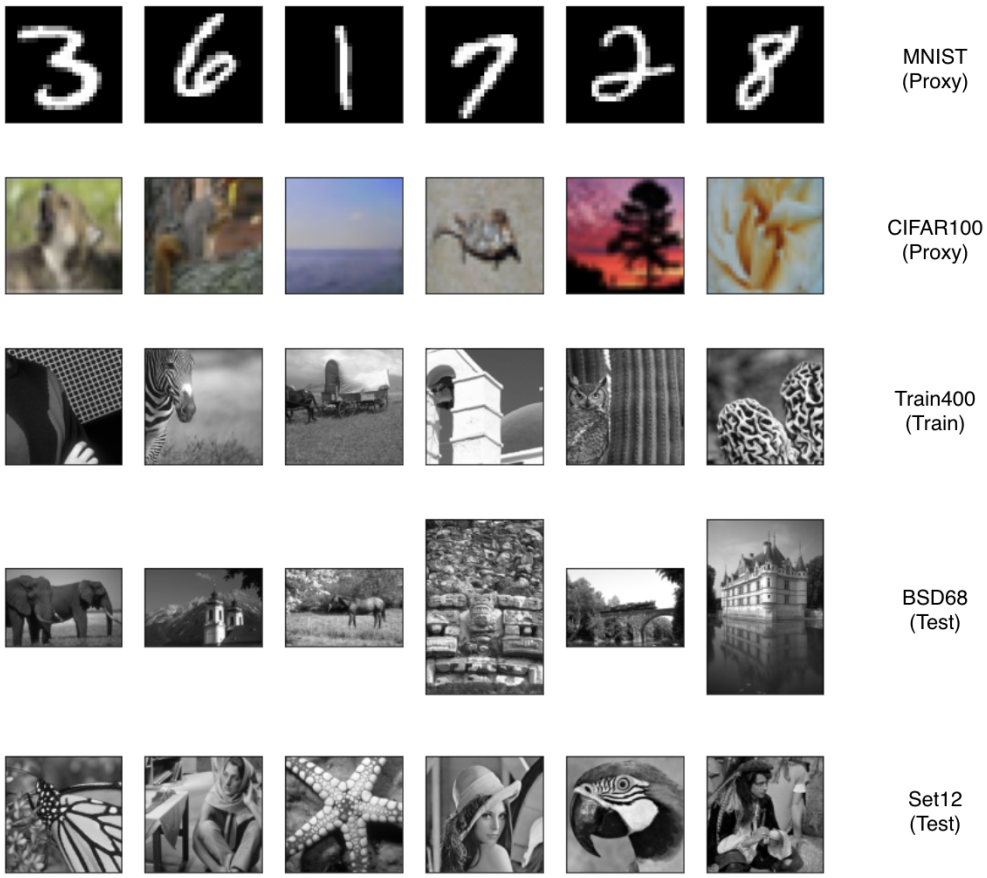

Figure 4.1: Example of images from datasets used in our experiments: MNIST and CIFAR100 are proxy datasets for searching the architecture. Train400 is our training sets with 400 180 × 180 images. BSD68 and Set12 are testing sets with 68 and 12 images of varied sizes.

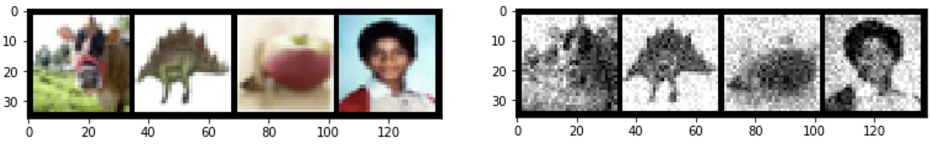

Figure 4.2: CIFAR100 dataset visualization (left) and its transformed grayscale version (right). The images look a bit fuzzy since they are reshaped to the size of 32 x 32 by default.

Given an $m \times n$ noise-free image X and its denoising approximation Y, the MSE is defined as:

$$MSE = \frac{1}{mn} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [X(i,j) - Y(i,j)]^2 \tag{4.1}$$

PSNR is an engineering term that expresses the ratio of the maximum possible power of a signal to the destructive noise power that affects its accuracy.

$$PSNR = 10 \cdot \log_{10} (\frac{MAX_X^2}{MSE}) \tag{4.2}$$

where MAX_X is the maximum possible pixel value of the image.

SSIM is an index to measure the similarity of two digital images, and it is more in line with the human eye's judgment of the image quality.

$$luminance : l(x,y) = \frac{2\mu_x\mu_y + C_1}{\mu_x^2 + \mu_y^2 + C_1}, \tag{4.3}$$

$$contrast : c(x,y) = \frac{2\sigma_x\sigma_y + C_2}{\sigma_x^2 + \sigma_y^2 + C_2}, \tag{4.4}$$

$$structure : s(x,y) = \frac{\sigma_{xy} + C_3}{\sigma_x\sigma_y + C_3}, \tag{4.5}$$

$$SSIM(x,y) = [l(x,y)]^\alpha [c(x,y)]^\beta [s(x,y)]^\gamma \tag{4.6}$$

where μ_x, μ_y, are the mean of images X and Y, $\sigma_x.\sigma_y$ are the standard deviation of images, and σ_{xy} is the covariance of images. C_1, C_2, C_3 are small constants to assure numerical stability when the denominators are close to zero at low luminance and contrast regions. Following [30], they are set to 1e-4, 9e-4 and 4.5e-4, respectively. α, β, and γ are parameters used for tuning the importance of luminations, contrast and structure components. We set them all to one for simplification by following [30].

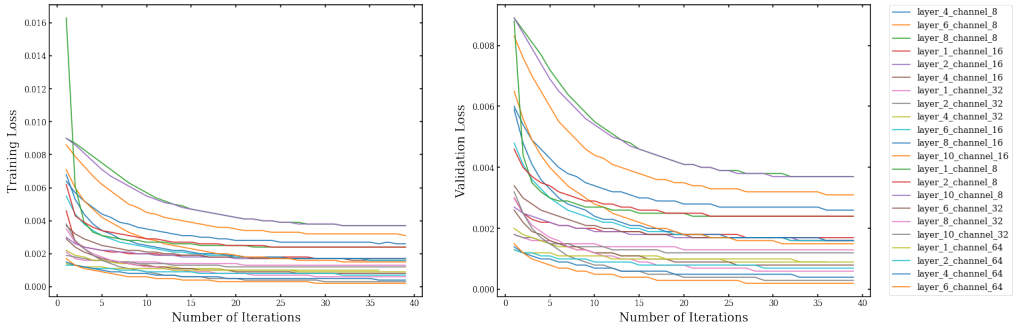

Figure 4.3: Exhaustive grid search on 22 candidate models. The plot shows these models' training loss (left) and validation loss (right) during searching. The model with six layers and 64 channels has the smallest validation loss.

4.3 Experiments

4.3.1 Cell Architecture Search

Exhaustive Grid Search on 22 Candidate Models

We search the cell architecture on 22 candidate models with MNIST as the proxy. In the train search stage, we train the architecture weights of the network and discretize inner cell architecture from the network that has the minimum validation loss. Each candidate model is trained (searched) for 40 epochs. The exhaustive training curves of these models are shown in Fig. 4.3. The model with *layer*_6_*channel*_64 has the smallest validation loss.

Same as Darts [23], we use SGD optimizer to update network weights with a learning rate of 0.025 and momentum of 0.9; CosineAnnealingLR as the scheduler for network weights; and Adam optimizer to update architecture weights with a learning rate of 3e-4 and weight decay of 1e-3.

Successive Halving Search on 22 Candidate Models

To speed up the searching process, we apply successive halving search on these candidate models on MNIST; the halving procedure is illustrated in Fig. 4.4. It starts from searching on all candidate models, but successively half the number of models at the end of 1st, 3rd, 7th, 15th, 31st epoch according to their validation loss at these epochs and resume training of selected models. The hyperparameter setting is the same as those in exhaustive grid search. The selected models are *layer*_6_*channel*_64 and *layer*_4_*channel*_64. The implementation of our SH slightly deviates from the default SH setting. We do not account for the previous running budget when running configurations for twice budgets. Therefore, more resources are allocated for configurations before the selection.

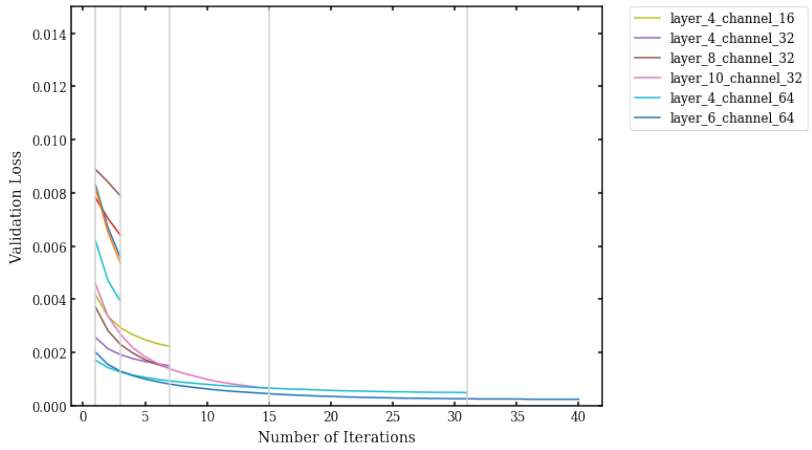

Figure 4.4: Successive halving search on 22 candidate models for fast searching. At the end of the 1st, 3rd, 7th, 15th, 31st epoch, the half of models are preserved for resumed training. The model with six layers and 64 channels is optimal at the stage of train search.

Conclusion: The experiments of grid search and successive halving search lead to the same result, that is, to select cell architecture searched on the *layer*_6_*channel*_64 model for further training (if we only pick one cell architecture).

Expanded Random Search on 128 Candidate Models

For grid search and successive halving search, 22 candidate models share the same set of training hyperparameters. We also run the experiments to expand the search space and vary the hyperparameters of different networks in the search stage. We randomly sample 128 unique configurations from the options of hyperparameters: number of cells: [1, 2, 4, 6, 8, 10]; number of channels: [8, 16, 32, 64]; learning rate of networks: [0.001, 0.005, 0.025]; learning rate of architectures: [1e-4, 3e-4, 6e-4]; batch_size: [24, 32, 40 , 48 , 64, 128, 256].

After sampling configurations, corresponding 128 candidate models are built for search. Based on their smallest validation loss, shown in Fig. 4.5, we select the best five candidate models and the derived cell architectures with combinations of network macro-architectures. The information of Top5 models, i.e., index of the selected model, setting of hyperparameters, validation loss, training loss and which epoch generates the best architecture among 40 iterations, is shown in Table 4.1.

4.3.2 Weight Training of 22 Searched Models

After an exhaustive grid search on candidate models, independent 22 inner cell architectures are discovered. We keep the combination of width and depth of network unchanged but replace the supercell with the discovered cell architecture to build the new models. New models are trained from scratch on the Train400 training set. Fig. 4.6

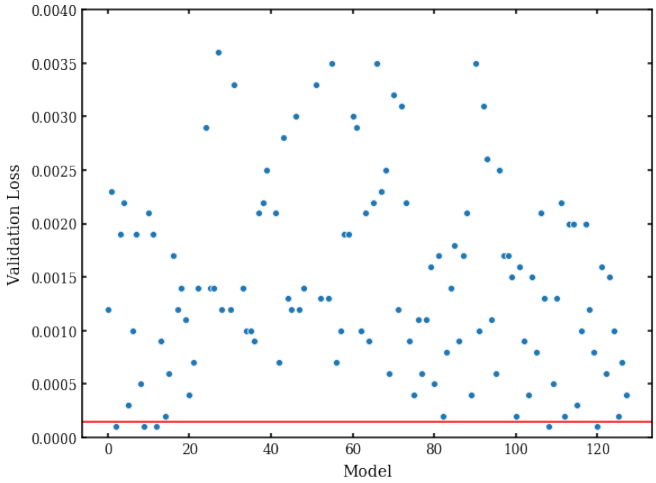

Figure 4.5: The minimum validation loss of expanded 128 models in the search stage. Top5 candidate models with searched cell architectures are selected according to their smallest validation loss.

Index	Model Name	Valid Loss	Train Loss	Epoch
2	ly_10_cn_32_bs_32_wlr_0.05_alr_0.0006	0.0001	0.0001	20
7	ly_8_cn_32_bs_40_wlr_0.05_alr_0.0003	0.0001	0.0002	38
14	ly_6_cn_64_bs_24_wlr_0.05_alr_0.0001	0.0001	0.0002	14
108	ly_10_cn_32_bs_32_wlr_0.05_alr_0.0001	0.0001	0.0002	19
120	ly_6_cn_64_bs_24_wlr_0.05_alr_0.0003	0.0001	0.0001	14

Table 4.1: The selected Top5 models with information. From left to right are the model index among 128 models, the model names indicating the searched hyperparameters, the smallest validation loss, the smallest training loss and which epoch generates the best model.

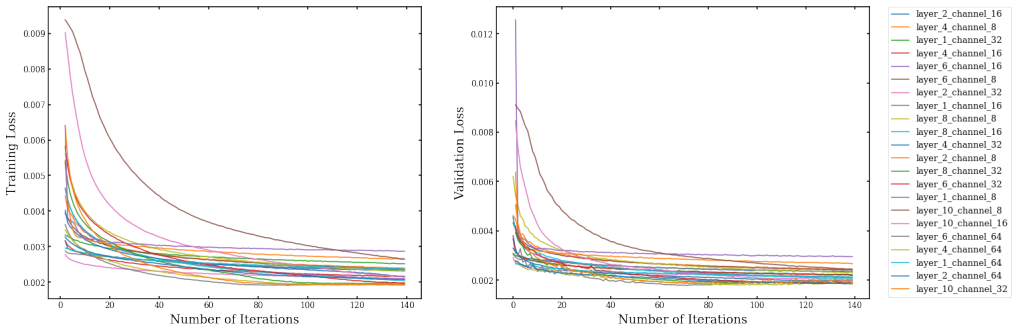

Figure 4.6: Exhaustive training on 22 candidate models. Twenty-two new models are built with the searched cell architecture. After training for 140 epochs, we pick the final model with the smallest validation loss for each model. The model with six layers and 64 channels has the smallest validation loss at the training stage.

shows the training curves of these models for 140 epochs. The model with six layers and 64 channels has the smallest validation loss.

The setting of hyperparameters is the same as in DARTS [23], which uses SGD optimizer, CosineAnnealingLR scheduler, a learning rate of 0.025, momentum of 0.9, and weight decay of 3e-4.

4.3.3 Evaluation of 22 Searched Models

After training, 22 searched models are tested on Set12 and BSD68 datasets, Fig. 4.7 and Fig. 4.8 give an overview of their PSNR performance on these two datasets. The number of channels groups the left figure, and the number of cells groups the right figure. (The SSIM performance could be seen from Fig. A.1 and Fig. A.2.)

It could be discovered that the model with *layer*_6_*channel*_64 has the best test performance, which is also the model that has the best performance in the train and train search stage (both grid search and successive halving search). In addition, deeper models not always outperform shallower models (eg: model with *layer*_8_*channel*_8 and *layer*_10_*channel*_8, or model with *layer*_8_*channel*_16 and *layer*_10_*channel*_16). The performance of models has the same trend on the two datasets.

4.3.4 Correlation of Model Performance at Different Stages

In practice, it is not ideal for applying exhaustive grid search and thorough training of all candidate models. Therefore we would like to investigate if it makes sense to pick the best (or TopK) models from successive halving searches and only train these models and any correlation between the searching and testing performance.

The correlation of models' performance with MNIST as the proxy is illustrated in Fig. 4.9. In each subplot, a dot represents a model, the top row shows the correlation of the train search with the test, the middle row presents the train with the test, and the bottom row shows the performance of the train against the train search. (There is only one subplot

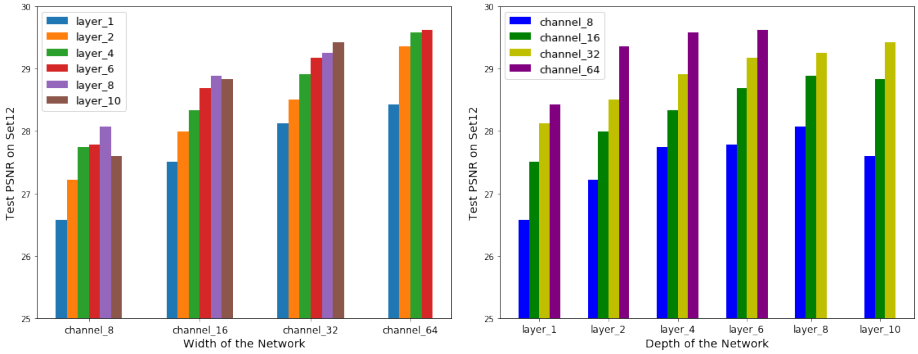

Figure 4.7: Test PSNR performance on Set12. The left figure groups models in terms of the number of channels, and the right figure groups them by the number of layers. Deeper models do not always perform better than shallower ones. For example, models have eight layers and ten layers with eight channels and ten channels.

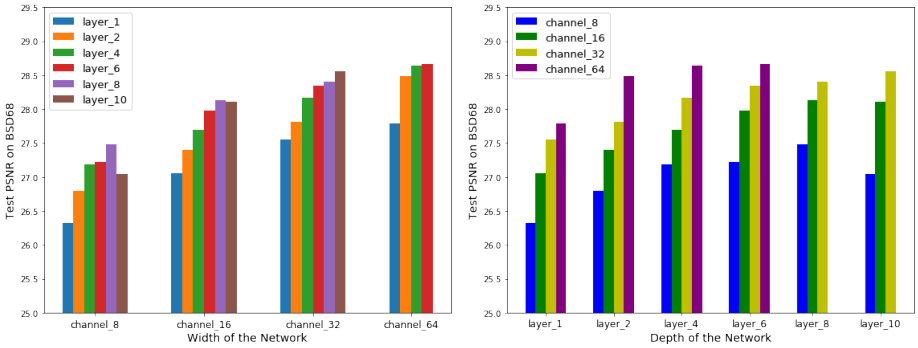

Figure 4.8: Test PSNR performance on BSD68. The left figure groups models in terms of the number of channels, and the right figure groups them by the number of layers. The performance of models has the same trend as in Set12.

since we search and train the networks on one dataset but test them on two datasets). The first row shows there is no solid linear correlation of performance at the stage of search and test on both datasets. For example, models with higher validation loss at the search stage may perform better after the training. However, the best model in the search stage still maintains the best in evaluation. Results of employing CIFAR100 as a proxy dataset further verified the statement. From the first row of Fig. 4.10, with CIFAR100 as the proxy, architecture searched on the model with the smallest validation loss in the search stage only ranked 6th among these models in evaluation. However, the model with the 2nd smallest validation loss in the search phase behaved best in testing. In these two cases, the best model in evaluation is among the TopK models in the search stage. However, the middle rows present a solid linear correlation of performance at the train and test stage. One possible reason is that the models used in the train and test stage are the same, while we use the supernet instead of the searched subnet in the train search stage.

Conclusion: There is a performance gap between the search and evaluation scenarios, the best model in the search stage is not always the best one in the testing stage. We should be cautious of applying successive halving searches at the search stage. At least TopK models should be taken into consideration for further training. Alternatively, we can conduct a complete search on all candidate models and only apply successive halving searches at the training stage.

4.3.5 Performance of Models with Different Proxies

The comparison of the performance of 22 candidate models with different proxies is illustrated in Fig. 4.11. For some models, using CIFAR100 as a proxy performs better than using MNIST, but the best model is searched on MNIST. One possible reason that models searched on CIFAR100 does not overwhelm those searched on MNIST could be the default scaling of images to 32 x 32, which blurred the pictures as shown in Fig. 4.2. It is noted that CIFAR100 is commonly used for classification tasks that distinguish a limited number of categories. Therefore, the default scaling may not significantly impact the classification performance; however, in our task, blurring brings random noise to the images, which might influence the performance of searched networks.

In addition to these two proxies, we also random sample 60000 clipped 40×40 patches from the actual training set Train400 to search architectures. The comparison with the two proxies above is provided in Fig. A.4. The subset of the training set does not show a noticeable advantage over the other two proxies, which indicates that MNIST serves as a good proxy.

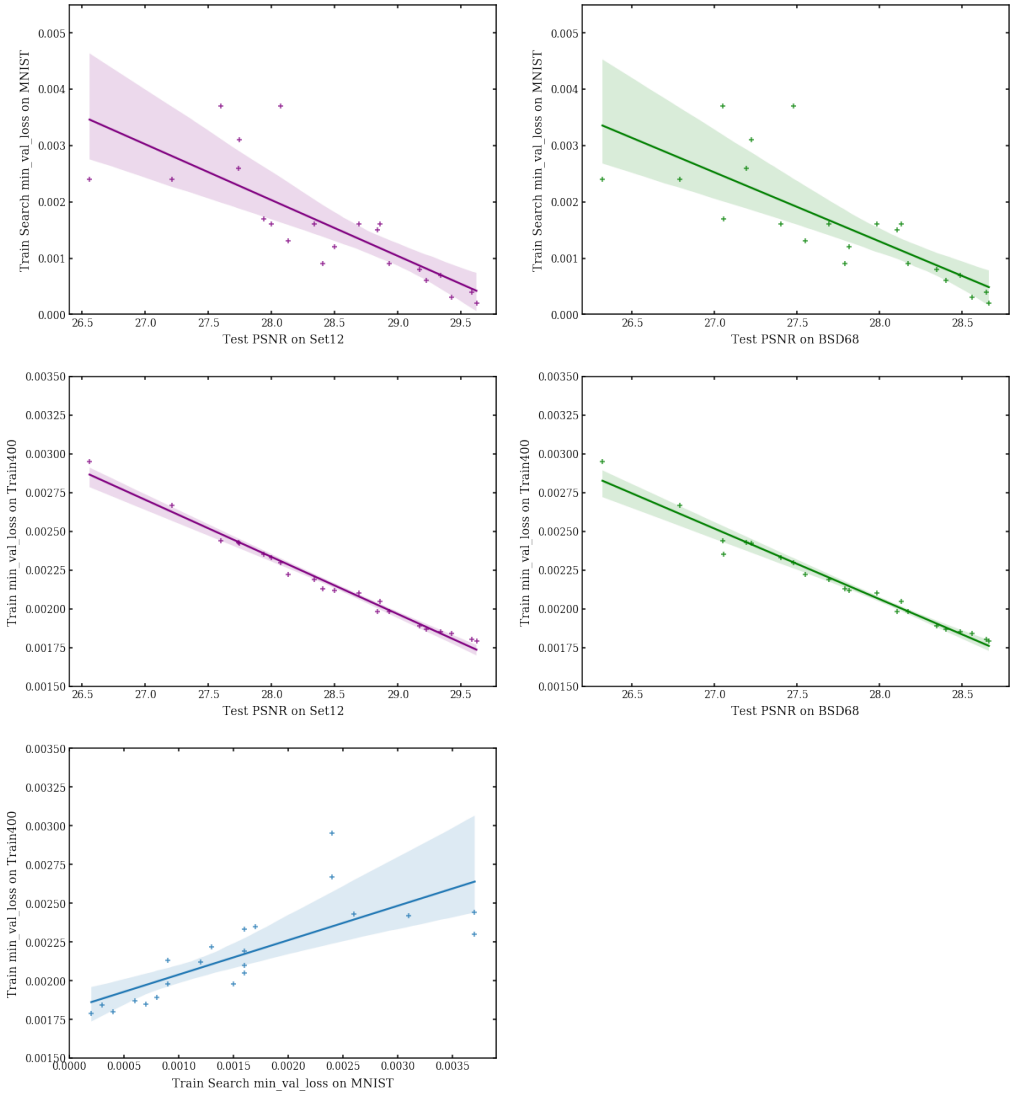

Figure 4.9: The correlation of model performance at different stages (with MNIST as proxy). From top to bottom, it shows the performance of search vs test, train vs test, and train vs search. Every dot represents a model, and the model with the smallest validation loss at the search stage is the model with the best performance in evaluation.

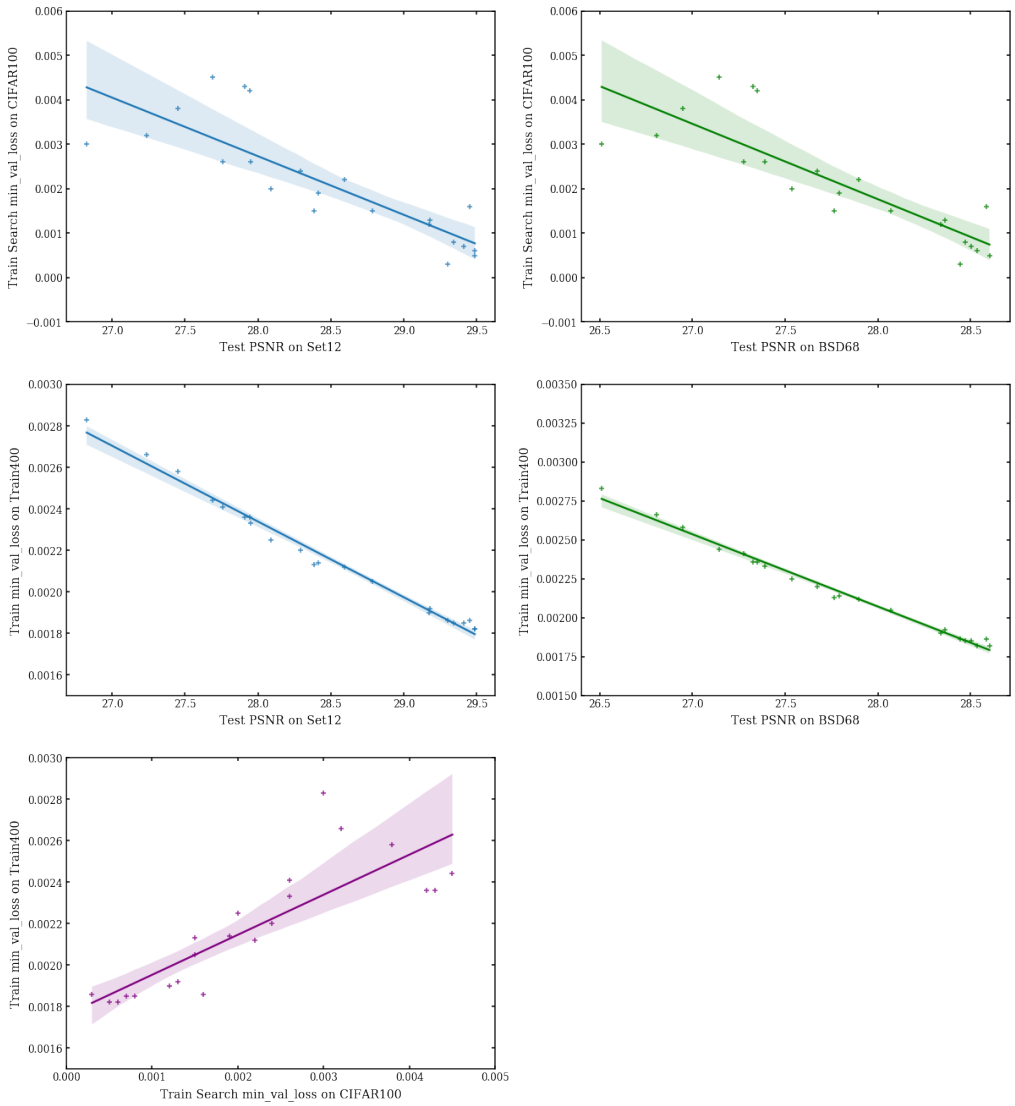

Figure 4.10: The correlation of model performance at different stages (with CIFAR100 as proxy). From top to bottom, it shows the performance of search vs test, train vs test, and train vs search. Every dot represents a model, and the model with the 2nd smallest validation loss at the search stage is the model with the best performance in evaluation.

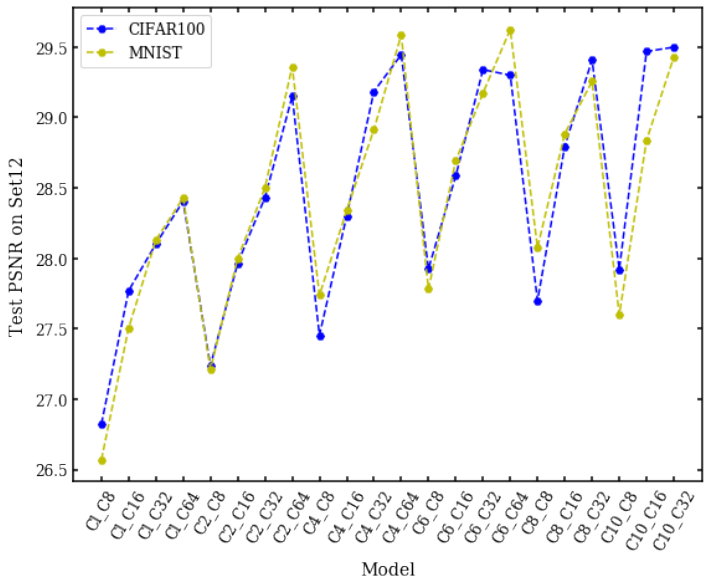

Figure 4.11: Test PSNR performance of models using CIFAR100 and MNIST as the proxy dataset. The best model (*C6_C64*: model with six cells and 64 channels) is searched on MNIST, while CIFAR100 also outperformed MNIST on some candidate models.

4.3.6 Flexible Search and Training with Optimization

Default Weight Training of Top5 Models

As aforementioned, we adopt a more flexible search by varying the number of cells, the number of channels, the learning rate of the network, the learning rate of architecture and batch_size and build 128 supernets for architecture search. After deriving the best five cell architectures from 128 searched models (Fig. 4.13), we build five new models with the searched cell architectures and keep the same set of width and depth as in searched models.

As before, we apply the default training hyperparameter setting in [23], i.e., SGD optimizer, a learning rate of 0.025, momentum of 0.9, and weight decay of 3e-4, to train the five models. The experiments are repeated five times, and the averaged test PNSR and SSIM curves of models on Set12 during training iterations are given in Fig. 4.12. Final evaluation metrics boxplots on Set12 and BSD68 are provided in Fig. 4.13. Model M14 has the best performance on two datasets with two evaluation metrics.

Training of Top5 Models with Optimization

At the same time, we also apply BOHB to optimize training hyperparameters before weight training. Involved hyperparameters are learning rate, type of the optimizer (SGD or Adam), weight decay and momentum (only applicable for SGD). Table 4.2 shows the designed search space for these hyperparameters. Optimizer is a categorical hyperparameter, and others are uniform float hyperparameters.

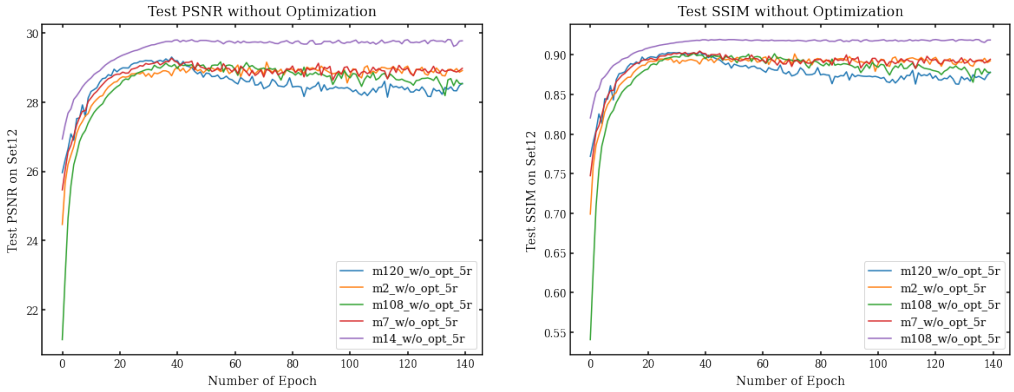

Figure 4.12: Averaged (five-run) PSNR and SSIM training curves of five models on Set12. The setting of hyperparameters is the same as in [23]. Therefore we name it to test PSNR and SSIM without optimization.

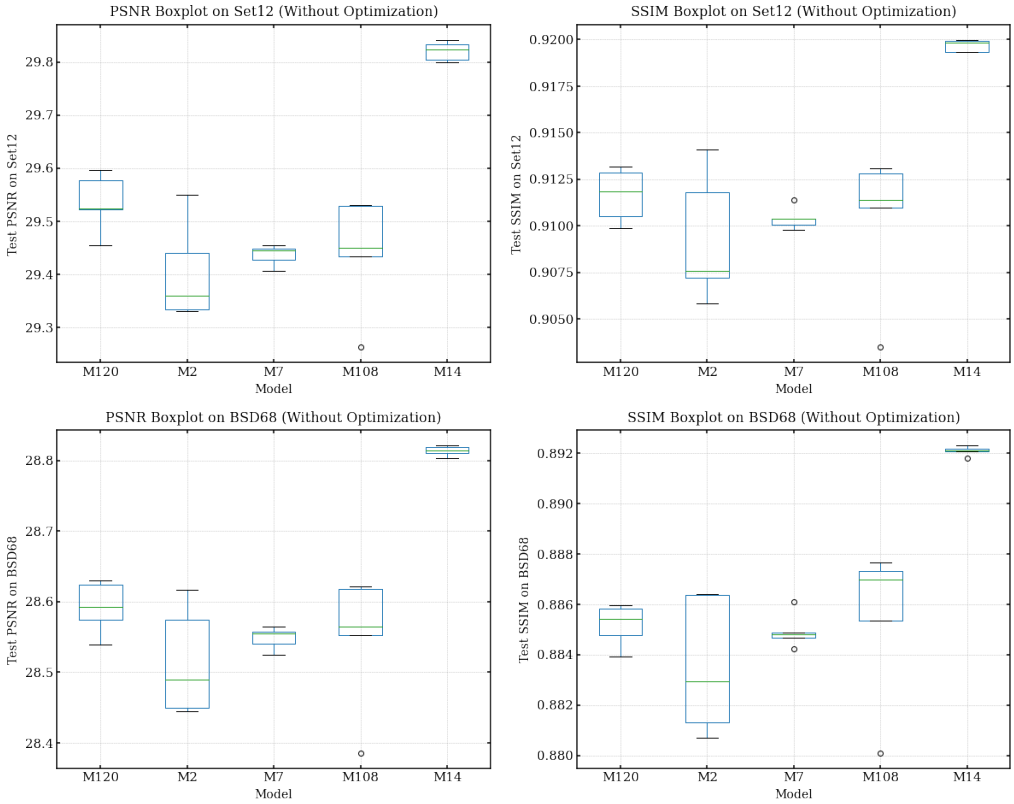

Figure 4.13: Boxplot of evaluation metrics on Set12 (top) and BSD68 (bottom) after training models using the default setting of hyperparameters. The performance difference between models is not significant. M14 has the best performance without optimization.

Hyperparameter	Range	Default Value	Log-Transform
Optimizer	[Adam, SGD]	-	-
Learning Rate	[1e-5, 1e-1]	1e-2	True
Weight Decay	[1e-5, 1e-1]	1e-4	False
SGD-Momentum	[0.0, 0.99]	0.9	False

Table 4.2: Search space of hyperparameters in the training stage.

BOHB allocates resources for sampled configurations based on its parameter settings. Therefore we specify values of parameters including the min_budget, max_budget and num_iterations based on the default setting (in Table 4.3). When we set the min_budget of and max_budget to 3 and 27 epochs, the numbers of configurations n_i and their running resources r_i could be seen from Table 4.4. It starts from sampling nine models (from the results of BO models) and running them for three epochs, then gradually successive halving the number of configurations and running more budgets, and s represents an iteration. If we set the min_budget as 3, max_budget as 27 and the num_iterations as 5 (setting abbreviation: 3_27_5), BOHB will train sampled models following the sequence (s=2, s=1, s=0, s=2, s=1), and will sample 35 models in total.

Parameter	Default	Description
eta	3	1/eta of configurations are advanced to the next round of SH
min_budget	0.01	minimum budget of BOHB
max_budget	1	maximum budget of BOHB
min_points_in_model	None	minimum number of observations required for building a KDE
top_n_percent	15	percentage of obersevations that are considered as good
num_samples	64	number of samples to optimize the expected improvement
random_fraction	1/3	fraction of random configurations sampled without KDE prior
bandwidth_factor	3	samples from a widened l(x) with bandwidth multiplying by the factor
min_bandwidth	1e-3	minimum bandwidth of a KDE model

Table 4.3: Default parameter settings of the BOHB algorithm [14]. On top of which, we specify the values of min_budget, max_budget and the num_iterations to be performed in this run.

Fig. 4.14 shows examples of models training with SGD and Adam optimizers with different learning rates. In the left figure, there are experiments on model M2 with the default hyperparameter setting in [23]; the default setting and the Xavier initialization [15]; SGD with learning rate 3e-2; and SGD with a very small learning rate 3e-5. It can be seen that hyperparameter settings significantly impact evaluation performance. The

	s = 2		s = 1		s = 0	
i	n_i	r_i	n_i	r_i	n_i	r_i
0	9	3	3	9	1	27
1	3	9	1	27		
2	1	27				

Table 4.4: The number of samples and resources allocated for BOHB with a minimum budget of 3 and a maximum budget of 27.

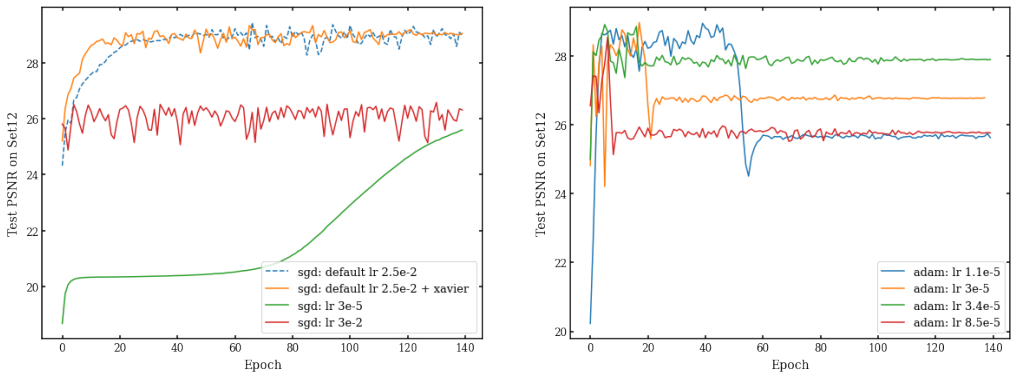

Figure 4.14: Test PSNR curves in the training stage on M2 with different optimizers and hyperparameter settings. The left ones are optimized by SGD optimizer, and the right ones are optimized by Adam optimizer.

Model	M2	M7	M14	M108	M120
Number of Parameters	824738	659618	1965890	824738	1965890

Table 4.5: The number of parameters of five models.

right figure shows the experiments with BOHB optimized hyperparameters in different parameter settings. In experiments, the algorithm always chooses Adam as the optimizer due to its faster convergence speed than SGD within limited allocated resources. However, Adam has the issue of learning only a few epochs (in Fig. 4.14 right). Considering the SGD optimizer with default hyperparameters has outperformed Adam, and the SGD can learn continuously, we decide to remove the type of optimizer from our search space. In the subsequent experiments, we only search training hyperparameters for the SGD optimizer.

We consider the number of parameters of the models (Table 4.5) and corresponding running time when specifying the parameters for BOHB optimization. For example, the model M2 has 824738 parameters and requires 20 minutes to train an epoch on a GeForce GTX 1080 Ti GPU. When setting the min_budget, max_budget, num_iterations of BOHB to 3_27_5, $27 * (3 + 2 + 1 + 3 + 2)/(3 * 24) \approx 4.13$ GPU days are required to complete the optimization. For the five models, we first set BOHB parameters to 3_27_5 to optimize hyperparameters. Fig. 4.15 records validation losses grouped by budgets of each model during optimization. Each dot represents one model with a sampled configuration in each subplot, and 35 configurations per model are sampled in the process. The configuration with the smallest validation loss is selected for training the corresponding model. After training them on the training set Train400, models are evaluated on test datasets.

In addition to BOHB parameter setting 3_27_5, we also specify another set of parameters as 3_27_15, which increases the number of iterations from 5 to 15, and the number of sampled configurations of each model increases from 35 to 90. The optimization

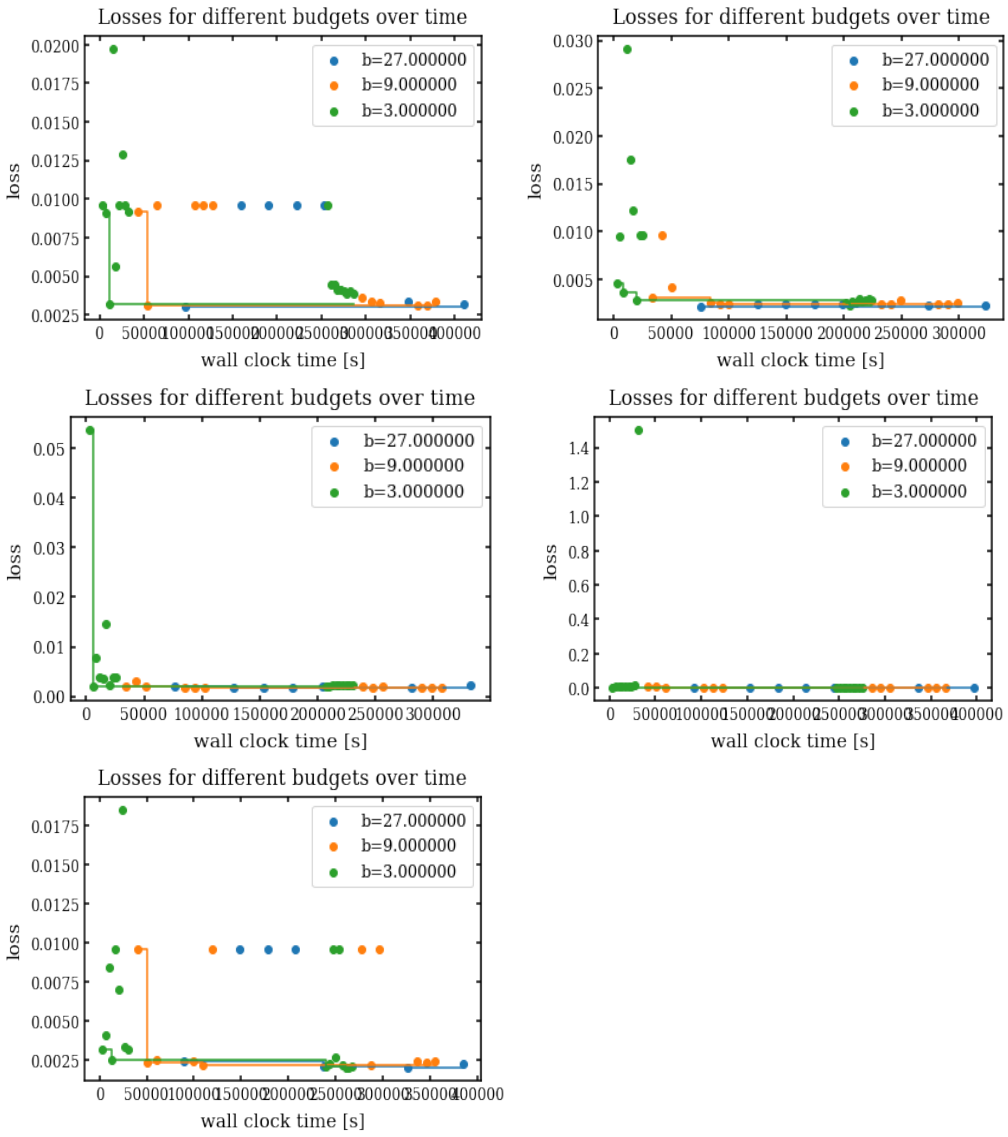

Figure 4.15: The observed validation losses of five models grouped by budget during BOHB optimization with parameters 3_27_5. From left to right are losses of models M2, M7, M14, M108 and M120.

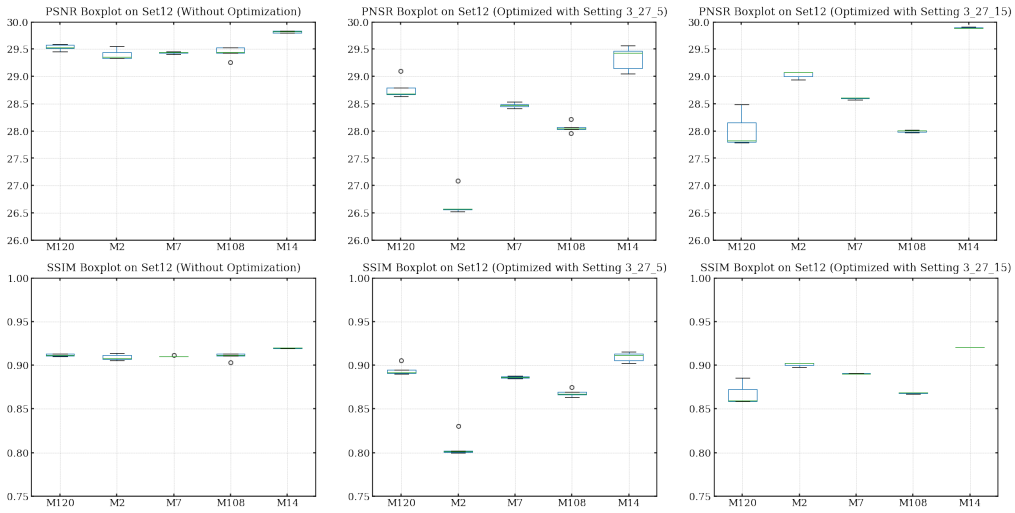

Figure 4.16: Comparison of test performance on Set12 before optimization (left), with BOHB optimization in setting 3_27_5 (middle) and with BOHB optimization in setting 3_27_15 (right).

period of each model also increases, e.g., 11.25 GPU days for model M2. Evaluation metrics boxplots of these models before and after BOHB optimization are provided in Fig. 4.16 and Fig. 4.17. The left columns are the test results with the default training hyperparameters. The middle columns are the results of models with BOHB optimized hyperparameters in setting 3_27_5. The right columns are the results trained with BOHB optimized hyperparameters in setting 3_27_15. Comparing middle columns with left columns, we could discover the optimized results are not as good as those before optimization, indicating current searched configurations are not good enough. Comparing BOHB optimized results in setting 3_27_15 with those optimized in setting 3_27_5, we see that providing more running time helped BOHB select better hyperparameter configurations for M14 and M7.

Furthermore, M14 with searched hyperparameters outperformed the model with hyperparameters stated in literature [23]. However, it cannot guarantee better configurations for all five models, which we attribute to the limited allocated computing resources. Even 90 configurations are sampled for each model. After allocating them to three budgets, on average, only 30 configurations are used to model densities l(x) and g(x), which may not be sufficient. We also provide the statistical evaluation performance results in Table 4.6. Models with hyperparameters optimized from BOHB setting 3_27_15 surpass the models with hyperparameters stated in literature [23].

4.4 Visualization

The visualization of some denoised images are provided in Fig. 4.18 and Fig. 4.19. The left columns are our synthetic Gaussian noise images. The middle columns are the denoised images from the network output. The right columns are original clean images

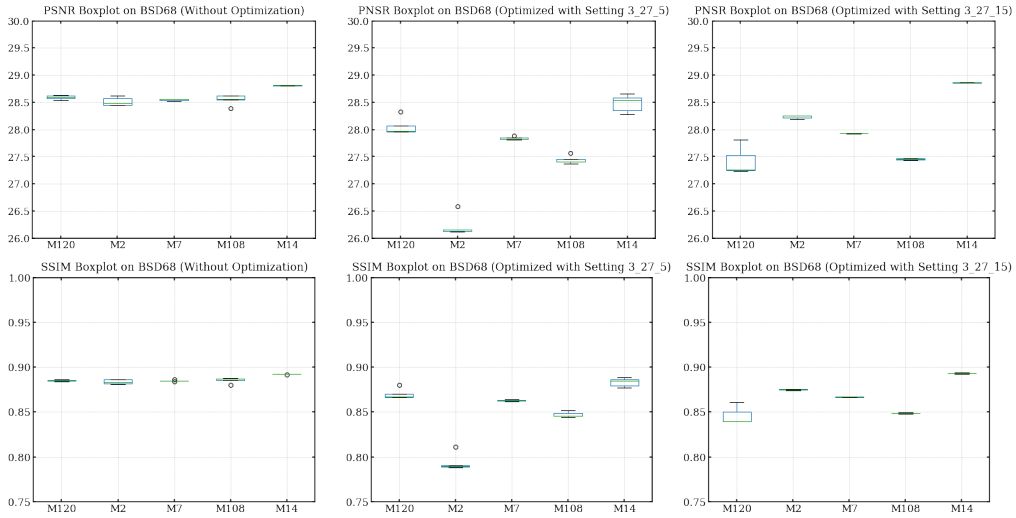

Figure 4.17: Comparison of test performance on BSD68 before optimization (left), with BOHB optimization in setting 3_27_5 (middle) and with BOHB optimization in setting 3_27_15 (right).

	w/o opt	w/ opt (3_27_5)	w/ opt (3_27_15)
PSNR on Set12	29.820 ± 0.018	29.335 ± 0.221	**29.893 ± 0.014**
PSNR on BSD68	28.814 ± 0.007	28.481 ± 0.162	**28.860 ± 0.003**
SSIM on Set12	0.920 ± 0.000	0.910 ± 0.005	**0.921 ± 0.000**
SSIM on BSD68	0.892 ± 0.000	0.883 ± 0.005	**0.893 ± 0.001**

Table 4.6: Model evaluation performance on Set12 and BSD68 before and after BOHB optimization with different settings. After BOHB optimization with setting 3_27_15, the model performance slightly improves upon the setting in literature [23].

Methods	Ours	BM3D	WNNM	EPLL	MLP	CSF	TNRD	DnCNN
BSD68	28.86	28.57	28.83	28.68	28.96	28.74	28.92	**29.23**

Table 4.7: PSNR results of different models on BSD68 and noisy images are with noisy level 25.

of the datasets. The overall noise has been removed from the noisy images. Generated denoised images preserve most details and textures in the source images. And the image quality is high and clear. We also notice that small details are missing after denoising in some examples, e.g., the peppers and house image.

4.5 Comparison with state-of-the-art

We compare the denoising performance of our model with several state-of-the-art gray-scale Gaussian denoising algorithms, including block matching 3D (BM3D) [10], weighted nuclear norm minimization (WNNM) [28], expected patch log likelihood (EPLL) [38], multi-layer perception (MLP) [3], cascade of shrinkage fields (CSF) [27], trainable non-linear reaction diffusion (TNRD) [9] and a deep neural network DnCNN [33]. From Table 4.7, it can be seen that using our model results in 0.29 dB, 0.03 dB, 0.18 dB, 0.12 dB PSNR improvements over BM3D, WNNM, EPLL and CFS on BSD68. From Table 4.8, our model shows 0.201 dB and 0.056 dB PSNR improvements over EPLL and CFS on Set12.

We also notice that MLP, TNRD and DnCNN perform better on both datasets than our model, in which MLP and DnCNN are deep learning networks and TNRD is a nonlinear diffusion model. These models are built and trained on the training dataset with the parameter or setting tuning relying on training data in experiments. While in our work, we directly transfer the resulting network searched on the proxy dataset to the training set for weight training, which may not mine as much training dataset information as these methods do.

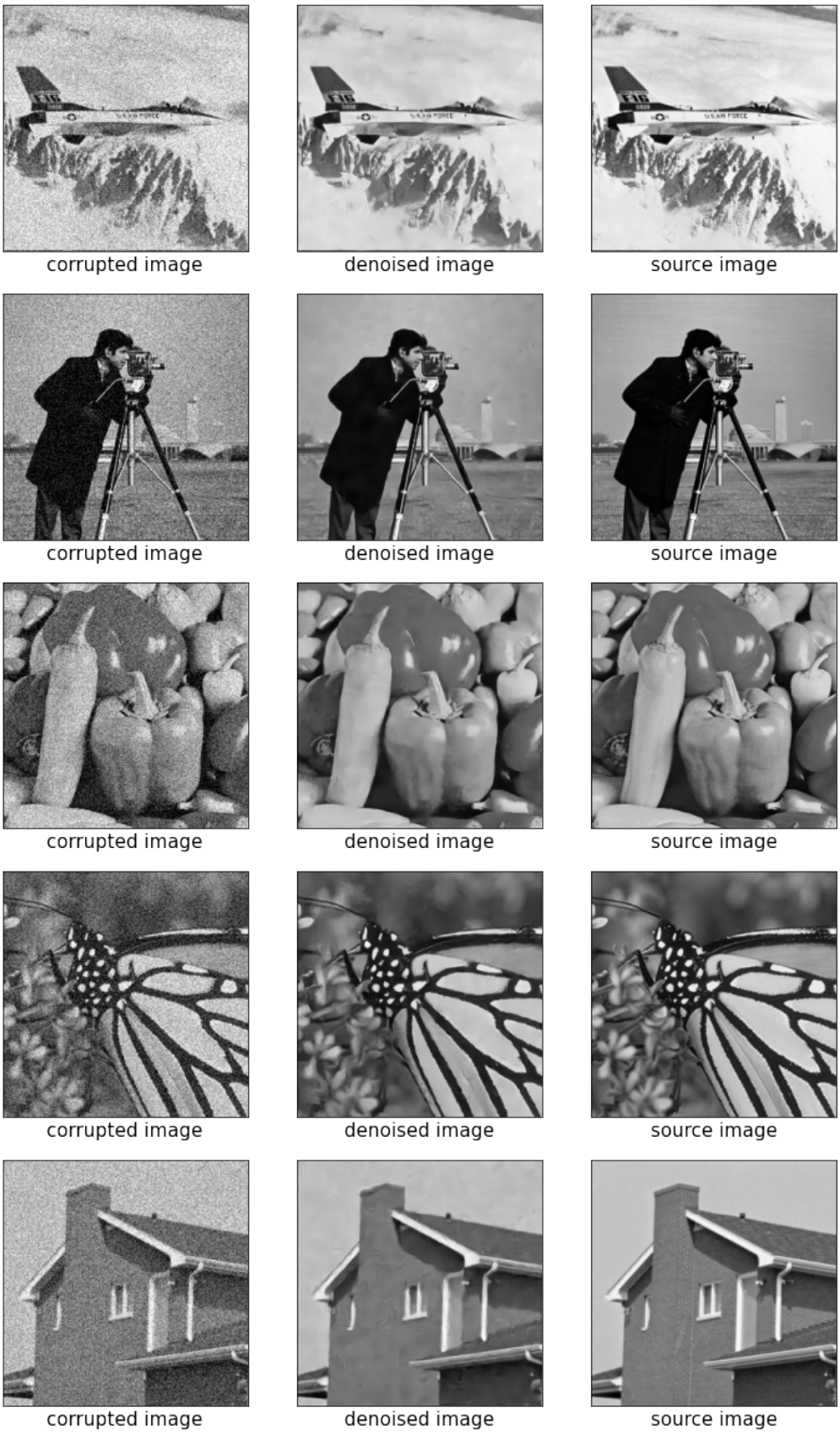

Figure 4.18: Visualization of denoising performance on Set12.

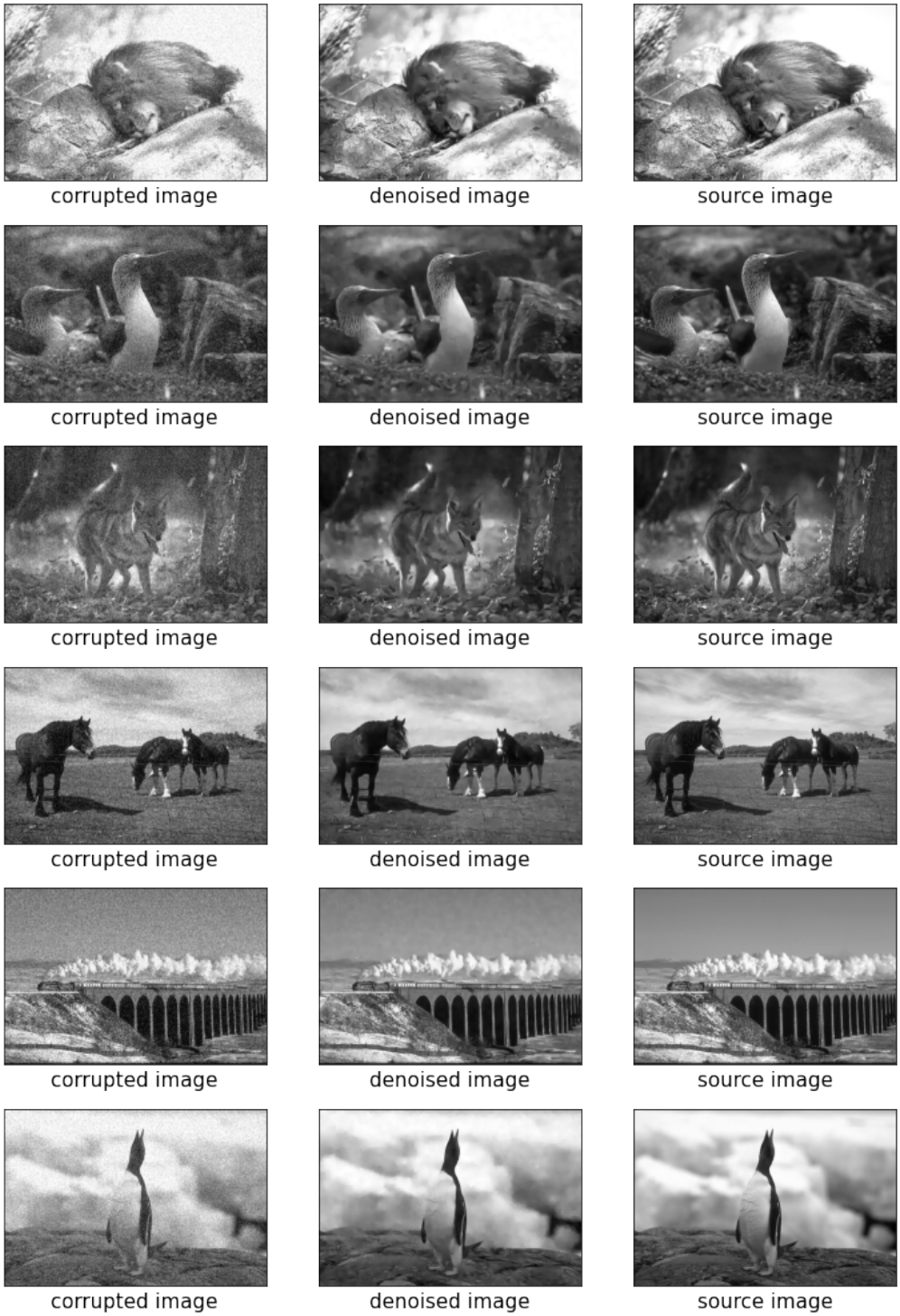

Figure 4.19: Visualization of denoising performance on BSD68.

Methods	Ours	BM3D	WNNM	EPLL	MLP	CSF	TNRD	DnCNN
C.man	29.73	29.45	29.64	29.26	29.61	29.48	29.72	30.18
House	32.20	32.85	33.22	32.17	32.56	32.39	32.53	33.06
Peppers	30.32	30.16	30.42	30.17	30.30	30.32	30.57	30.87
Starfish	28.99	28.56	29.03	28.51	28.82	28.80	29.02	29.41
Monar.	29.91	29.25	29.84	29.39	29.61	29.62	29.85	30.28
Airpl.	28.85	28.42	28.69	28.61	28.82	28.72	28.88	29.13
Parrot	29.24	28.93	29.15	28.95	29.25	28.90	29.18	29.43
Lena	31.72	32.07	32.24	31.73	32.25	31.79	32.00	32.44
Barbara	28.91	30.71	31.24	28.61	29.54	29.03	29.41	30.00
Boat	29.78	29.90	30.03	29.74	29.97	29.76	29.91	30.21
Man	29.78	29.61	29.76	29.66	29.88	29.71	29.87	30.10
Couple	29.50	29.71	29.82	29.53	29.73	29.53	29.71	30.12
Average	29.893	29.969	30.257	29.692	30.027	29.837	30.055	**30.436**

Table 4.8: PSNR results of different models on Set12 and noisy images are with noisy level 25.

Chapter 5

Conclusion and Future Work

5.1 Conclusions

Unlike the chain-structured neural networks that are manually designed for the image denoising task, in this work, we encode the search space in the cell architecture and based on differentiable architecture search methods to search the operations and connections in cell architectures.

Noticing the apparent impact of depth and width of the network to model performance, we also incorporate the search of macro-architecture in the search stage by varying the number of cells and channels when building the searched model. Grid search and successive halving search for 22 candidate models are included in this stage. We discover that these two methods lead to the same result, while successive halving search reduces the searching cost. For these models, it is not the case that deeper models consistently outperformed shallower ones. In addition, we compare the performance of models when using different proxy datasets to search, and our results show that MNIST serves as a better proxy than CIFAR100.

We also adopt a more flexible search by including depth, width, batch_size, the network's learning rate, the architecture's learning rate into the macro search space, and randomly sample 128 configurations to form the searched models. The Top5 models are selected for training, and hyperparameter optimization is performed as well. We compare the performance of models using the default hyperparameter setting in [23] with those with optimized results from BOHB. BOHB promotes searching for better configurations with more running time, although it is not guaranteed for every model. When we run BOHB with min_budget of 3, max_budget of 27, num_iteration of 15, the best model performance is slightly better than those with hyperparameter setting in [23]. Lastly, our work has achieved promising results on classical image processing datasets Set12 and BSD68.

5.2 Limitations and Future Work

For neural architecture search, the computational cost is a typical issue, especially in our case, the training set Train400 consists of images of 180 x 180, and some of the searched models are also quite large. We choose to apply different proxy datasets in the search stage and only use the training set in the training stage. However, it is hard to distinguish if the searched architecture on a proxy dataset perfectly matches the training set. We want to investigate how to use the information of training set in the train search stage and preventing unnecessary long-term search. We think a direct search on the training set can further boost the performance of our model, where we can develop strategies to reduce the computational cost.

At the same time, in BOHB optimization, there are also parameters worth tuning, for instance, min_budget, max_budget, num_iteration. Sampling more configurations will promote better configurations, and setting larger budgets will lead to more stable results. While in our experiment, the maximum number of sampled configurations for each model is 90 since it has spent 12 GPU days optimizing one model on a single GeForce GTX 1080 Ti GPU. Tuning the suitable parameters of BOHB is our next step to promote network performance.

Appendix A

Appendices

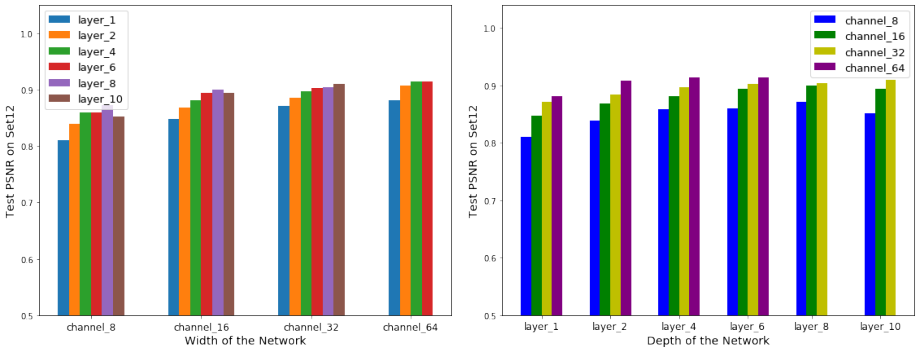

Figure A.1: Test SSIM performance on Set12. The left figure groups models in terms of the number of channels, and the right figure groups them by the number of layers. Deeper models do not always perform better than shallower ones. For example, models have eight layers and ten layers with eight channels and ten channels.

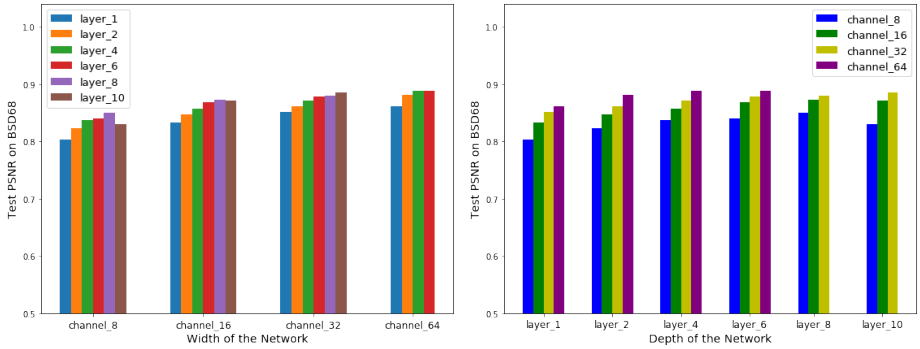

Figure A.2: Test SSIM performance on BSD68. The left figure groups models in terms of the number of channels, and the right figure groups them by the number of layers. The performance of models has the same trend as in Set12.

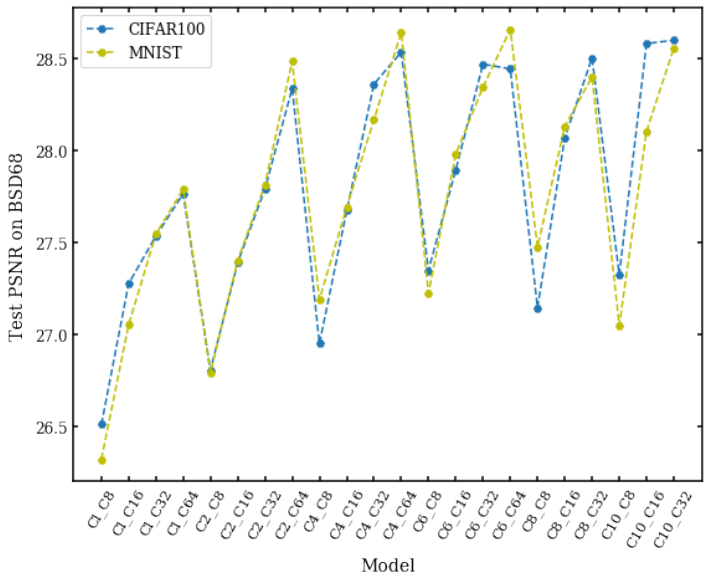

Figure A.3: The performance of models on BSD68 using CIFAR100 and MNIST as the proxy dataset. Best model *C6_C64* (model with six cells and 64 channels) is searched on MNIST, while CIFAR100 also outperformed MNIST on some candidate models.

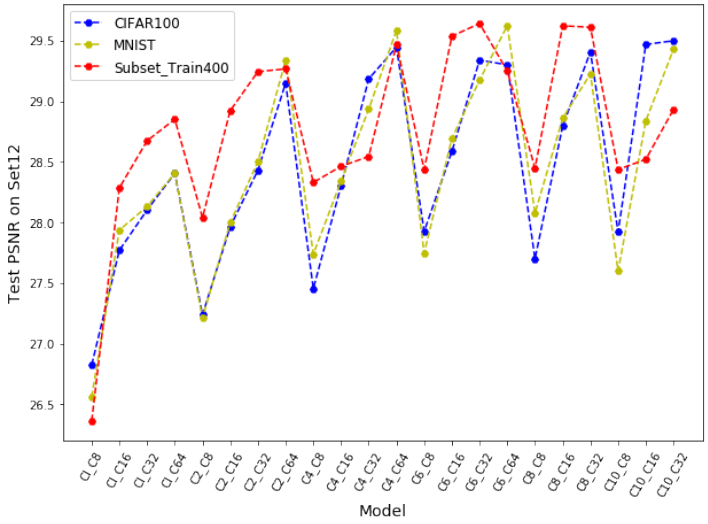

Figure A.4: The performance of models on Set12 using CIFAR100, MNIST and subset of Train400 as the proxy dataset. The subset of Train400 is the collection of 60000 random sampling patches from the training set. The best model searched on the Train400 subset has almost the same performance as that on MNIST.

Bibliography

[1] A. Buades, B. Coll, and J.-M. Morel. "A non-local algorithm for image denoising". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2005, pp. 60–65.

[2] A. Buades, B. Coll, and J.-M. Morel. "Nonlocal image and movie denoising". In: *International Journal of Computer Vision* 76.2 (2008), pp. 123–139.

[3] H. C. Burger, C. J. Schuler, and S. Harmeling. "Image denoising: Can plain neural networks compete with BM3D?" In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2012, pp. 2392–2399.

[4] H. Cai, L. Zhu, and S. Han. "Proxylessnas: Direct neural architecture search on target task and hardware". In: *arXiv preprint arXiv:1812.00332* (2018).

[5] S. Cha and T. Moon. "Fully convolutional pixel adaptive image denoiser". In: *Proceedings of the IEEE Conference on International Conference on Computer Vision*. 2019, pp. 4160–4169.

[6] L.-C. Chen, M. D. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. "Searching for efficient multi-scale architectures for dense image prediction". In: *arXiv preprint arXiv:1809.04184* (2018).

[7] X. Chen, L. Xie, J. Wu, and Q. Tian. "Progressive differentiable architecture search: Bridging the depth gap between search and evaluation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1294–1303.

[8] Y. Chen and T. Pock. "Trainable nonlinear reaction diffusion: A flexible framework for fast and effective image restoration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (2016), pp. 1256–1272.

[9] Y. Chen and T. Pock. "Trainable nonlinear reaction diffusion: A flexible framework for fast and effective image restoration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39.6 (2016), pp. 1256–1272.

[10] K. Dabov, A. Foi, V. Katkovnik, and K. Egiazarian. "Image denoising by sparse 3-D transform-domain collaborative filtering". In: *IEEE Transactions on Image Processing* 16.8 (2007), pp. 2080–2095.

[11] W. Dong, L. Zhang, G. Shi, and X. Li. "Nonlocally centralized sparse representation for image restoration". In: *IEEE Transactions on Image Processing* 22.4 (2012), pp. 1620–1630.

[12] M. Elad and M. Aharon. "Image denoising via sparse and redundant representations over learned dictionaries". In: *IEEE Transactions on Image Processing* 15.12 (2006), pp. 3736–3745.

[13] T. Elsken, J. H. Metzen, F. Hutter, et al. "Neural architecture search: A survey." In: *J. Mach. Learn. Res.* 20.55 (2019), pp. 1–21.

[14] S. Falkner, A. Klein, and F. Hutter. "BOHB: Robust and efficient hyperparameter optimization at scale". In: *Proceedings of the IEEE Conference on International Conference on Machine Learning*. 2018, pp. 1437–1446.

[15] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Artificial Intelligence and Statistics*. 2010, pp. 249–256.

[16] K. Jamieson and A. Talwalkar. "Non-stochastic best arm identification and hyperparameter optimization". In: *Artificial Intelligence and Statistics*. 2016, pp. 240–248.

[17] X. Lan, S. Roth, D. Huttenlocher, and M. J. Black. "Efficient belief propagation with learned higher-order markov random fields". In: *Proceedings of the European Conference on Computer Vision*. 2006, pp. 269–282.

[18] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. "Hyperband: A novel bandit-based approach to hyperparameter optimization". In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 6765–6816.

[19] S. Z. Li. *Markov random field modeling in image analysis*. 2009.

[20] C. Liu, L.-C. Chen, F. Schroff, H. Adam, W. Hua, A. L. Yuille, and L. Fei-Fei. "Auto-deeplab: Hierarchical neural architecture search for semantic image segmentation". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 82–92.

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

[22] D. Liu, B. Wen, Y. Fan, C. C. Loy, and T. S. Huang. "Non-local recurrent network for image restoration". In: *arXiv preprint arXiv:1806.02919* (2018).

[23] H. Liu, K. Simonyan, and Y. Yang. "Darts: Differentiable architecture search". In: *arXiv preprint arXiv:1806.09055* (2018).

[24] J. Mairal, F. Bach, J. Ponce, G. Sapiro, and A. Zisserman. "Non-local sparse models for image restoration". In: *Proceedings of the IEEE Conference on International Conference on Computer Vision*. 2009, pp. 2272–2279.

[25] T. Plötz and S. Roth. "Neural nearest neighbors networks". In: *Advances in neural Information Processing Systems* 31 (2018), pp. 1087–1098.

[26] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. "Regularized evolution for image classifier architecture search". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 33. 01. 2019, pp. 4780–4789.

[27] U. Schmidt and S. Roth. "Shrinkage fields for effective image restoration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 2774–2781.

[28] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[29] D. Ulyanov, A. Vedaldi, and V. Lempitsky. "Deep image prior". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9446–9454.

[30] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. "Image quality assessment: from error visibility to structural similarity". In: *IEEE Transactions on Image Processing* 13.4 (2004), pp. 600–612.

[31] Y. Xu, L. Xie, X. Zhang, X. Chen, G.-J. Qi, Q. Tian, and H. Xiong. "PC-DARTS: Partial channel connections for memory-efficient architecture search". In: *arXiv preprint arXiv:1907.05737* (2019).

[32] H. Zhang, Y. Li, H. Chen, and C. Shen. "Memory-efficient hierarchical neural architecture search for image denoising". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2020, pp. 3657–3666.

[33] K. Zhang, W. Zuo, Y. Chen, D. Meng, and L. Zhang. "Beyond a gaussian denoiser: Residual learning of deep cnn for image denoising". In: *IEEE Transactions on Image Processing* 26.7 (2017), pp. 3142–3155.

[34] K. Zhang, W. Zuo, and L. Zhang. "FFDNet: Toward a fast and flexible solution for CNN-based image denoising". In: *IEEE Transactions on Image Processing* 27.9 (2018), pp. 4608–4622.

[35] Y. Zhang, Y. Tian, Y. Kong, B. Zhong, and Y. Fu. "Residual dense network for image restoration". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 43.7 (2020), pp. 2480–2495.

[36] Y. Zhao, Z. Jiang, A. Men, and G. Ju. "Pyramid real image denoising network". In: *IEEE Visual Communications and Image Processing*. 2019, pp. 1–4.

[37] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. "Learning transferable architectures for scalable image recognition". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 8697–8710.

[38] D. Zoran and Y. Weiss. "From learning models of natural image patches to whole image restoration". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2011, pp. 479–486.