

# BlockQNN: Efficient Block-wise Neural Network Architecture Generation

Zhao Zhong, Zichen Yang, Boyang Deng, Junjie Yan, Wei Wu, Jing Shao, and Cheng-Lin Liu, *Fellow, IEEE*

**Abstract**—Convolutional neural networks have gained a remarkable success in computer vision. However, most usable network architectures are hand-crafted and usually require expertise and elaborate design. In this paper, we provide a block-wise network generation pipeline called BlockQNN which automatically builds high-performance networks using the Q-Learning paradigm with epsilon-greedy exploration strategy. The optimal network block is constructed by the learning agent which is trained to choose component layers sequentially. We stack the block to construct the whole auto-generated network. To accelerate the generation process, we also propose a distributed asynchronous framework and an early stop strategy. The block-wise generation brings unique advantages: (1) it yields state-of-the-art results in comparison to the hand-crafted networks on image classification, particularly, the best network generated by BlockQNN achieves 2.35% top-1 error rate on CIFAR-10. (2) it offers tremendous reduction of the search space in designing networks, spending only 3 days with 32 GPUs. A faster version can yield a comparable result with only 1 GPU in 20 hours. (3) it has strong generalizability in that the network built on CIFAR also performs well on the larger-scale dataset. The best network achieves very competitive accuracy of 82.0% top-1 and 96.0% top-5 on ImageNet.

**Index Terms**—Convolutional Neural Network, Auto-Generated Network, Reinforcement Learning, Q-Learning.

## 1 INTRODUCTION

DURING the last decades, Convolutional Neural Networks (CNNs) have shown remarkable potentials almost in every field in the computer vision society [1]. It achieved successes first in image classification [2], and then in object detection [3], [4], semantic segmentation [5], [6] and tracking [7], [8]. For example, the network evolution from AlexNet [2], VGG [9], Inception [10] to ResNet [11] has improved the top-5 performance on ImageNet challenge steadily from 83.6% to 96.43%. However, as the performance gain usually requires an increasing network capacity, a high-performance network architecture generally possesses a tremendous number of possible configurations about the number of layers, hyperparameters in each layer and type of each layer. It is hence infeasible to find the optimal network structure by manually exhaustive search, and the design of successful hand-crafted networks heavily rely on expert knowledge and experience. Therefore, constructing network in a smart and automatic manner remains an open problem.

Although some recent works have attempted computer-aided or automated network design [12], [13], there are several challenges unsolved: (1) The large number of convolutional layers and the numerous options in type and hyperparameters of each make huge search space and heavy computational costs for network generation. (2) The network designed on a specific dataset or task yields inferior performance when transferred to other datasets or tasks. In this paper, we aim to solve the aforementioned

challenges by proposing a novel fast Q-learning framework, called *BlockQNN*, for automatically designing the network architecture, as shown in Fig. 1.

Particularly, to make the network generation efficient and generalizable, we generate the network in block-wise, *i.e.*, stacking personalized blocks rather than tedious per-layer network piling. This is inspired by some modern CNN architectures such as Inception [10], [14], [15] and ResNet Series [11], [16] which are assembled as the stack of basic block structures. For example, the inception and residual blocks shown in Fig. 1 are repeatedly concatenated to construct the entire network. With such kind of block-wise network architecture, the generated network owns a powerful generalization to other task domains or different datasets.

In comparison to previous methods like NAS [13] and MetaQNN [12], as depicted in Fig. 1, we present a more readily and elegant model generation method that specifically designed for block-wise generation. Motivated by the unsupervised reinforcement learning paradigm, we employ the well-known Q-learning [17] with experience replay [18] and epsilon-greedy strategy [19] to effectively and efficiently search for the optimal block structure. The network block is constructed by the learning agent which is trained sequentially to choose component layers. Afterwards we stack the block to construct the whole auto-generated network. Moreover, we propose an early stop strategy to enable efficient search with fast convergence. A novel reward function is designed to ensure the accuracy of the early stopped network to be positively correlated with the converged network. Good blocks can be selected in reduced training time using this property. With this acceleration strategy, we can construct a Q-learning agent to learn the optimal block-wise network structure for a given task with limited resources (*e.g.* few GPUs or short time period). The generated architectures are thus succinct and have powerful generalization ability compared to the networks generated by the other automatic network generation methods.

A preliminary version of this manuscript was published previ-

- Z. Zhong is with the NLPR, Institute of Automation of Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing 100190, P.R. China. E-mail: zhao.zhong@nlpr.ia.ac.cn.
- Z. Yang, B. Deng, J. Yan, W. Wu and J. Shao are with SenseTime Research Institute. E-mail: {yangzichen, dengboyang, yanjunjie, wuwei, shaojing}@sensetime.com
- C.-L. Liu is with the NLPR, Institute of Automation of Chinese Academy of Sciences, Beijing, China, and the CAS Center for Excellence in Brain Science and Intelligence Technology, University of Chinese Academy of Sciences, Beijing 100190, P.R. China. E-mail: liucl@nlpr.ia.ac.cn.

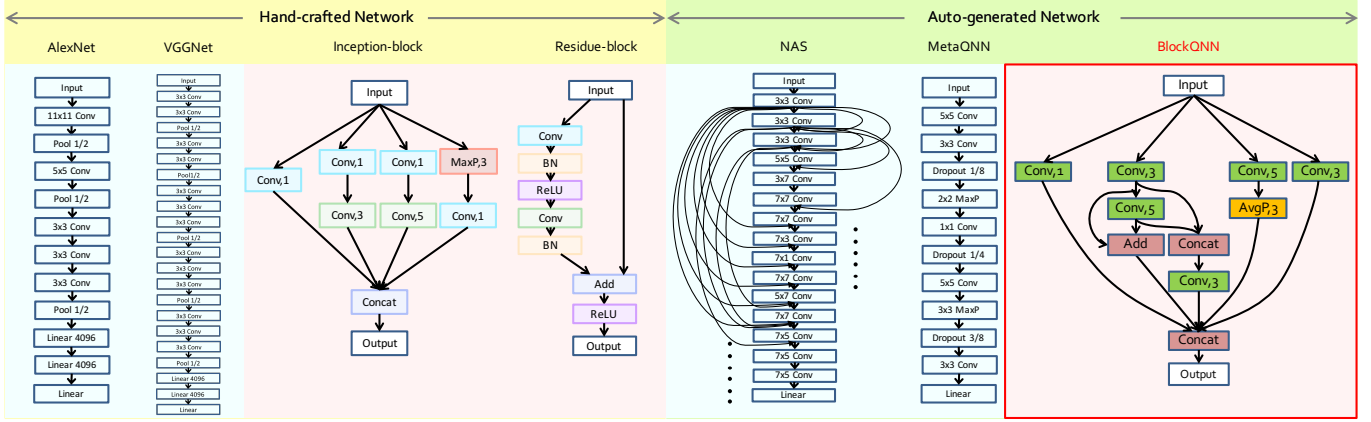


Figure 1. The proposed **BlockQNN** (right in red box) compared with the hand-crafted networks marked in yellow and the existing auto-generated networks in green. Automatically generating the plain networks [12], [13] marked in blue need large computational costs in searching optimal layer types and hyperparameters for each single layer, while the block-wise network heavily reduces the cost to search structures only for one block. The entire network is then constructed by stacking the generated blocks. Similar block concept has been demonstrated its superiority in hand-crafted networks, such as inception-block and residue-block marked in red.

ously in conference [20]. Since then, the block conception for auto-generated network have been adopted and generalized to other methods, such as [21], [22], [23], [24], [25]. Searching transferable blocks (referred to as cells in [26]) and assembling them into a network becomes an universal formulation in automating neural network design area.

After that, we introduce more advanced depthwise convolution operation [27] to update the state-of-the-art performance on image classification, and in this paper we analyze the different connection styles between blocks instead of stacking block structures sequentially. Also, we propose the accelerated block-wise network generation with network performance prediction, called faster BlockQNN, which only costs 20 hours with 1 GPU on CIFAR.

The proposed block-wise network generation method brings a few advantages as follows:

- *Effectiveness.* The automatically generated networks present state-of-the-art performance compare to those of hand-crafted networks with human expertise. The proposed method is also superior to the existing automatic works and achieves a leading performance on CIFAR-10 with 2.35% error rate.
- *Efficiency.* We are the first to consider block-wise setup in automatic network generation. The block-wise setup and the proposed early stop strategy result in a fast search process. The network generation for CIFAR task reaches convergence with only 32 GPUs in 3 days, which is much more efficient than that by NAS [13] with 800 GPUs in 28 days. Moreover, the faster version can get a comparable result with only 1 GPU in 20 hours, which is affordable for common deep learning researchers.
- *Transferability.* The proposed method offers surprisingly superior transferable ability that the network generated for CIFAR can be transferred to ImageNet with outstanding performance with little modification. The best network achieves very competitive accuracy of 82.0% top-1 and 96.0% top-5 on ImageNet.

## 2 RELATED WORK

Early works, from 1980s, have made efforts on automating neural network design which often searched good architecture by the

genetic algorithm or other evolutionary algorithms [28], [29], [30], [31], [32], [33], [34]. Nevertheless, these works, to our best knowledge, cannot perform competitively compared with hand-crafted networks. Recent works, *i.e.* Neural Architecture Search (NAS) [13] and MetaQNN [12], adopted reinforcement learning to automatically search a good network architecture. Although they can yield good performance on small datasets such as CIFAR-10, CIFAR-100, the direct use of MetaQNN or NAS for architecture design on big datasets like ImageNet [35] is computationally expensive via searching in a huge space. Besides, the network generated by this kind of methods is task-specific or dataset-specific, that is, it cannot be well transferred to other tasks nor datasets with different input data sizes. For example, the network designed for CIFAR-10 cannot be generalized to ImageNet.

Instead, our approach is aimed for designing network block architecture by an efficient search method with a distributed asynchronous Q-learning framework as well as an early-stop strategy. The block design conception follows the modern convolutional neural networks such as Inception [10], [14], [15] and Resnet [11], [16]. The inception-based networks construct the inception blocks via a hand-crafted multi-level feature extractor strategy by computing  $1 \times 1$ ,  $3 \times 3$ , and  $5 \times 5$  convolutions, while the Resnet uses residue blocks with shortcut connection to make it easier to represent the identity mapping which allows a very deep network. The blocks automatically generated by our approach have similar structures such as some blocks contain short cut connections and inception-like multi-branch combination. We will discuss the details in Section 5.1. Concurrent with our work, the NASNet [26] is developed for learning block structures (referred to as cells in [26]) to construct the whole network with RNN-based controller.

There is a growing interest in improving the efficiency of automatic network generation for Common researchers who have limited computing resources. Baker et al. [36] use standard frequentist regression models to predict the final performance, Brock et al. [37] propose SMASH which designs an architecture and then uses a hyper-network to generate its weights. These methods, however, can not compete with state-of-the-art networks, and the networks generated by them are still task-specific or dataset-specific.

Table 1

Network Structure Code Space. The space contains seven types of commonly used layers. Layer index stands for the position of the current layer in a block, the range of the parameters is set to be  $T = \{1, 2, 3, \dots, \max \text{ layer index}\}$ . Three kinds of kernel sizes are considered for convolution layer and two sizes for pooling layer. Pred1 and Pred2 refer to the predecessor parameters which is used to represent the index of preceding layers, the allowed range is  $K = \{1, 2, \dots, \text{current layer index} - 1\}$

Name	Index	Type	Kernel Size	Pred1	Pred2
Convolution	T	1	1, 3, 5	K	0
Max Pooling	T	2	1, 3	K	0
Average Pooling	T	3	1, 3	K	0
Identity	T	4	0	K	0
Elemental Add	T	5	0	K	K
Concat	T	6	0	K	K
Terminal	T	7	0	0	0

Other related works include hyper-parameter optimization [38], meta-learning [39] and learning to learn methods [40], [41]. The goal of these works is to use meta-data to improve the performance of the existing algorithms, such as finding the optimal learning rate of optimization methods or the optimal number of hidden layers to construct the network. In this paper, we focus on learning the entire topological structure of network blocks to improve the performance.

### 3 METHODOLOGY

In this section, we first present the basic designs and properties of the proposed BlockQNN framework. Extension of the framework to block connection style will be described in Section 3.4. The faster version with network performance prediction called faster BlockQNN will be introduced in Section 3.5.

#### 3.1 Convolutional Neural Network Blocks

The modern CNN architectures, *e.g.* Inception and Resnet, are designed by stacking several `blocks` each of which shares similar structure but with different weights and filter numbers to construct the network. With the block-wise design, the network can not only achieves high performance but also generalizes well to different datasets and tasks. Unlike previous research on automating neural network design which generates the entire network directly, we aim at designing the `block` structure.

As a CNN contains a feed-forward computation procedure, we represent it by a directed acyclic graph (DAG), where each node corresponds to a layer in the CNN while directed edges stand for data flow from one layer to another. To turn such a graph into a uniform representation, we propose a novel layer representation scheme called **Network Structure Code** (NSC), as shown in Table 1. Each block is then depicted by a set of 5-D NSC vectors. In NSC, the first three numbers stand for the *layer index*, *operation type* and *kernel size* respectively. The last two are *predecessor parameters* which refer to the position of a layer’s predecessor layer in structure codes. The second predecessor (Pred2) is set for the layer owns two predecessors, and for the layer with only one predecessor, Pred2 will be set to zero. This design is motivated by the current powerful hand-crafted networks like Inception and Resnet which own their special block structures. This kind of block structure shares similar properties such as containing more

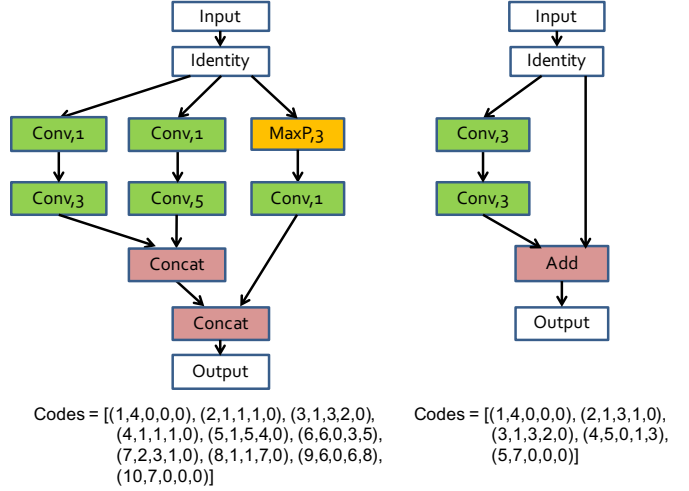


Figure 2. Representative block exemplars with their Network structure codes (NSC) respectively: the block with multi-branch connections (left) and the block with shortcut connections (right).

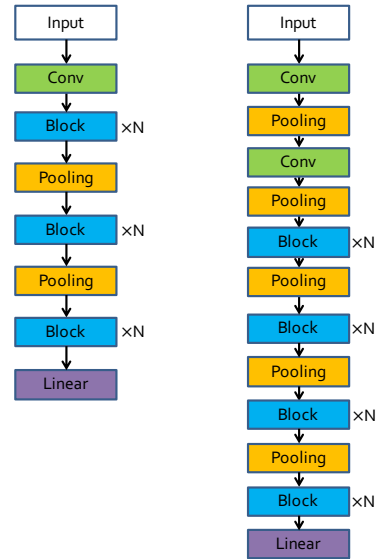


Figure 3. Auto-generated networks on CIFAR-10 (left) and ImageNet (right). Each network starts with a few convolution layers to learn low-level features, and followed by multiple repeated `blocks` with several pooling layers inserted for downsampling.

complex connections, *e.g.* shortcut connections or multi-branch connections, unlike the simple connections in plain networks such as AlexNet. Thus, the proposed NSC can encode complexity architectures as shown in Fig. 2. In addition, all layers without successor in the block are concatenated together to provide the final output. Note that each convolution operation, same as the declaration in Resnet [16], refers to a **Pre-activation Convolutional Cell** (PCC) with three components, *i.e.* *ReLU*, *Convolution* and *Batch Normalization*. This results in a smaller search space than that with three components separately searchable, and hence with the PCC, we can get better initialization for searching and generating optimal block structure with a quick training process.

Based on the above defined `blocks`, we construct the complete network by stacking these block structures sequentially which turn a common plain network into its counterpart block

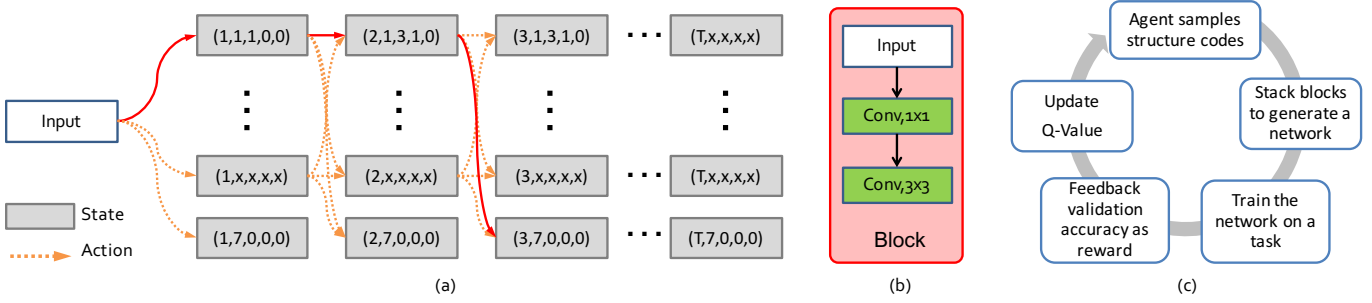


Figure 4. Q-learning process illustration. (a) The state transition process by different action choices. The block structure in (b) is generated by the red solid line in (a). (c) The flow chart of the Q-learning procedure.

version. Two representative auto-generated networks on CIFAR and ImageNet tasks are shown in Fig. 3. There is no down-sampling operation within each block. We perform down-sampling directly by the pooling layer. If the size of feature map is halved by pooling operation, the block’s weights will be doubled. The architecture for ImageNet contains more pooling layers than that for CIFAR because of their different input sizes, *i.e.*  $224 \times 224$  for ImageNet and  $32 \times 32$  for CIFAR. More importantly, the blocks can be repeated for arbitrary  $N$  times to fulfill different demands, and can even be placed in other manner, such as inserting the block into the Network-in-Network [42] framework or setting short cut connection between different blocks. We will discuss the block connection later in the Section 3.4.

### 3.2 Designing Network Blocks With Q-Learning

Albeit we squeeze the search space of the entire network design by focusing on constructing network blocks, there is still a large amount of possible structures to seek. Therefore, we employ reinforcement learning rather than random sampling for automatic design. Our method is based on standard tabular Q-learning, a kind of reinforcement learning, which concerns how an agent ought to take actions so as to maximize the cumulative reward. The Q-learning model consists of an *agent*, *states* and a set of *actions*.

In this paper, the *state*  $s \in S$  represents the status of the current layer which is defined as a Network Structure Code (NSC) claimed in Section 3.1, *i.e.* 5-D vector {layer index, layer type, kernel size, pred1, pred2}. The *action*  $a \in A$  is the decision for the next successive layer. Thanks to the defined NSC set with a limited number of choices, both the *state* and *action* space are thus finite and discrete to ensure a relatively small search space. The state transition process  $(s_t, a(s_t)) \rightarrow (s_{t+1})$  is shown in Fig. 4(a), where  $t$  refers to the current layer. The block example in Fig. 4(b) is generated by the red solid lines in Fig. 4(a). The learning agent is given the task of sequentially picking NSC of a block. The structure of block can be considered as an action selection trajectory  $\tau_{a_{1:T}}$ , *i.e.* a sequence of NSCs. We model the layer selection process as a Markov Decision Process with the assumption that a well-performing layer in one block should also perform well in another block [12]. To find the optimal architecture, the agent maximizes its expected *reward* over all possible trajectories, denoted by  $R_\tau$ ,

$$R_\tau = \mathbb{E}_{P(\tau_{a_{1:T}})}[\mathbb{R}], \quad (1)$$

where the  $\mathbb{R}$  is the cumulative reward. The expected reward can be maximized using the recursive Bellman Equation. Given a state

$s_t \in S$  and subsequent action  $a \in A(s_t)$ , we define the maximum total expected reward to be  $Q^*(s_t, a)$  which is known as Q-value of state-action pair. The recursive Bellman Equation is then written as

$$Q^*(s_t, a) = \mathbb{E}_{s_{t+1}|s_t, a}[\mathbb{E}_{r|s_t, a, s_{t+1}}[r|s_t, a, s_{t+1}] + \gamma \max_{a' \in A(s_{t+1})} Q^*(s_{t+1}, a')]. \quad (2)$$

Empirically, the above quantity can be formulated as an iterative update:

$$Q(s_T, a) = 0, \quad (3)$$

$$Q(s_{T-1}, a_T) = (1 - \alpha)Q(s_{T-1}, a_T) + \alpha r_T, \quad (4)$$

$$Q(s_t, a) = (1 - \alpha)Q(s_t, a) + \alpha[r_t + \gamma \max_{a'} Q(s_{t+1}, a')], t \in \{1, 2, \dots, T-2\}, \quad (5)$$

where  $\alpha$  is the learning rate which determines how the newly acquired information overrides the old information,  $\gamma$  is the discount factor which measures the importance of future rewards;  $r_t$  denotes the intermediate reward observed for the current state  $s_t$ , and  $s_T$  refers to final state, *i.e.* terminal layers;  $r_T$  is the validation accuracy of corresponding network trained convergence on training set for  $a_T$ , *i.e.* action to final state. Since the reward  $r_t$  cannot be explicitly measured in our task, we use reward shaping [43] to speed up training. The shaped intermediate reward is defined as:

$$r_t = \frac{r_T}{T}. \quad (6)$$

Previous works [12] ignore these rewards in the iterative process by simply setting them to zero, which may cause a slow convergence in the beginning. This is known as the temporal credit assignment problem which makes RL time consuming [44]. In this case, the Q-value of  $s_T$  is much higher than others in early stage of training and thus leads the agent prefer to stop searching at the very beginning, *i.e.* tend to build small block with fewer layers. As the comparison result in Fig. 5 shows, the learning process of the agent with our shaped reward  $r_t$  is convergent much faster than previous method.

We summarize the learning procedure in Fig. 4(c). The agent first samples a set of structure codes to build the block architecture, based on which the entire network is constructed by stacking these blocks sequentially. We then train the generated network on a certain task, and the validation accuracy is regarded as the reward to update the Q-value. Afterwards, the agent picks another set of structure codes to get a better block structure.



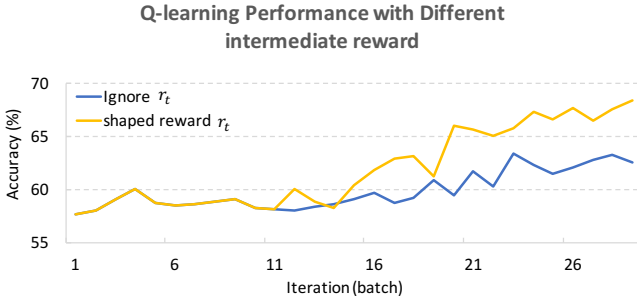


Figure 5. Comparison results of Q-learning with and without the shaped intermediate reward  $r_t$ . By taking our shaped reward, the learning process convergent faster than that without shaped reward start from the same exploration.

### 3.3 Early Stop Strategy

Introducing block-wise generation indeed increases the efficiency. However, it is still time consuming to complete the search process. To further accelerate the learning process, we introduce an early stop strategy. However, early stopping training process may result in a poor accuracy. Fig. 6 shows an example, where the early-stop accuracy in yellow line is much lower than the final accuracy in orange line, which means that some good blocks unfortunately perform worse than bad blocks when stop training early. Meanwhile, we notice that the FLOPs and density of the corresponding blocks have a negative correlation with the final accuracy. Thus, we redefine the reward function as

$$\begin{aligned} \text{reward} = & \text{ACC}_{\text{EarlyStop}} - \mu \log(\text{FLOPs}) \\ & - \rho \log(\text{Density}), \end{aligned} \quad (7)$$

where FLOPs [45] refer to an estimation of computational complexity of the block, and Density is the edge number divided by the dot number in DAG of the block. There are two hyperparameters,  $\mu$  and  $\rho$ , to balance the weights of FLOPs and Density. With the redefined reward function, the reward is more relevant to the final accuracy.

With this early stop strategy and small search space of network blocks, the training process just costs 3 days to complete the searching process with only 32 GPUs, which is superior to that of [13], which spends 28 days with 800 GPUs to achieve the similar performance. However, the use of 32 GPUs is still not common for most deep learning practitioners. Accordingly, we will further accelerate the searching process using an algorithm called faster BlockQNN described in Section 3.5.

### 3.4 Connection Style Between Blocks

Our method can search for the optimal block structure effectively and efficiently, but still utilizes manual rules, *i.e.* stacking the block structures sequentially and when the size of feature map is halved by pooling operation, the block’s weights will be doubled or increased. With this block connection rule, we can transfer the block structure between different datasets and tasks easily.

However, stacking the block structures sequentially may not be the optimal connection style between blocks. Hence, we trade off some transferable ability and design the connection between specific block structures automatically by search. The only difference from the above method lies in the definition of Network Structure Code: *convolution layer* is replaced by *blocks structure*

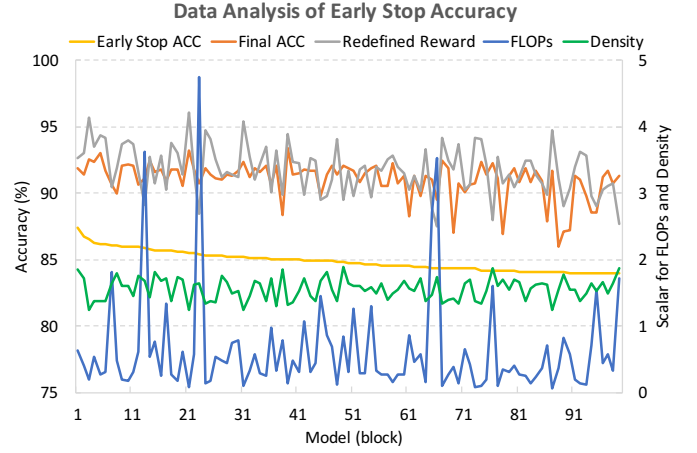


Figure 6. The performance of early stop training is poorer than the final accuracy of a complete training. With the help of FLOPs and Density, it squeezes the gap between the redefined reward function and the final accuracy.

and the *kernel size* for convolutional operation is substituted by *channel numbers*. Firstly, the block structures are connected sequentially and we use the *predecessor parameters* to represent the additional connection between different blocks. We perform down-sampling only by the pooling layer with stride 2 and use  $1 \times 1$  convolutions to match the different dimensions for connected layers.

With the block connection auto-generation module, the Block-QNN can be regarded as a two-stage framework: (1) find the optimal block, and (2) find the optimal connection for optimal block. We can further improve the performance of auto-generated network with this two-stage strategy, it proves stacking the block structures sequentially is not the best choice. But the generated network is dataset-specific that cannot be well transferred to other tasks with different input data size. The exploration of universal block connection formulation with transferable ability is still an open problem.

### 3.5 Predicting Network Performance Before Training

To further accelerate the block searching process, for the common deep learning practitioners who have limited computing resources, we propose a strategy to predict the network performance before training. As we know, the most time-consuming portion in network generation is the training of the sampled network to get the validation accuracy as reward. To mitigate this cost, we assess a network architecture quantitatively before investing resources in training it. The design method with network performance prediction is called as Faster BlockQNN, which is depicted in Figure 7.

The network performance prediction model can be formalized as a function, denoted by  $f$ . The function  $f$  takes two arguments, a network architecture  $x$  and an epoch index  $t$ , and produces a scalar value  $f(x, t)$  as the prediction of the accuracy at the end of the  $t$ -th epoch. Here, incorporating the epoch index  $t$  as an input to  $f$  is reasonable, as the validation accuracy generally changes as the training proceeds. Therefore, when we predict performance, we have to be specific about the time point of the prediction.

Note that this formulation differs fundamentally from previous works [33], [36], [46], which require the observation of the

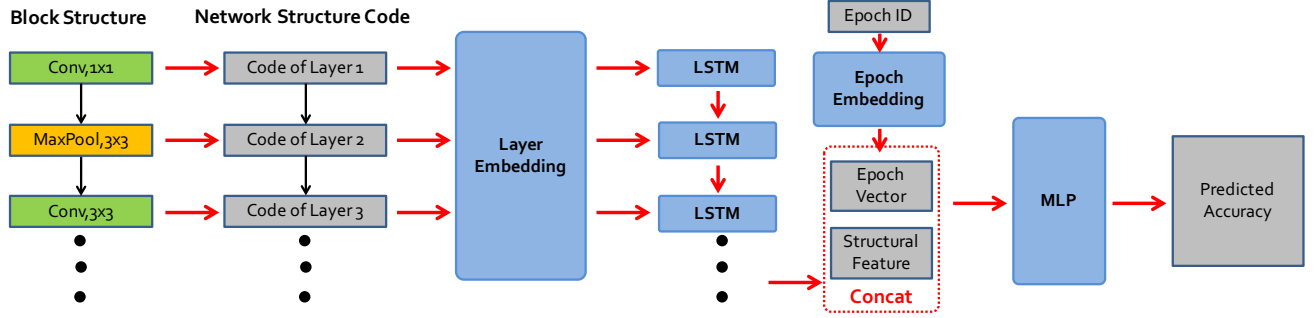


Figure 7. The overall pipeline of the *Faster BlockQNN* framework. Given a network architecture, it first encodes each layer into a vector through integer coding and layer embedding. Subsequently, it applies a recurrent network with LSTM units to integrate the information of individual layers following the network topology into a *structural feature*. This structural feature together with the epoch index (also embedded into a vector) will finally be fed to an MLP to predict the accuracy at the corresponding time point, *i.e.* the end of the given epoch. Note that the blocks indicated by blue color, including the embeddings, the LSTM, and the MLP, are jointly learned in an end-to-end manner.

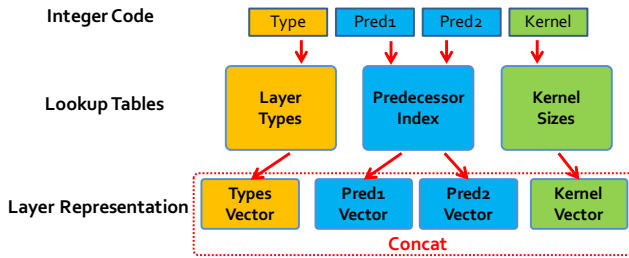


Figure 8. The layer embedding component. It takes the integer codes as input, maps them to embedded vectors respectively via table lookup, and finally concatenates them into a real vector representation. Note that *Pred1* and *Pred2* share the same lookup table.

initial part (usually 25%) of the training curve and extrapolate the remaining part. On the contrary, our method aims to predict the entire curve, relying only on the network architecture. In this way, it can provide feedback much quicker and thus is particularly suited for large-scale search of network designs.

As stated in Section 3.1, we define the various layers in network as Network Structure Code (NSC), *i.e.* 5-D vector {layer index, layer type, kernel size, pred1, pred2}. While capturing the key information for a layer, this *discrete* representation is not amenable to complex numerical computation and deep pattern recognition. Inspired by word embedding [47], a strategy proven to be very effective in natural language processing, we take one step further and develop *Layer Embedding*, a scheme to turn the integer codes into a unified real-vector representation.

As shown in Figure 8, the embedding is done by table lookup. Specifically, this module is associated with three lookup tables, respectively for *layer types*, *kernel sizes*, and *predecessor*. Note that the predecessor table is used to encode both *Pre1* and *Pre2*. Given a tuple of integers, we can convert its element into a real vector by retrieving from the corresponding lookup table. Then by concatenating all the embedded vectors derived respectively from individual integers, we can form a vector representation of the layer.

With the layer-wise representations based on Network Structure Code (NSC) and Layer Embedding, the next is to aggregate

them into an overall representation for the entire network. Inspired by the success of recurrent networks in sequential modeling, *e.g.* in language modeling [48] and video analytics [49], we choose to explore recurrent networks in our problem. Specifically, we adopt the *Long-Short Term Memory (LSTM)* [50], an effective variant of RNN, for integrating the information along a sequence of layers. Along the way from low-level to high-level layers, the LSTM network would gradually incorporate layer-wise information into the hidden state. At the last step, *i.e.* the layer right before the fully connected layer for classification, we extract the hidden state of the LSTM cell to represent the overall structure of the network, which we refer to as the *structural feature*. Note that we would reverse the topological sequence of architectures before feeding it to LSTM since we think Long-term dependencies are better recognized by LSTM if the signal occurs at the starting point.

As shown in Figure 7, the *Faster BlockQNN* framework will finally combine this structural feature with the epoch index (also embedded into a real-vector) and use a Multi-Layer Perceptron (MLP) to make the final prediction of accuracy. In particular, the MLP component at the final step is comprised of three fully connected layers with Batch Normalization and ReLU activation. The output of this component is a real value that serves as an estimate of the accuracy.

Given a set of sample networks  $\{x_i\}_{1:N}$ , we can obtain a performance curves  $y_i(t)$  for each network  $x_i$ , *i.e.* the validation accuracy as a function of epoch numbers, by training the network on a given dataset. Hence, we can obtain a set of pairs  $\mathcal{D} = \{(x_i, y_i)\}_{1:N}$  and learn the parameters of the predictor in a supervised way.

Specifically, we formulate the learning objective with the *smooth L1* loss, denoted by  $l$ , as below:

$$\mathcal{L}(\mathcal{D}; \theta) = \frac{1}{N} \sum_{i=1}^n l(f(x_i, T), y_i(T)). \quad (8)$$

Here,  $\theta$  denotes the predictor parameters. Note that we train each sample network with  $T$  epochs, and use the results of the final epoch to supervise the learning process. Our framework is very flexible – with the entire learning curves, in principle, one can use the results at multiple epochs for training. However, we found empirically that using only the final epochs already yields reasonably good results.

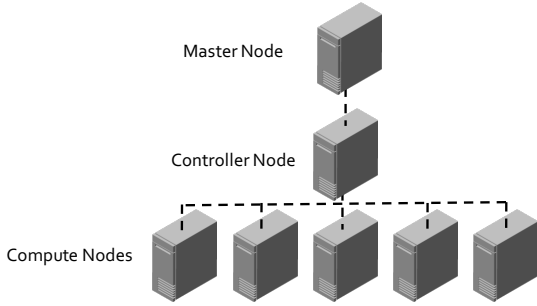


Figure 9. The distributed asynchronous framework. It contains three parts: master node, controller node and compute nodes.

Table 2  
Epsilon Schedules. The number of iteration the agent trains at each epsilon( $\epsilon$ ) state.

$\epsilon$	1.0	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
Iters	95	7	7	7	10	10	10	10	10	12

With the Faster BlockQNN, the demand of computing resource will be further reduced. More accurately, we can get a comparable result with only 1 GPU in 20 hours which is afforded for common deep learning researcher.

## 4 TRAINING DETAILS

**Distributed Asynchronous Framework.** To speed up the learning of agent, we use a distributed asynchronous framework as illustrated in Fig. 9. It consists of three parts: master node, controller node and compute nodes. The agent first samples a batch of block structures in master node. Afterwards, we store them in a controller node which uses the block structures to build the entire networks and allocates these networks to compute nodes. It can be regarded as a simplified parameter-server [51], [52]. Specifically, the network is trained in parallel on each of compute nodes and returns the validation accuracy as reward by controller nodes to update agent. With this, we can generate network efficiently on multiple machines with multiple GPUs.

**Epsilon-greedy Strategy.** The agent is trained using Q-learning with experience replay [18] and epsilon-greedy strategy [19]. With epsilon-greedy strategy, the random action is taken with probability  $\epsilon$  and the greedy action is chosen with probability  $1 - \epsilon$ . We decrease epsilon from 1.0 to 0.1 following the epsilon schedule as shown in Table 2 such that the agent can transform smoothly from exploration to exploitation. We find that the result goes better with a longer exploration, since the search scope would become larger and the agent can see more block structures in the random exploration period.

**Experience Replay.** Following [12], we employ a replay memory to store the validation accuracy and block description after each iteration. Within a given interval, *i.e.* each training iteration, the agent samples 64 blocks with their corresponding validation accuracies from the memory and updates Q-value 64 times.

**BlockQNN Generation.** In the Q-learning update process, the learning rate  $\alpha$  is set to 0.01 and the discount factor  $\gamma$  is 1. We set the hyperparameters  $\mu$  and  $\rho$  in the redefined reward function as 1 and 8, respectively. The agent samples 64 sets of NSC vectors at a time to compose a mini-batch and the maximum layer index

for a block is set to 23. We train the agent with 178 iterations, *i.e.* sampling 11,392 blocks in total.

During the block searching phase, the compute nodes train each generated network for a fixed 12 epochs on CIFAR-100 using the early top strategy as described in Section 3.3. CIFAR-100 contains 60,000 samples with 100 classes which are divided into training and test set with the ratio of 5 : 1. We train the network without any data augmentation procedure. The batch size is set to 256. We use Adam optimizer [53] with  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$ ,  $\epsilon = 10^{-8}$ . The initial learning rate is set to 0.001 and is reduced with a factor of 0.2 every 5 epochs. All weights are initialized as in [54]. If the training result after the first epoch is worse than the random guess, we reduce the learning rate by a factor of 0.4 and restart training, with a maximum of 3 times for restart-operations.

After obtaining one optimal block structure, we build the whole network with stacked blocks and train the network until converging to get the validation accuracy as the criterion to pick the best network. In this phase, we augment data with randomly cropping the images with size of  $32 \times 32$  and horizontal flipping. Besides, we also apply the cutout regularization during training [55]. All models use the SGD optimizer with momentum rate set to 0.9 and weight decay set to 0.0005. We start with a learning rate of 0.1 and train the models for 300 epochs with a single period cosine decay as in [56]. The batch size is set to 128 and all weights are initialized with MSRA initialization [54].

**Transferable BlockQNN.** We also evaluate the transferability of the best auto-generated block structure searched on CIFAR-100 to a smaller dataset, CIFAR-10, with only 10 classes and a larger dataset, ImageNet, containing 1.2M images with 1000 classes. All the experimental settings on CIFAR-10 are the same as those on the CIFAR-100 stated above. ImageNet models are trained on  $224 \times 224$  images and evaluated on  $224 \times 224$  or  $320 \times 320$  images with center crop. The training is conducted with a mini-batch size of 512 where each image has the same data augmentation procedures as described previously [15], and is optimized with SGD strategy. The initial learning rate, weight decay and momentum are set as 0.1, 0.0001 and 0.9, respectively. We decay the learning rate with a single period cosine annealing as in [56]. Additionally, we use label smoothing with a value of 0.1 and an auxiliary classifier located at  $2/3$  of the way up the network with the weight of 0.4 for all ImageNet models as done in [15]. Dropout is applied to the final softmax with probability 0.4.

**Block Connection.** The training process is basically same as the block searching phase. The agent samples 64 sets of NSC vectors for Block Connection at a time to compose a mini-batch and the maximum layer index for a network is set to 12. We train the agent with 46 iterations, *i.e.* sampling 2,944 connection styles in total. Specifically, we allow 5 pooling layers at most in CIFAR task to ensure the resolution is not too small. Moreover, we use the Block-QNN-S, introduced in section 5.4, as the basic block structure for the connection generation.

**Faster BlockQNN.** For the Faster BlockQNN model, we use 40-dimensional vectors for both layer embedding and epoch embedding. The dimension of the hidden states in LSTM is set to 160. The Multi-Layer Perceptron (MLP) for final prediction comprises 3 linear layers, each with 200 hidden units. We randomly sample 2000 block structures and train them on CIFAR-100 to get performance curves for the predictor, the experimental setup is same as the block searching phase. After that, the training process for sampled block structures is replaced by the Faster BlockQNN

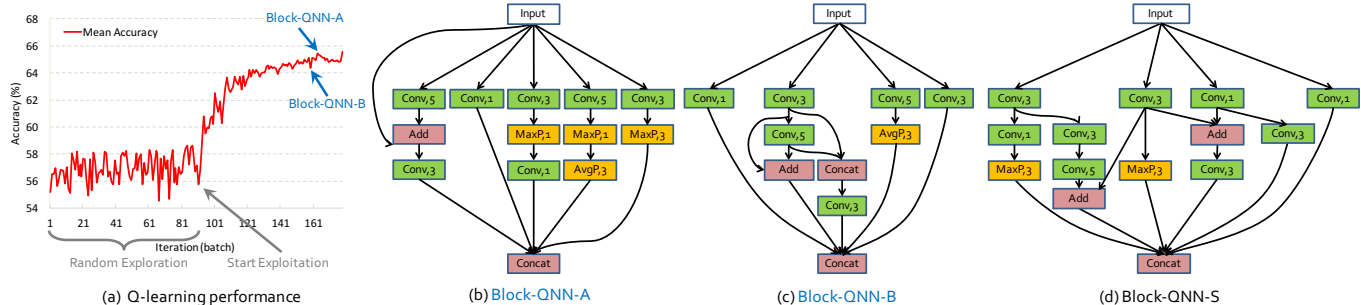


Figure 10. (a) Q-learning performance on CIFAR-100. The accuracy goes up with the epsilon decrease and the top models are all found in the final stage, show that our agent can learn to generate better block structures instead of random searching. (b-c) Topology of the Top-2 block structures generated by our approach. We call them Block-QNN-A and Block-QNN-B. (d) Topology of the best block structures generated with limited parameters, named Block-QNN-S.

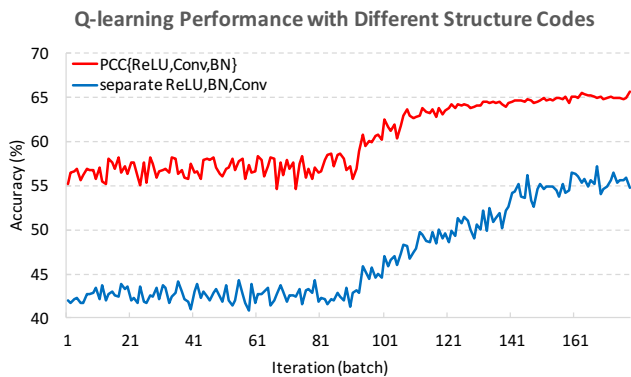


Figure 11. Q-learning result with different NSC on CIFAR-100. The red line refers to searching with PCC, *i.e.* combination of ReLU, Conv and BN. The blue stands for separate searching with ReLU, BN and Conv. The red line is better than blue from the beginning with a big gap.

predictor in the searching phase, other setting is the same as the standard BlockQNN generation.

Our framework is implemented under the PyTorch scientific computing platform. We use the CUDA backend and cuDNN accelerated library in our implementation for high-performance GPU acceleration. Our experiments are carried out on 32 NVIDIA 1080Ti GPUs and took about 3 days to complete searching. Moreover, the faster version can also get a comparable result with only 1 GPU in 20 hours.

## 5 EXPERIMENTAL RESULTS

In this section, we will analyze the training process of block searching and block connection searching. And then, we present experiments on CIFAR and ImageNet, for the purpose of evaluating and comparing the proposed model with other state-of-the-art approaches. Moreover, we will also discuss properties of the evolutionary process and the efficiency of the network architecture generation.

### 5.1 Block Searching Analysis

Fig. 10(a) provides early stop accuracies over 178 batches on CIFAR-100, each of which is averaged over 64 auto-generated block-wise network candidates within in each mini-batch. After random exploration, the early stop accuracy grows steadily till

converges. The mean accuracy within the period of random exploration is 56% while finally achieves 65% in the last stage with  $\epsilon = 0.1$ . We choose top-100 block candidates and train their respective networks to verify the best block structure. We show top-2 block structures in Fig. 10(b-c), denoted as **Block-QNN-A** and **Block-QNN-B**. As shown in Fig. 10(a), both top-2 blocks are found in the final stage of the Q-learning process, which proves the effectiveness of the proposed method in searching optimal block structures rather than randomly searching a large amount of models. Furthermore, we observe that the generated blocks share similar properties with those state-of-the-art hand-crafted networks. For example, Block-QNN-A and Block-QNN-B contain short-cut connections and multi-branch structures which have been manually designed in residual-based and inception-based networks. Compared to other auto-generated methods, the networks generated by our approach are more elegant and can automatically and effectively reveal the beneficial properties for optimal network structure.

To squeeze the search space, as stated in Section 3.1, we define a Pre-activation Convolutional Cell (PCC) consists of three components, *i.e.* ReLU, convolution and batch normalization (BN). We show the superiority of the PCC, searching a combination of three components, in Fig. 11, compared to the separate search of each component. Searching the three components separately is more likely to generate “bad” blocks and also needs more search space and time to pursue “good” blocks.

### 5.2 Block Connection Analysis

Fig. 12 provides early stop accuracies over 46 batches on CIFAR-100, each of which is averaged over 64 auto-generated block connection candidates within in each mini-batch. After random exploration, the early stop accuracy grows steadily. The mean accuracy within the period of random exploration is 56% while finally achieves 62% in the last stage with  $\epsilon = 0.1$ . It is shown that the convergent curve of the Q-learning performance saw in the above is similar with Fig. 10(a). The basic block structure we used is Block-QNN-S will be introduced in section 5.4. We choose top-64 block connection and train their respective networks to verify the best connection style. We show the top block connection in Fig. 13(a), denoted as **Block-QNN-Connection**. Moreover, we observe that the generated blocks connection style is totally different from the sequentially connection. The connection prefer to connect blocks with different resolutions (high resolutions to low resolutions). It may improve the network performance by



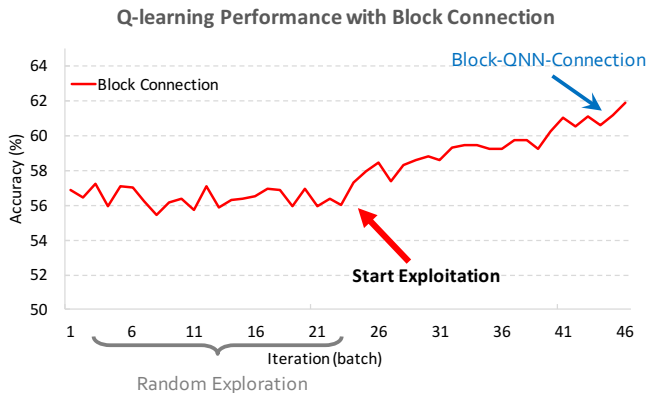


Figure 12. Q-learning performance in block connection generation on CIFAR-100. The accuracy goes up with the epsilon decrease and the top model is found in the final stage. The training process is relatively shorter than the standard BlockQNN because the search space of connection is smaller than the block.

combining features across multiple resolutions. This conception has been widely used in segmentation area [5], but our method found that it also work well in classification task with appropriate connection style. The only weakness is that the generated network is dataset-specific that can not transfer to other task with different input data size. Here, the Block-QNN-Connection can only handle image with  $32 \times 32$ . The exploration of universal block connection formulation with transferable ability is still a open problem.

### 5.3 Faster BlockQNN with 1 GPUs in 20 Hours

As shown in Fig. 14, the mean accuracies over 178 batches on CIFAR-100, each of which is averaged over 64 auto-generated block-wise network candidates within in each mini-batch. After random exploration, the mean accuracy grows shakily. The mean accuracy within the period of random exploration is 56% while finally achieves 60% in the last stage with  $\epsilon = 0.1$ . We can find that the convergent curve of the Q-learning performance saw in the above is different with Fig. 10(a). The mean accuracy is also lower than the standard BlockQNN. It may caused by the errors between Faster BlockQNN predictor and early stop training. We choose top-100 block candidates and train their respective networks to verify the best block structure. As shown in Fig. 14, the top block is found in the final stage of the Q-learning process, besides the mean accuracy grows even if the curve is shakily, which proves the effectiveness of the proposed faster method in searching optimal block structures. We show the top block structures in Fig. 13(b), denoted as **Block-QNN-Faster**. Furthermore, we observe that the top blocks generated by Faster BlockQNN are also different from the top blocks by standard BlockQNN. Block-QNN-Faster doesn't contain multi-branch structure, instead it can be seen as a variant of DenseNet and ResNet. Although the generated block is not complex as the standard BlockQNN blocks, it needs very little computing resources to get comparable performance which is afforded for common deep learning researcher.

### 5.4 Results on CIFAR

Due to the small size of images (*i.e.*  $32 \times 32$ ) in CIFAR, we set block stack number as  $N = 4$ . We compare our generated best architectures with the state-of-the-art hand-crafted networks or auto-generated networks in Table 3.

*Comparison with hand-crafted networks* - It shows that our Block-QNN networks (*i.e.* **Block-QNN-A** and **Block-QNN-B**) outperform most hand-crafted networks. The DenseNet-BC [57] uses additional  $1 \times 1$  convolutions in each composite function and compressive transition layer to reduce parameters and improve performance, which is not adopted in our design. Our performance can be further improved by using this prior knowledge.

*Comparison with auto-generated networks* - Our approach achieves a significant improvement to the MetaQNN [12], and even better than NAS's best model (*i.e.* *NASv3 more filters*) [13] proposed by Google brain which needs an expensive costs on time and GPU resources. As shown in Table 4, NAS trains the whole system on **800 GPUs** in **28 days** while we only need **32 GPUs** in **3 days** to get state-of-the-art performance. Moreover, the faster version can get a comparable result with only **1 GPU** in **20 hours** which is **100x** less expensive than standard BlockQNN.

*Transfer block from CIFAR-100 to CIFAR-10* - We transfer the top blocks learned from CIFAR-100 to CIFAR-10 dataset, all experiment settings are the same. As shown in Table 3, the blocks can also achieve state-of-the-art results on CIFAR-10 dataset with 3.60% error rate that proved Block-QNN networks have powerful transferable ability.

*Analysis on network parameters* - The networks generated by our method might be complex with a large amount of parameters since we do not add any constraints during training. We further conduct an experiment on searching networks with limited parameters and adaptive block numbers. We set the maximal parameter number as 10M and obtain an optimal block (*i.e.* **Block-QNN-S**) which outperforms NASv3 with less parameters, as shown in Fig. 10(d). In addition, when involving more filters in each convolutional layer (*e.g.* from [32,64,128] to [80,160,320]), we can achieve even better result (2.80%).

*Block connection* - The optimal block connection style we find is much better than the sequentially connection with the basic block structure, *i.e.* Block-QNN-S, as shown in Table 3 (*i.e.* from 3.30% to 3.01% with even less parameters). Moreover, when involving more filters in each convolutional layer, we can further improve the performance (2.35%). The empirical evidence indicates that the connection style between blocks can further improve the performance of networks. But as discussed above, the only weakness of block connection is that the generated network is dataset-specific that can not transfer to other task with different input data size.

*Advanced depthwise convolution operation* - To further reduce the computational complexity of the generated network, we introduce the advanced depthwise convolution operation. Each convolution operation refers to a cell with four components, *i.e.* *ReLU*, *Depthwise Convolution*, *Pointwise Convolution* and *Batch Normalization*. Note that we remove Batch Normalization and ReLU between the depthwise and pointwise operations in the cell. After the searching process with depthwise convolution, we obtain an optimal block only with 3.3M parameters (*i.e.* **Block-QNN-depthwise**) which achieve a result with 2.58% error rate on CIFAR-10, as shown in Fig. 13(c). Moreover, our model outperforms all other networks with similar parameters.

### 5.5 Transfer to ImageNet

To demonstrate the generalizability of our approach, we transfer the block structure learned from CIFAR to ImageNet dataset.

For the ImageNet task, we set block repeat number  $N = 3$  and add more down sampling operation before blocks, the filters for

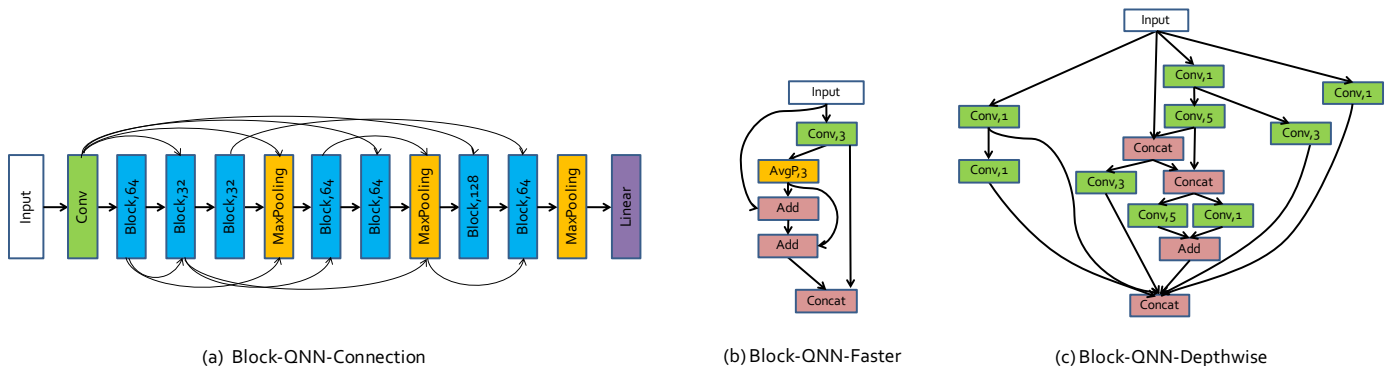


Figure 13. (a) Topology of the best connection style between blocks generated by block Connection, named Block-QNN-Connection. Note that the basic block we used here is Block-QNN-S, *i.e.* Fig. 10(d). (b) Topology of the top block structures generated by Faster BlockQNN. We call it Block-QNN-Faster. (c) Topology of the best block structures generated with advanced depthwise convolution operation, named Block-QNN-Depthwise. All convolution operation be replaced by a cell with four components, *i.e.* *ReLU*, *Depthwise Convolution*, *Pointwise Convolution* and *Batch Normalization*.

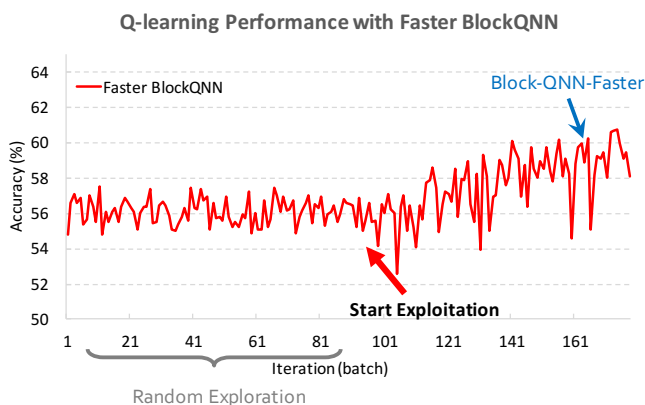


Figure 14. Q-learning performance in faster BlockQNN generation on CIFAR-100. The accuracy goes up with the epsilon decrease and the top model is found in the final stage. We can find that the convergent curve of the Q-learning performance grows shakily, it may caused by the errors between Faster BlockQNN predictor and early stop training

convolution layers in different level blocks are [64,128,256,512]. We use the best blocks structure learned from CIFAR-100 directly without any fine-tuning, and the generated network initialized with MSRA initialization as same as above. The experimental results are shown in Table 5. The network generated by our framework can get competitive result compared with other human designed models. The recently proposed methods such as Xception [27] and ResNext [59] use special depthwise convolution operation to reduce their total number of parameters and to improve performance. In our work, the block structures with depthwise convolution operation, *i.e.* **Block-QNN-Depthwise**, can outperform all these hand-crafted networks. SENet [60] use fully connection layer to recalibrate channel-wise feature responses which we do not adopt this operation in our search space. Besides, SENet can be seen as a Plug-in block to any backbone network, we will consider this in our future work to further improve the performance.

As far as we known, most previous works of automatic network generation did not report competitive result on large scale image classification datasets. With the conception of block learning, we can transfer our architecture learned in small datasets to big dataset like ImageNet task easily. Furthermore, the model

Table 3  
Block-QNN’s results (error rate) compare with state-of-the-art methods on CIFAR-10 (C-10) and CIFAR-100 (C-100) dataset.

Method	Depth	Para	C-10	C-100
VGG [9]	-		7.25	-
ResNet [11]	110	1.7M	6.61	-
Wide ResNet [58]	28	36.5M	4.17	20.5
ResNet (pre-activation) [16]	1001	10.2M	4.62	22.71
DenseNet (k = 12) [57]	40	1.0M	5.24	24.42
DenseNet (k = 12) [57]	100	7.0M	4.10	20.20
DenseNet (k = 24) [57]	100	27.2M	3.74	19.25
DenseNet-BC (k = 40) [57]	190	25.6M	3.46	17.18
MetaQNN (ensemble) [12]	-	-	7.32	-
MetaQNN (top model) [12]	-	11.2M	6.92	27.14
NAS v1 [13]	15	4.2M	5.50	-
NAS v2 [13]	20	2.5M	6.01	-
NAS v3 [13]	39	7.1M	4.47	-
NAS v3 more filters [13]	39	37.4M	3.65	-
NASNet-A (6 @ 768) [26]	-	3.3M	2.65	-
Block-QNN-A, N=4 [20]	25	-	3.60	18.64
Block-QNN-B, N=4 [20]	37	-	3.80	18.72
Block-QNN-S, N=2	19	6.1M	3.30	17.05
Block-QNN-S more filters	22	39.8M	2.80	15.56
Block-QNN-Faster	25	3.9M	3.57	18.21
Block-QNN-Faster more filters	25	34.4M	3.15	16.74
Block-QNN-Connection	22	3.9M	3.01	16.07
Block-QNN-Connection more filters	22	33.3M	2.35	14.83
Block-QNN-Depthwise, N=3	19	3.3M	2.58	15.28

Block-QNN-Depthwise achieves a surprising performance on ImageNet (82.0% Top1 96.0% Top5) when trained on 224x224 and tested on 320x320 images based on the single crop and single model condition. With limited compute resource, we can not train our model on high resolution images (*i.e.* 331x331) directly like NASNet [26]. For a fair comparison, we train Block-QNN-Depthwise and NASNet-A(6 @ 4032) on 224x224 images with same parameter setting. As can be seen from Table 5, our Block-QNN-Depthwise reduces the top-1 error rate by an absolute value

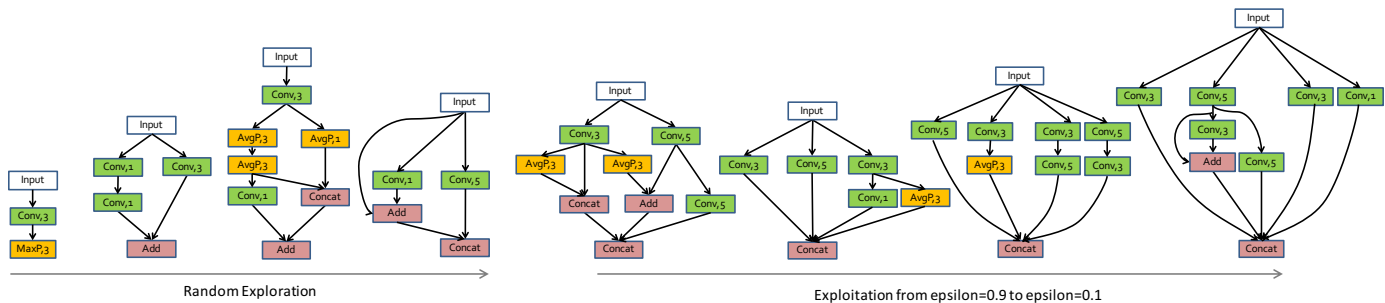


Figure 15. Evolutionary process of blocks generated by BlockQNN. We sample the block structures with median performance at iteration [1, 30, 60, 90, 110, 130, 150, 170] to compare the difference between the blocks in the random exploration stage and the blocks in the exploitation stage.

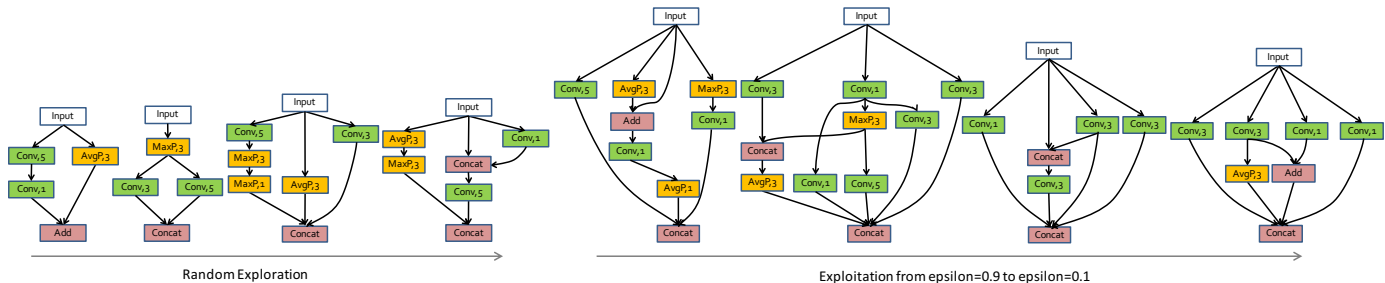


Figure 16. Evolutionary process of blocks generated by BlockQNN with limited parameters and adaptive block numbers (BlockQNN-L). We sample the block structures with median performance at iteration [1, 30, 60, 90, 110, 130, 150, 170] to compare the difference between the blocks in the random exploration stage and the blocks in the exploitation stage.

Table 4

The required computing resource and time of our approach compare with other automatic designing network methods.

Method	Best Model on CIFAR10	GPUs	Time(days)
MetaQNN [12]	6.92	10	10
NAS [13]	3.65	800	28
NASNet-A [26]	2.65	450	3-4
BlockQNN	2.80	32	3
Faster BlockQNN	3.15	1	0.8
Depthwise BlockQNN	2.58	32	3
Connection BlockQNN	2.35	32	1

Table 5

Block-QNN's results (single-crop error rate) compare with modern methods on ImageNet-1K Dataset.

Method	Input Size	Para	Top-1	Top-5
VGG [9]	x224	138M	28.5	9.90
Inception V1 [10]	x224	7M	30.2	10.10
Inception V2 [14]	x224	11M	25.2	7.80
ResNet-50 [16]	x224	26M	24.7	7.80
ResNet-152 [16]	x224	60M	23.0	6.70
Xception(our test)	x224	23M	23.6	7.10
ResNext-101(64x4d) [59]	x224	84M	20.4	5.30
DPN-131 [61]	x224	80M	19.93	5.12
Xception [27]	x299	23M	21.00	5.50
Inception-resnet-v2 [62]	x299	56M	19.90	4.90
Very Deep Inception-ResNet [63]	x299	130M	19.10	4.48
PolyNet [63]	x331	92M	18.71	4.25
DPN-131 [61]	x320	80M	18.55	4.16
NASNet-A(6 @ 4032)(our test)	x224	89M	19.90	5.27
NASNet-A(6 @ 4032) [26]	x331	89M	17.30	3.80
Block-QNN-B, N=3 [20]	x224	-	24.3	7.40
Block-QNN-S, N=3	x224	95M	21.9	6.16
Block-QNN-Depthwise	x224	91M	19.00	4.58
Block-QNN-Depthwise	train x224 test x320	91M	18.00	4.00

of 0.9% compared with the NASNet-A(6 @ 4032). Moreover, the inference speed of Block-QNN-Depthwise (281 Image/Second) is around 30% faster than NASNet-A (222 Image/Second) tested on same computing platform and hardware environment (PyTorch and 8x NVIDIA 1080Ti). The experimental result on ImageNet shows that our auto-generated model can achieve very competitive performances compare with state-of-the-art models.

## 5.6 Evolutionary Process of Auto-Generated Blocks

We sample the block structures with median performance generated by our approach in different stage, *i.e.* at iteration [1, 30, 60, 90, 110, 130, 150, 170], to show the evolutionary process. As illustrated in Figure 15 and Figure 16, *i.e.* BlockQNN and BlockQNN-L respectively, the block structures generated in the random exploration stage is much simpler than the structures generated in the exploitation stage.

In the exploitation stage, the multi-branch structures appear frequently. Note that the connection numbers is gradually increase and the block tend choose "Concat" as the last layer. And we can

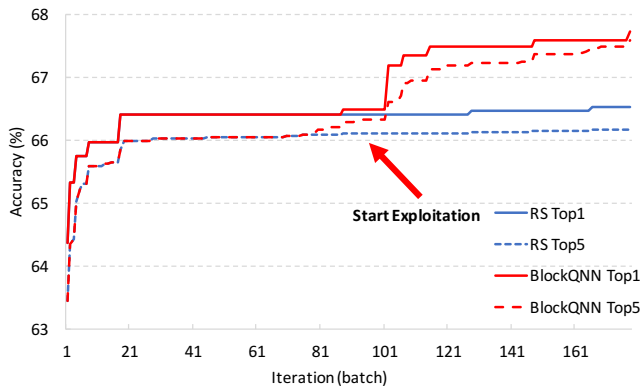


Figure 17. Measuring the efficiency of BlockQNN to random search (RS) for learning neural architectures. The x-axis measures the training iterations (batch size is 64), *i.e.* total number of architectures sampled, and the y-axis is the early stop performance after 12 epochs on CIFAR-100 training. Each pair of curves measures the mean accuracy across top ranking models generated by each algorithm. Best viewed in color.

find that the short-cut connections and elemental add layers are common in the exploitation stage. Additionally, blocks generated by BlockQNN-L have less "Conv,5" layers, *i.e.* convolution layer with kernel size of 5, since the limitation of the parameters.

These prove that our approach can learn the universal design concepts for good network blocks. Compare to other automatic network architecture design methods, our generated networks are more elegant and model explicable.

## 5.7 Efficiency of BlockQNN

We demonstrate the effectiveness of our proposed BlockQNN on network architecture generation on the CIFAR-100 dataset as compared to random search given an equivalent amount of training iterations, *i.e.* number of sampled networks. We define the effectiveness of a network architecture auto-generation algorithm as the increase in top auto-generated network performance from the initial random exploration to exploitation, since we aim to getting optimal auto-generated network instead of promoting the average performance.

Figure 17 shows the performance of BlockQNN and random search (RS) for a complete training process, *i.e.* sampling 11,392 blocks in total. We can find that the best model generated by BlockQNN is markedly better than the best model found by RS by over 1% in the exploitation phase on CIFAR-100 dataset. We observe this in the mean performance of the top-5 models generated by BlockQNN compares to RS. Note that the compared random search method start from the same exploration phase as BlockQNN for fairness.

Figure 18 shows the performance of BlockQNN with limited parameters and adaptive block numbers (BlockQNN-L) and random search with limited parameters and adaptive block numbers (RS-L) for a complete training process. We can see the same phenomenon, BlockQNN-L outperform RS-L by over 1% in the exploitation phase. These results prove that our BlockQNN can learn to generate better network architectures rather than random search.

## 5.8 Additional Experiment

We also use BlockQNN to generate optimal model on person key-points task. The training process is conducted on MPII dataset,

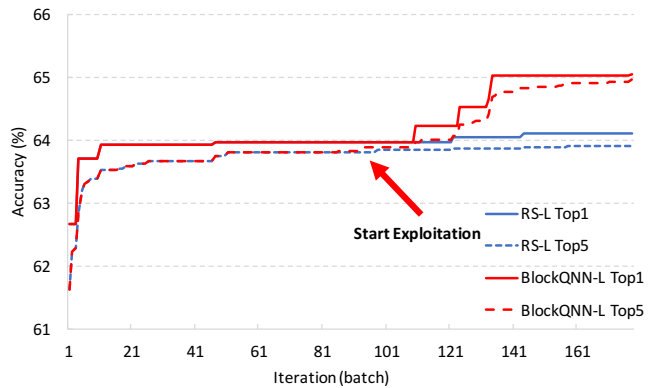


Figure 18. Measuring the efficiency of BlockQNN with limited parameters and adaptive block numbers (BlockQNN-L) to random search with limited parameters and adaptive block numbers (RS-L) for learning neural architectures. The x-axis measures the training iterations (batch size is 64), *i.e.* total number of architectures sampled, and the y-axis is the early stop performance after 12 epochs on CIFAR-100 training. Each pair of curves measures the mean accuracy across top ranking models generated by each algorithm. Best viewed in color.

and then, we transfer the best model found in MPII to COCO challenge. It costs 5 days to complete the searching process. The auto-generated network for key-points task outperform the state-of-the-art hourglass 2 stacks network, *i.e.* 70.5 AP compares to 70.1 AP on COCO validation dataset.

## 6 CONCLUSION AND FUTURE WORK

In this paper, we showed how to efficiently design high performance network blocks with Q-learning, with a distributed asynchronous Q-learning framework and an early stop strategy for fast block structures search. We applied the framework to automatic block generation for constructing good convolutional network. Our Block-QNN networks outperform modern hand-crafted networks and other auto-generated networks in image classification tasks. The best block structure, which achieves a state-of-the-art performance on CIFAR, can be transferred to the large-scale dataset ImageNet easily, and also yield a very competitive performance compared with best hand-crafted networks. We showed that searching with the block design strategy can get more elegant and model explicable network architectures. Furthermore, we propose a faster version which is 100x less expensive than standard BlockQNN and still give a comparable result. The proposed framework makes it possible for common deep learning researchers to join automated neural network design on limited computing resource. We also discussed the different connection style between blocks, and the empirical results showed that sequentially connection is not the optimal style.

This work highlights the general trend from hand-crafted networks to auto-generated networks in deep learning community. In the future, we will try to automatic generate backbone network for other task directly such as detection, segmentation and tracking. For these task, they often use classification network as an backbone, which may be not the optimal choice. Another important future research topic is the automatically preprocessing the input data, designing the network and setting the training hyperparameters in an unified framework.



## ACKNOWLEDGMENTS

This work has been supported by the National Natural Science Foundation of China (NSFC) Grants 61721004 and 61633021.

## REFERENCES

- [1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 2015. **1**
- [2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “Imagenet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, 2012, pp. 1097–1105. **1**
- [3] R. Girshick, “Fast r-cnn,” in *IEEE International Conference on Computer Vision*, 2015, pp. 1440–1448. **1**
- [4] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” in *Advances in Neural Information Processing Systems*, 2015, pp. 91–99. **1**
- [5] J. Long, E. Shelhamer, and T. Darrell, “Fully convolutional networks for semantic segmentation,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3431–3440. **1, 9**
- [6] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, “DeepLab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 40, no. 4, pp. 834–848, 2018. **1**
- [7] H. Nam and B. Han, “Learning multi-domain convolutional neural networks for visual tracking,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 4293–4302. **1**
- [8] L. Bertinetto, J. Valmadre, J. F. Henriques, A. Vedaldi, and P. H. Torr, “Fully-convolutional siamese networks for object tracking,” in *European Conference on Computer Vision*. Springer, 2016, pp. 850–865. **1**
- [9] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *3rd International Conference for Learning Representations*, 2015. **1, 10, 11**
- [10] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, “Going deeper with convolutions,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 1–9. **1, 2, 11**
- [11] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 770–778. **1, 2, 10**
- [12] B. Baker, O. Gupta, N. Naik, and R. Raskar, “Designing neural network architectures using reinforcement learning,” in *6th International Conference on Learning Representations*, 2017. **1, 2, 4, 7, 9, 10, 11**
- [13] B. Zoph and Q. V. Le, “Neural architecture search with reinforcement learning,” in *6th International Conference on Learning Representations*, 2017. **1, 2, 5, 9, 10, 11**
- [14] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *International Conference on Machine Learning*, 2015, pp. 448–456. **1, 2, 11**
- [15] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2818–2826. **1, 2, 7**
- [16] K. He, X. Zhang, S. Ren, and J. Sun, “Identity mappings in deep residual networks,” in *European Conference on Computer Vision*. Springer, 2016, pp. 630–645. **1, 2, 3, 10, 11**
- [17] C. J. C. H. Watkins, “Learning from delayed rewards,” Ph.D. dissertation, King’s College, Cambridge, 1989. **1**
- [18] L.-J. Lin, “Reinforcement learning for robots using neural networks,” Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, Tech. Rep., 1993. **1, 7**
- [19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015. **1, 7**
- [20] Z. Zhong, J. Yan, W. Wu, J. Shao, and C.-L. Liu, “Blockqnn: Practical block-wise neural network architecture generation,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. **2, 10, 11**
- [21] H. Liu, K. Simonyan, O. Vinyals, C. Fernando, and K. Kavukcuoglu, “Hierarchical representations for efficient architecture search,” *arXiv preprint arXiv:1711.00436*, 2017. **2**
- [22] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, “Regularized evolution for image classifier architecture search,” *arXiv preprint arXiv:1802.01548*, 2018. **2**
- [23] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, “Faster discovery of neural architectures by searching for paths in a large model,” *International Conference on Learning Representations Workshop*, 2018. **2**
- [24] C. Liu, B. Zoph, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, “Progressive neural architecture search,” *arXiv preprint arXiv:1712.00559*, 2017. **2**
- [25] H. Cai, J. Yang, W. Zhang, S. Han, and Y. Yu, “Path-level network transformation for efficient architecture search,” *arXiv preprint arXiv:1806.02639*, 2018. **2**
- [26] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. **2, 10, 11**
- [27] F. Chollet, “Xception: Deep learning with depthwise separable convolutions,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017. **2, 10, 11**
- [28] J. D. Schaffer, D. Whitley, and L. J. Eshelman, “Combinations of genetic algorithms and neural networks: A survey of the state of the art,” in *International Workshop on Combinations of Genetic Algorithms and Neural Networks*. IEEE, 1992, pp. 1–37. **2**
- [29] K. O. Stanley and R. Miikkulainen, “Evolving neural networks through augmenting topologies,” *Evolutionary Computation*, vol. 10, no. 2, pp. 99–127, 2002. **2**
- [30] K. O. Stanley, D. B. D’Ambrosio, and J. Gauci, “A hypercube-based encoding for evolving large-scale neural networks,” *Artificial Life*, vol. 15, no. 2, pp. 185–212, 2009. **2**
- [31] M. Sugauma, S. Shirakawa, and T. Nagao, “A genetic programming approach to designing convolutional neural network architectures,” in *Genetic and Evolutionary Computation Conference*, 2017, pp. 497–504. **2**
- [32] S. Saxena and J. Verbeek, “Convolutional neural fabrics,” in *Advances in Neural Information Processing Systems*, 2016, pp. 4053–4061. **2**
- [33] T. Domhan, J. T. Springenberg, and F. Hutter, “Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves,” in *International Joint Conference on Artificial Intelligence*, 2015, pp. 3460–3468. **2, 5**
- [34] L. Xie and A. Yuille, “Genetic cnn,” in *International Conference on Computer Vision*, 2017. **2**
- [35] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, “Imagenet: A large-scale hierarchical image database,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255. **2**
- [36] B. Baker, O. Gupta, R. Raskar, and N. Naik, “Practical neural network performance prediction for early stopping,” *arXiv preprint arXiv:1705.10823*, 2017. **2, 5**
- [37] A. Brock, T. Lim, J. M. Ritchie, and N. Weston, “Smash: one-shot model architecture search through hypernetworks,” *arXiv preprint arXiv:1708.05344*, 2017. **2**
- [38] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, “Algorithms for hyper-parameter optimization,” in *Advances in Neural Information Processing Systems*, 2011, pp. 2546–2554. **3**
- [39] R. Vilalta and Y. Drissi, “A perspective view and survey of meta-learning,” *Artificial Intelligence Review*, vol. 18, no. 2, pp. 77–95, 2002. **3**
- [40] S. Hochreiter, A. S. Younger, and P. R. Conwell, “Learning to learn using gradient descent,” in *International Conference on Artificial Neural Networks*. Springer, 2001, pp. 87–94. **3**
- [41] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, “Learning to learn by gradient descent by gradient descent,” in *Advances in Neural Information Processing Systems*, 2016, pp. 3981–3989. **3**
- [42] M. Lin, Q. Chen, and S. Yan, “Network in network,” in *International Conference on Learning Representations*, 2013. **4**
- [43] A. Y. Ng, D. Harada, and S. Russell, “Policy invariance under reward transformations: Theory and application to reward shaping,” in *International Conference on Machine Learning*, vol. 99, 1999, pp. 278–287. **4**
- [44] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1. **4**
- [45] K. He and J. Sun, “Convolutional neural networks at constrained time cost,” in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 5353–5360. **5**
- [46] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, “Learning curve prediction with bayesian neural networks,” *International Conference on Learning Representations*, 2017. **5**
- [47] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, “Distributed representations of words and phrases and their compositionality,” in *Advances in Neural Information Processing Systems*, 2013, pp. 3111–3119. **6**
- [48] T. Mikolov, M. Karafiát, L. Burget, J. Černocký, and S. Khudanpur, “Recurrent neural network based language model,” in *Eleventh Annual*

- Conference of the International Speech Communication Association*, 2010. 6
- [49] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 4694–4702. 6
- [50] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997. 6
- [51] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems*, 2012, pp. 1223–1231. 7
- [52] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola, "Parameter server for distributed machine learning," in *Big Learning NIPS Workshop*, vol. 6, 2013, p. 2. 7
- [53] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations*, 2015. 7
- [54] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *IEEE International Conference on Computer Vision*, 2015, pp. 1026–1034. 7
- [55] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout," *arXiv preprint arXiv:1708.04552*, 2017. 7
- [56] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations*, 2016. 7
- [57] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017. 9, 10
- [58] S. Zagoruyko and N. Komodakis, "Wide residual networks," in *British Machine Vision Conference*, 2016. 10
- [59] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2017, pp. 5987–5995. 10, 11
- [60] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, 2018. 10
- [61] Y. Chen, J. Li, H. Xiao, X. Jin, S. Yan, and J. Feng, "Dual path networks," in *Advances in Neural Information Processing Systems*, 2017, pp. 4467–4475. 11
- [62] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI Conference on Artificial Intelligence*, vol. 4, 2017, p. 12. 11
- [63] X. Zhang, Z. Li, C. C. Loy, and D. Lin, "Polynet: A pursuit of structural diversity in very deep networks," in *IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 2017, pp. 3900–3908. 11