DYNAMIC GRAPH: LEARNING INSTANCE-AWARE CONNECTIVITY FOR NEURAL NETWORKS

Anonymous authors
Paper under double-blind review

ABSTRACT

One fundamental principle in the design of networks is applying the same architecture to every input sample in a dataset. However, a static architecture may not be representative enough for the whole dataset with high diversity and variety. To promote the model capacity, existing approaches usually add more layers or enlarge the size of convolutional kernels, which may introduce an additional large computational cost. In this paper, we address this issue by raising Dynamic Graph (DY-Graph), which learns the instance-aware connectivity of neural networks, allowing different forward paths for input samples. Topologically, we formulate the network into a directed acyclic graph, where nodes represent convolutional blocks and edges indicate connections. We further rewire the graph as a complete graph and assign learnable weights to edges to adjust the connectivity. During the forward procedure, instead of sharing the calculation paths, DY-Graph aggregates features dynamically in each node based upon their attention, which is input dependent. This allows the network to have more representation power since these features are aggregated in a non-linear way via attention. To adapt to batch training, we represent the connectivity of each sample in an adjacency matrix. And matrices in a batch are cached in a buffer, which is updated along with the forward procedure. In this way, DY-Graph is easy and memory-conserving to train. To demonstrate the superiority of DY-Graph, we validate on several static architectures, including MobileNetV2, ResNet, ResNeXt, and RegNet. Extensive experiments are also conducted on ImageNet classification and COCO object detection to illustrate the effectiveness and generalization of our method.

1 INTRODUCTION

The success of deep convolutional neural networks has driven a shift from feature engineering to feature learning in computer vision. Improvements in performance largely come from well-designed networks with increasing capacity of models (He et al., 2016a; Xie et al., 2017; Huang et al., 2017; Tan & Le, 2019). One fundamental assumption in the design of networks is that the same architecture are applied to every input sample in a dataset. However, the large distribution variance brings difficulties to feature representation. To handle this problem, model developers usually add more layers (Szegedy et al., 2015) or expand the size of existing convolutions (kernel width, number of channels) (Huang et al., 2019; Tan & Le, 2019; Mahajan et al., 2018) to increase the capacity of a model. In either case, the additional cost of capacity increases deploying burden and limits applications with strict computational constraints. To this end, a more flexible *sample-dependent* network is needed to accommodate the input variance without introducing much more computational cost.

Relevantly, there exists some work that attempts to solve this problem using conditional networks with extra sample-dependent modules. Squeeze-and-Excitation network (SENet) (Hu et al., 2018) learns to scale the activations in the *channel* dimension conditionally on the input. Conditionally Parameterized Convolution (CondConv) (Yang et al., 2019) uses over-parameterization weights and generates individual convolutional *kernels* for each sample, achieving large improvements. GaterNet (Chen et al., 2018) adopts a gate network to extract features and generate sparse binary masks for selecting *filters* in the backbone network based upon inputs. But these methods mainly focus on the adjustment of the *micro* structure of neural networks. They are more about combining or reweighting features or weights at the same level of representation. Meanwhile, features generated by convolutional layers with different depths own different semantic information according to

some research (Le et al., 2012; Zeiler & Fergus, 2014). Therefore, we intend to consider a *macro* conditional network, where the features generated by different layers are combined for reconstruction, to strengthen some signals and suppress some signals according to the input. This is very similar to the mammalian brain mechanism in biology (Rauschecker, 1984), where neurons are linked by synapses and responsible for sensing different information. When perceiving external information, synapses are activated to varying degrees.

To adapt the network in a macro type, in this paper, we learn to optimize the connectivity of neural networks based upon inputs. Instead of using stacked-style or hand-designed manners, we allow more flexible selection for forwarding paths. Specifically, we reformulate the network into a directed acyclic graph, where nodes represent aggregation and convolution while edges indicate connections. Different from randomly wired neural networks (Xie et al., 2019) that generate random graphs as connectivity using predefined generators, we rewire the graph as a complete graph, in which all nodes establish connections with each other. This allows for more possible connectivity. In this way, finding the most suitable connectivity for each sample is equivalent to finding the optimal subgraph in the complete graph. In the graph, each node aggregates features from preorder nodes, performs feature transformation (e.g. convolution, normalization, and non-linear operations), and distributes the transformed features to postorder nodes. The output feature of the last node in the topological order is employed as the representation through the graph. To adjust the contribution of different nodes to the feature representation, we further assign weights to the edges in the graph. The weights are generated dynamically for each input via input dependent attention. This is implemented by adding an extra module (denoted as *router*) along with each node, which predicts the weights of edges connected with the node. During inference, only crucial connections are maintained. We call this method *dynamic graph* (denoted as DY-Graph).

In DY-Graph, the connectivity for each sample is generated through non-linear functions determined by routers, which has more representation power than its static counterpart. Meanwhile, DY-Graph is computationally efficient. It does not increase the depth or width of the network. It only introduces extra computational cost to compute attention and aggregate features. The key insight is that within reasonable cost of model size to increase the representation capacity of a network. The main contributions of our work are as follows, and we believe this is a promising direction and hope it would have a border impact on the vision community:

- We *first* explore increasing the model capacity of neural networks through introducing dynamic connectivity based upon inputs. Without bells and whistles, simply replacing static connectivity with dynamic one in many networks achieves solid improvement with only a slight increase of ($\sim 1\%$) parameters and ($\sim 2\%$) computational cost (see table 1).
- DY-Graph is easy and memory-conserving to train. The parameters of networks and routers can be optimized jointly in a differentiable manner. To support batch training and be compatible with current deep learning libraries, we propose a caching mechanism that keeps the information of connectivity along with the forward procedure.
- We show that DY-Graph not only improves the performance for human-designed networks (e.g. MobielNetV2, ResNet, ResNeXt), but also boosts the performance for automatically searched architectures (e.g. RegNet).
- DY-Graph demonstrates good generalization ability on ImageNet classification (see table 1) and COCO object detection (see table 2) tasks.

2 Related Work

Non-modular Network Wiring. Different from the modularized designed network which consists of topologically identical blocks, there exists some work that explores more flexible wiring patterns. MaskConnect (Ahmed & Torresani, 2018) removes predefined architectures and learns the connections between modules in the network with k conenctions. Randomly wired neural networks (Xie et al., 2019) use classical graph generators to yield random wiring instances and achieve competitive performance with manually designed networks. DNW (Wortsman et al., 2019) treats each channel as a node and searches a fine-grained sparse connectivity among layers. TopoNet (Yuan et al., 2020) learns to optimize the connectivity of neural networks in a complete graph that adapt to the specific task. Prior work demonstrates the potential of exploring more flexible wirings, our work on DY-Graph

pushes the boundaries of this paradigm, by enabling each example to be processed with different connectivity.

Dynamic Networks. Dynamic networks, adjusting the network architecture to the corresponding input, have been recently studied in the computer vision domain. SkipNet (Wang et al., 2018), BlockDrop (Wu et al., 2018) and HydraNet (Mullapudi et al., 2018) use reinforcement learning to learn the subset of blocks needed to process a given input. Some approaches prune channels (Lin et al., 2017a; You et al., 2019) for efficient inference. However, most prior methods are challenging to train, because they need to obtain discrete routing decisions from individual examples. Different from these approaches, DY-Graph learns continuous weights for connectivity to enable ramous propagation of features, so can be easily optimized in a differentiable way.

Conditional Attention. Some recent work proposes to adapt the distribution of features or weights through attention conditionally on the input. SENet (Hu et al., 2018) boosts the representational power of a network by adaptively recalibrating channel-wise feature responses by assigning attention over channels. CondConv (Yang et al., 2019) and dynamic convolution (Chen et al., 2020) are restricted to modulating different experts/kernels, resulting in attention over convolutional weights. Attention-based models are also widely used in language modeling (Luong et al., 2015; Bahdanau et al., 2015; Vaswani et al., 2017), which scale previous sequential inputs based on learned attention weights. In the vision domain, previous methods most compute attention over *micro* structure, ignoring the influence of the features produced by different layers on the final representation. Unlike these approaches, DY-Graph focuses on learning the connectivity based upon inputs, which can be seen as attention over features with different semantic hierarchy.

Neural Architecture Search. Recently, Neural Architecture Search (NAS) has been widely used for automatic network architecture design. With evolutionary algorithm (Real et al., 2019), reinforcement learning (Pham et al., 2018) or gradient descent (Liu et al., 2019), one can obtain task-dependent architectures. Different from these NAS-based approaches, which search for a single architecture, the proposed DY-Graph generates forward paths on the fly according to the input without searching. We also notice a recent method InstaNAS (Cheng et al., 2020) that generates domain-specific architectures for different samples. It trained a controller to select child architecture from the defined meta-graph by reinforcement learning, achieving latency reduction during inference. Different from them, DY-Graph focuses on learning connectivity in a complete graph using a differentiable way and achieves higher performance.

3 METHODOLOGY

3.1 FORMULATING CONNECTIVITY USING DAGS

Regular networks usually use chain-like wiring patterns. ResNets (He et al., 2016a) use $x + \mathcal{F}(x)$ as a regular connectivity template and stack modules to form the network. DenseNets (Huang et al., 2017) concat features from previous layers with $[x, \mathcal{F}(x)]$. But their connectivity is inflexible, and arbitrary changes can lead to a mismatch in the dimension of channels. Some NAS methods (Real et al., 2019; Liu et al., 2019) also attempt to find efficient patterns of wiring and operation. However, these methods constrain the connectivity in a small space, in which layers only connect two immediately preceding layers and generate output to a subsequent layer. To remove these limitations and gain larger search space, the connectivity pattern needs to be redefined.

In this work, we convert the network into a directed acyclic graph (DAG), in which nodes stand for feature transformation (e.g. aggregation, convolution, normalization, and non-linear activation) and edges represent connections between layers. For simplicity, we denote a DAG with N ordered nodes as $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes, and $\mathcal{E}$ denotes the set of edges and $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$.

In the graph, nodes aggregate features from connected preceding layers and conduct transformations. Specifically, the first node in the topological order is set to be the *input* node that only allocates features to posterior nodes. And the last node is set to be the *output* one, which only aggregates features from preceding nodes for the final representation. For the i-th node, corresponding mapping function is denoted as $o^i(\cdot, \cdot)$. The set of edges can be represented as $\mathcal{E} = \{e^{(i,j)} | 1 \leq i < j \leq N\}$, where $e^{(i,j)}$ indicates a directed edge from the i-th node to the j-th node. With each edge $e^{(i,j)}$, we

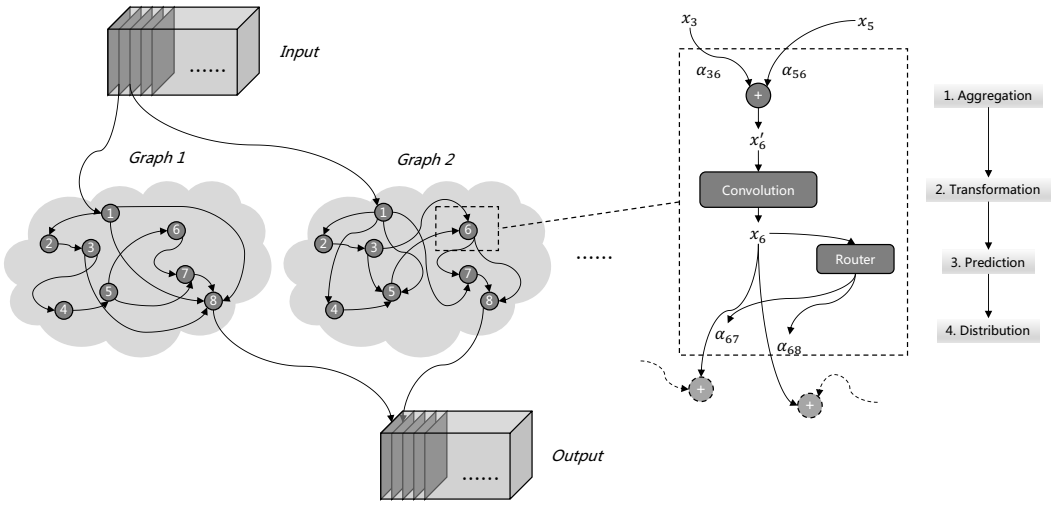

Figure 1: The framework of DY-Graph. *Left*: For a training batch, each sample performs different forward paths that are determined by the data-dependent macro connectivity. *Right*: Node operations at the micro level. Here we illustrate a node with 2 input edges and 2 output edges. First, it aggregates input features from preceding nodes by weighted sum. Second, convolutional blocks make a transformation to the aggregated features. Third, a router predicts routing weights for each sample on the output edges according to the transformed features. Last, the transformed data is sent out by the output edges to the following nodes. Arrows indicate the data flow.

associate a probability α^{ij} which determines the importance of connections. For a node, the number of input edges is called *in-degree* and denoted as δ^i. While the number of output edges is called *out-degree* and denoted as ζ^i. In a common *static* graph, the output at the j-th node can be formulated as:

$$\mathbf{x}^j = o^j(\mathbf{w}_o^j, \sum_{e^{(i,j)} \in \mathcal{E}} \alpha^{ij} \cdot \mathbf{x}^i) \tag{1}$$

where $\mathbf{x}^i \in \mathbb{R}^{B \times C \times H \times W}$ denotes the transformed feature from the i-th node and $\mathbf{w}_o^j$ are the convolutional weights of the j-th node.

To expand the search space with more possible connectivity, two modifications are made in DY-Graph. First, we remove the constraint on the in/out-degree of nodes, so that the connectivity is rewired in a *complete* graph, which contains all possible combinations of forwarding paths. In this way, finding good connectivity is akin to finding an optimal sub-graph. Second, instead of selecting discrete edges in a hard way where $\alpha^{ij} \in \{0, 1\}$, we convert the connections to a *soft* form with $\alpha^{ij} \in [0, 1)$. When $\alpha^{ij} = 0$, the edge from i-th node to j-th node will be marked as closed. And all the edges with $\alpha^{ij} > 0$ will be reserved, continuously enabling feature fusion. This makes the connectivity can be optimized in a differentiable manner, and will be covered in detail in section 3.3.

3.2 Instance-aware Connectivity Through Routing Mechanism

We propose a dynamic graph, which does not increase either the depth or the width of the network, but increase the model capability by aggregating features generated by different nodes via attention. Note that these features are assembled differently for different input images. For each input sample $\mathbf{x}_b$, as shown in the left of Fig. 1, there exists an individual connectivity $\mathcal{E}_b$ to determine the forward paths. Compared with the *static* type in Eq. (1), for a batch containing B samples, the *dynamic* type can be rewrittern as:

$$\mathbf{x}^j = [\mathbf{x}_1^j, \cdots, \mathbf{x}_b^j, \cdots, \mathbf{x}_B^j], \text{ and } \mathbf{x}_b^j = o^j(\mathbf{w}_o^j, \sum_{e^{(i,j)} \in \mathcal{E}_b} \alpha_b^{ij} \cdot \mathbf{x}_b^i) \tag{2}$$

where $\mathbf{x}_b^i \in \mathbb{R}^{1 \times C \times H \times W}$, and α_b^{ij} represents the instance-aware probability of the edge. In particular, except for the probabilities of edges, the parameters of the network are shared.

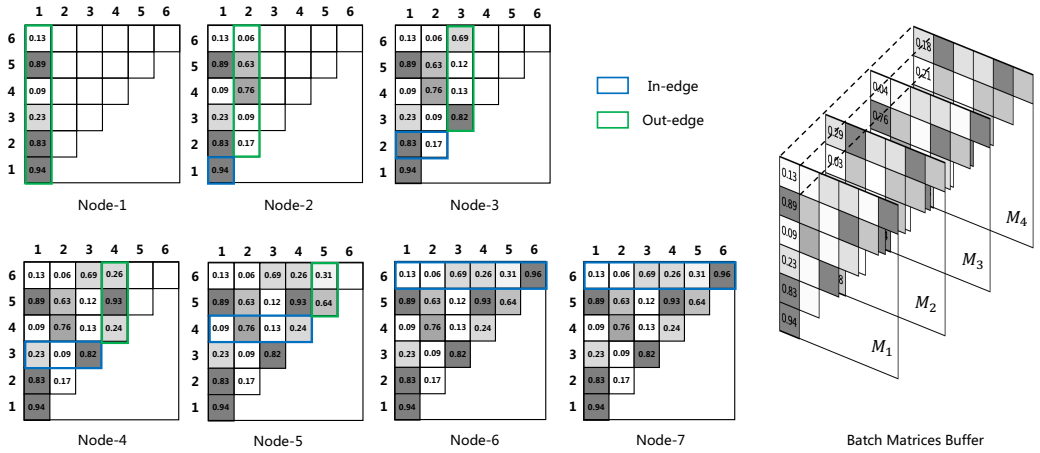

Figure 2: The procedure of updating the coefficients in the adjacency matrix (left) and the proposed buffer for storing matrices in a batch (right). The i-th node obtains the weights of input edges from the (i-1)-th row (blue) and generates weights to output edges saving in the i-th column (green). The matrices are saved in a buffer that supports training in a large batch efficiently.

The probabilities of edges are generated from an extra router module along with each node, as presented in the right diagram of Fig. 1. For the i-th node, the router receives the transformed feature and apply squeeze-and-excitation to compute attentions $\boldsymbol{\pi}^i$ over connections with posterior nodes. Structurally, the router applies a lightweight module consisted of a global average pooling $\varphi(\cdot)$, a fully-connected layer $\mathcal{F}(\cdot,\cdot)$ and a sigmoid activation $\sigma(\cdot)$. The global spatial information is firstly squeezed by global average pooling. Then we use a fully-connected layer and sigmoid to generate normalized routing attentions for ζ^i output edges. The operations of router can be written as:

$$\boldsymbol{\pi}^i = \sigma(\mathcal{F}(\mathbf{w}_r^i, \varphi(\mathbf{x}^i)) + \mathbf{b}_r^i) \tag{3}$$

where $\mathbf{w}_r^i$ and $\mathbf{b}_r^i$ are weights and bias of the fully-connected layer. And $\boldsymbol{\pi}^i \in \mathbb{R}^{B\times\zeta^i}$ contains routing weights of existing connections for a batch. Specifically, the b-th input will generate individual $\alpha_b^{ij} = \boldsymbol{\pi}_{b,j}^i \in \mathbb{R}^{1\times1\times1\times1}$, which means the connectivity varies with inputs, or so called *instance-aware*. Particularly, DY-Graph is computationally efficient because of the dimension reduction of $\varphi(\cdot)$ from $C \times H \times W$ to a 1-D dimension. This is much less than the computational cost of convolution,

Besides, we set a learnable weight of τ that acts as a threshold for each node to control the connectedness. When the weight is less than the threshold, the connection will be closed. If the input or output edges for a node are closed, the node will be removed to accelerate inference. This can be noted as:

$$\alpha^{ij} = \begin{cases} 0 & \alpha^{ij} < \tau \\ \alpha^{ij} & \alpha^{ij} \geq \tau \end{cases} \tag{4}$$

3.3 Buffer Mechanism for Batch Training

DY-Graph needs individual graphs to adjust the aggregation of features for each sample during the forward procedure. But most current neural network libraries such as PyTorch (Paszke et al., 2019) or TensorFlow (Abadi et al., 2015) do not work well with dynamic networks that change for every training instance in a batch. It is inefficient to take one sample at a time during training. Meanwhile, some optimization techniques in deep learning such as BatchNorm (Ioffe & Szegedy, 2015) is highly dependent on large batch training. To resolve this contradiction and fully utilize GPU accelerators, we propose using a buffer mechanism to support batch training.

Specifically, for the b-th input, the connectivity which determines the graph can be represented in an adjacency matrix (denoted as $\boldsymbol{M}_b \in \mathbb{R}^{N\times N}$). An example is given in the left of Fig. 2, where rows reflect the coefficients of input edges and columns are of output edges for a node. During the forward procedure, each node performs aggregation through weights *acquired* from the corresponding row of $\boldsymbol{M}_b$. Then the node generates attentions over output edges through accompanying the router and

stores them into the columns of $\boldsymbol{M}_b$. In this way, the adjacency matrix is updated progressively and shared within the graph. For a batch with B samples, different matrices are concatenated and cached in a defined buffer (denoted as $\mathbf{M} \in \mathbb{R}^{B\times N\times N}$, where $\mathbf{M}_{b,:,:} = \boldsymbol{M}_b$), as shown in the right of Fig. 2. Under this mechanism, the DY-Graph is compatible with batch training. The predicted weights in Eq. (3) can be saved by $\mathbf{M}_{:,i,:} = \boldsymbol{\pi}^i$. For feature aggregation, the attention weights can also be obtained through $\boldsymbol{\pi}^j = \mathbf{M}_{:,:,j}$. This guarantees that DY-Graph can be trained like ordinary batch training without introducing excessive computation or time-consuming burden.

3.4 Optimization of DY-Graph

During training, the parameters of the network $\boldsymbol{\mathcal{W}}_o = \{\mathbf{w}_o^i\}$, as well as the parameters of routers $\boldsymbol{\mathcal{W}}_r = \{\mathbf{w}_r^i, \mathbf{b}_r^i\}$, are optimized simultaneously using gradients back-propagation. Given an input $\mathbf{x}$ and corresponding label $\mathbf{y}$, the objective function can be represented as:

$$\min_{\boldsymbol{\mathcal{W}}_o, \boldsymbol{\mathcal{W}}_r} \mathcal{L}_t(\Gamma(\mathbf{x}; \boldsymbol{\mathcal{W}}_o, \boldsymbol{\mathcal{W}}_r), \mathbf{y}) \tag{5}$$

where $\Gamma(\cdot,\cdot)$ is the mapping function from the sample to the feature representation, and $\mathcal{L}_t(\cdot,\cdot)$ denotes the loss function w.r.t specific tasks (e.g. cross-entropy loss for image classification).

4 Experiments

4.1 ImageNet Classification

Dataset and evaluation metrics. We evaluate our approach on the ImageNet 2012 classification dataset (Russakovsky et al., 2015). The ImageNet dataset consists of 1.28 million training images and 50,000 validation images from 1000 classes. We train all models on the entire training set and compare the single-crop top-1 validation set accuracy with input image resolution 224×224. We measure performance as ImageNet top-1 accuracy relative to the number of parameters and computational cost in FLOPs.

Model selection and training setups. We validate our approach on a number of widely used models including MobileNetV2-1.0 (Sandler et al., 2018), ResNet-18/50/101 (He et al., 2016a) and ResNeXt50-32x4d (Xie et al., 2017). To further test the effectiveness of DY-Graph, we attempt to optimize recent NAS-based networks of RegNets (Radosavovic et al., 2020), which are the best models out of a search space with $\sim 10^{18}$ possible configurations. Our implementation is based on PyTorch (Paszke et al., 2019) and all experiments are conducted using 16 NVIDIA Tesla V100 GPUs with a total batch of 1024. All models are trained using SGD optimizer with 0.9 momentum. Detailed information about the training setting can be seen in appendix 6.1.

Table 1: ImageNet validation accuracy (%) and inference cost for our DY-Graph models on several baseline model architectures. DY-Graph improves the accuracy of all baseline architectures with small relative increase in the number of parameters and inference cost.

Network	Baselines			DY-Graph			Δ Top-1
	Params(M)	FLOPs(M)	Top-1	Params(M)	FLOPs(M)	Top-1	
MobileNetV2-1.0	3.51	299	72.60	3.58	312	**73.17**	0.57
ResNet18	11.69	1813	70.30	11.71	1826	**71.30**	1.00
ResNet50	25.55	4087	76.70	25.62	4125	**78.00**	1.30
ResNet101	44.54	7799	78.29	44.90	7837	**79.68**	1.39
ResNeXt50-32x4d	25.02	4228	77.97	25.09	4305	**79.18**	1.21
RegNet-X-600M	6.19	600	74.03	6.22	601	**74.60**	0.57
RegNet-X-1600M	9.19	1602	77.26	9.22	1604	**77.86**	0.60

Analysis of experimental results. We verify that DY-Graph improves performance on a wide range of architectures in Table 1. For fair comparison, we retrain all of our baseline[1] models with the same

[1]Our re-implementation of the baseline models and our DY-Graph models use the same hyperparameters. For reference, published results for baselines are: MobileNetV2-1.0 (Sandler et al., 2018): 72.00%, ResNet-

hyperparameters as the DY-Graph models. Compared with baselines, DY-Graph gets considerable gains with a small relative increase in the number of parameters ($< 2\%$) and inference cost of FLOPs ($< 1\%$). This includes architectures with mobile setting (Sandler et al., 2018), classical residual wirings (He et al., 2016a; Xie et al., 2017), multi-branch operation (Xie et al., 2017) and architecture search (Radosavovic et al., 2020). We further find that DY-Graph benefits from the large search space which can be seen in the improvements of ResNets. With the increase of the depth from 18 to 101, the formed complete graph includes more nodes, resulting in larger search space and more possible connectivity. And the gains raise from 1.00% to 1.39% in top-1 accuracy.

4.2 COCO Object Detection

We report the transferability results by fine-tuning the networks for COCO object detection (Lin et al., 2014). We use Faster R-CNN (Ren et al., 2015) with FPN (Lin et al., 2017b) as the object detector. Our fine-tuning is based on the $1\times$ setting of the publicly available `Detectron` (Girshick et al., 2018). We replace the backbone with those in Table 1.

The object detection results are given in Table 2. And FLOPs of the backbone are computed with an input size of 800×1333. Compared with the static network, DY-Graph improves AP by 1.55% with ResNet-50 backbone. When using a larger search space of ResNet101, our method significantly improves the performance by 2.63% in AP. It is worth noting that stable gains are obtained for objects of different scales varying from small to large. This further verifies that instance-aware connectivity can improve the representation capacity toward the dataset with a large distribution variance.

Table 2: COCO object detection *minival* performance of our DY-Graph with FPN as the object detection method. APs (%) of bounding box detection are reported. Note that DY-Graph brings consistently and substantially improvement across multiple backbones on all scales.

Backbone	Method	GFLOPs	AP	$AP_{.5}$	$AP_{.75}$	AP_S	AP_M	AP_L
ResNet50	Baseline	174	36.42	58.54	39.11	21.93	40.02	46.58
	DY-Graph	176	**37.97**$_{+1.55}$	60.42	40.79	23.40	41.36	48.21
ResNet101	Baseline	333	38.59	60.56	41.63	22.45	43.08	49.46
	DY-Graph	335	**41.22**$_{+2.63}$	63.51	44.86	25.57	45.47	52.56
ResNeXt50-32x4d	Baseline	181	38.07	60.42	41.01	22.97	42.10	48.68
	DY-Graph	183	**39.22**$_{+1.15}$	62.16	42.41	25.23	43.12	49.59

4.3 Compared with Related Work

Compared with InstaNAS (Cheng et al., 2020). InstaNAS generates data-dependent networks from a designed meta-graph. During inference, it uses a controller to sample possible architectures by a Bernoulli distribution. But it needs to carefully design the training process to avoid collapsing the controller. Differently, DY-Graph builds continuous connections between nodes, which allowing more possible connectivity. And the proposed method is compatible with gradient descent, and can be trained in a differentiable way easily. MobileNetV2 is used as the backbone network in InstaNAS. It provides multiple searched architectures under different latencies. For a fair comparison, DY-Graph adopts the same structure as the backbone and reports the results of ImageNet. The latency is tested using the same hardware. The results in Table 3 demonstrate DY-Graph can generate better instance-aware architectures in the dimension of connectivity.

Compared with RandWire (Xie et al., 2019). Randomly wired neural networks explore using flexible graphs generated by different graph generators as networks, losing the constraint on wiring patterns. But for the entire dataset, the network architecture it uses is still consistent. Furthermore, DY-Graph allows instance-aware connectivity patterns learned from the complete graph. We compare three types of generators in their paper with best hyperparameters, including Erdös-Rényi (ER), Barabási-Albert (BA) and Watts-Strogatz (WS). Since the original paper does not release codes,

18 (He et al., 2016b), 69.57%, ResNet-50 (Goyal et al., 2017): 76.40%, ResNet-101 (Goyal et al., 2017): 77.92%, ResNeXt50-32x4d (Xie et al., 2017): 77.80%, RegNetX-600M (Radosavovic et al., 2020): 74.10%, RegNetX-1600M (Radosavovic et al., 2020): 77.00%.

we reproduce these graphs using `NetworkX`[2]. We follow the small computation regime to form networks. Experiments are performed in ImageNet using its original training setting except for the DropPath and DropOut. Comparision results are shown in Table 4. DY-Graph is superior to three classical graph generators in a similar computational cost. This proves that under the same search space, the optimized data-dependent connectivity is better than randomly wired static connectivity.

Table 3: Compared with InstaNAS under comparable latency in ImageNet.

Model	Top-1	Latency (ms)
MobileNetV2-1.0	72.6	$0.257_{\pm 0.015}$
InstaNAS-ImgNet-A	71.9	$0.239_{\pm 0.014}$
InstaNAS-ImgNet-B	71.1	$0.189_{\pm 0.012}$
InstaNAS-ImgNet-C	69.9	$0.171_{\pm 0.011}$
DY-Graph-MBv2-1.0	**73.2**	$0.257_{\pm 0.015}$

Table 4: Compared with RandWire under small computation regime in ImageNet.

Wiring Type	Top-1	FLOPs(M)
ER (P=0.2)	71.5	602
BA (M=5)	71.2	582
WS (K=4, P=0.75)	72.2	572
DY-Graph	**73.1**	611

Compared with state-of-the-art NAS-based methods. For completeness, we compare with the most accurate NAS-based networks under the mobile setting ($\sim$ 600M FLOPs) in ImageNet. It is worth noting that this is *not* the focus of this paper. We select RegNet as the basic architecture as shown in Table 1. For fair comparisons, here we train 250 epochs and other settings are the same with section 4.1. We note RegNet-X with the dynamic graph as DY-Graph-A and RegNet-Y with dynamic graph as DY-Graph-B (with SE-module for comparison with particular searched architectures e.g. EfficientNet). The experimental results are given in Table 5. It shows that with a single operation type (*Regular Bottleneck*), DY-Graph can obtain considerable performance with other NAS methods.

Table 5: Comparision with NAS methods under mobile setting. Here we train for 250 epochs similar to (Zoph et al., 2018; Real et al., 2019; Xie et al., 2019; Liu et al., 2018; 2019), for fair comparisons.

Network	Params(M)	FLOPs(M)	Top-1	Top-5
NASNet-A (Zoph et al., 2018)	5.3	564	74.0	91.6
NASNet-B (Zoph et al., 2018)	5.3	488	72.8	91.3
NASNet-C (Zoph et al., 2018)	4.9	558	72.5	91.0
Amoeba-A (Real et al., 2019)	5.1	555	74.5	92.0
Amoeba-B (Real et al., 2019)	5.3	555	74.0	91.5
Amoeba-C (Real et al., 2019)	6.4	570	75.7	92.4
RandWire-WS (Xie et al., 2019)	5.6	583	74.7	92.2
PNAS (Liu et al., 2018)	5.1	588	74.2	91.9
DARTS (Liu et al., 2019)	4.9	595	73.1	91.0
EfficientNet-B0 (Tan & Le, 2019)	5.3	390	76.3	93.2
DY-Graph-A	6.2	600	75.4	92.3
DY-Graph-B	6.2	600	76.7	92.9

5 Conclusion and Future Work

In this paper, we present the Dynamic Graph (noted as DY-Graph), that allows learning instance-aware connectivity for neural networks. Without introducing much computation cost, the model capacity can be increased to ease the difficulties of feature representation for samples with high diversity and variety. We show that DY-Graph is superior to many static networks, including human-designed and automatically searched architectures. Besides, DY-Graph demonstrates good generalization ability on ImageNet classification as well as COCO object detection. It also achieved SOTA results compared with related work. DY-Graph explores the connectivity in an enlarged search space, which we believe is a new research direction. In future work, we consider verifying DY-Graph on more NAS-searched architectures. Moreover, we will study learning dynamic operations beyond the connectivity as well as adjusting the computation cost based upon the difficulties of samples.

[2] https://networkx.github.io

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

Karim Ahmed and Lorenzo Torresani. Maskconnect: Connectivity learning by gradient descent. In *ECCV (5)*, volume 11209 of *Lecture Notes in Computer Science*, pp. 362–378. Springer, 2018.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *ICLR*, 2015.

Yinpeng Chen, Xiyang Dai, Mengchen Liu, Dongdong Chen, Lu Yuan, and Zicheng Liu. Dynamic convolution: Attention over convolution kernels. In *CVPR*, pp. 11027–11036. IEEE, 2020.

Zhourong Chen, Yang Li, Samy Bengio, and Si Si. Gaternet: Dynamic filter selection in convolutional neural network via a dedicated global gating network. *CoRR*, abs/1811.11205, 2018.

An-Chieh Cheng, Chieh Hubert Lin, Da-Cheng Juan, Wei Wei, and Min Sun. Instanas: Instance-aware neural architecture search. In *AAAI*, pp. 3577–3584. AAAI Press, 2020.

Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár, and Kaiming He. Detectron. https://github.com/facebookresearch/detectron, 2018.

Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677, 2017.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, pp. 770–778. IEEE Computer Society, 2016a.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *ECCV (4)*, volume 9908 of *Lecture Notes in Computer Science*, pp. 630–645. Springer, 2016b.

Jie Hu, Li Shen, and Gang Sun. Squeeze-and-excitation networks. In *CVPR*, pp. 7132–7141. IEEE Computer Society, 2018.

Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *CVPR*, pp. 2261–2269. IEEE Computer Society, 2017.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *NeurIPS*, pp. 103–112, 2019.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML*, volume 37 of *JMLR Workshop and Conference Proceedings*, pp. 448–456. JMLR.org, 2015.

Quoc V. Le, Marc'Aurelio Ranzato, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Jeffrey Dean, and Andrew Y. Ng. Building high-level features using large scale unsupervised learning. In *ICML*. icml.cc / Omnipress, 2012.

Ji Lin, Yongming Rao, Jiwen Lu, and Jie Zhou. Runtime neural pruning. In *NIPS*, pp. 2181–2191, 2017a.

Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In *ECCV (5)*, volume 8693 of *Lecture Notes in Computer Science*, pp. 740–755. Springer, 2014.

Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. In *CVPR*, pp. 936–944. IEEE Computer Society, 2017b.

Chenxi Liu, Barret Zoph, Maxim Neumann, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. In *ECCV (1)*, volume 11205 of *Lecture Notes in Computer Science*, pp. 19–35. Springer, 2018.

Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. In *ICLR (Poster)*. OpenReview.net, 2019.

Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *EMNLP*, pp. 1412–1421. The Association for Computational Linguistics, 2015.

Dhruv Mahajan, Ross B. Girshick, Vignesh Ramanathan, Kaiming He, Manohar Paluri, Yixuan Li, Ashwin Bharambe, and Laurens van der Maaten. Exploring the limits of weakly supervised pretraining. In *ECCV (2)*, volume 11206 of *Lecture Notes in Computer Science*, pp. 185–201. Springer, 2018.

Ravi Teja Mullapudi, William R. Mark, Noam Shazeer, and Kayvon Fatahalian. Hydranets: Specialized dynamic architectures for efficient inference. In *CVPR*, pp. 8080–8089. IEEE Computer Society, 2018.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, pp. 8024–8035, 2019.

Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. In *ICML*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4092–4101. PMLR, 2018.

Ilija Radosavovic, Raj Prateek Kosaraju, Ross B. Girshick, Kaiming He, and Piotr Dollár. Designing network design spaces. *CoRR*, abs/2003.13678, 2020.

JP Rauschecker. Neuronal mechanisms of developmental plasticity in the cat's visual system. *Human neurobiology*, 3(2):109–114, 1984.

Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, pp. 4780–4789. AAAI Press, 2019.

Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. In *NIPS*, pp. 91–99, 2015.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Fei-Fei Li. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015.

Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *CVPR*, pp. 4510–4520. IEEE Computer Society, 2018.

Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1): 1929–1958, 2014.

Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *CVPR*, pp. 1–9. IEEE Computer Society, 2015.

Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, pp. 4278–4284. AAAI Press, 2017.

Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *ICML*, volume 97 of *Proceedings of Machine Learning Research*, pp. 6105–6114. PMLR, 2019.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *NIPS*, pp. 5998–6008, 2017.

Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In *ECCV (13)*, volume 11217 of *Lecture Notes in Computer Science*, pp. 420–436. Springer, 2018.

Mitchell Wortsman, Ali Farhadi, and Mohammad Rastegari. Discovering neural wirings. In *NeurIPS*, pp. 2680–2690, 2019.

Zuxuan Wu, Tushar Nagarajan, Abhishek Kumar, Steven Rennie, Larry S. Davis, Kristen Grauman, and Rogério Schmidt Feris. Blockdrop: Dynamic inference paths in residual networks. In *CVPR*, pp. 8817–8826. IEEE Computer Society, 2018.

Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *CVPR*, pp. 5987–5995. IEEE Computer Society, 2017.

Saining Xie, Alexander Kirillov, Ross B. Girshick, and Kaiming He. Exploring randomly wired neural networks for image recognition. In *ICCV*, pp. 1284–1293. IEEE, 2019.

Brandon Yang, Gabriel Bender, Quoc V. Le, and Jiquan Ngiam. Condconv: Conditionally parameterized convolutions for efficient inference. In *NeurIPS*, pp. 1305–1316, 2019.

Zhonghui You, Kun Yan, Jinmian Ye, Meng Ma, and Ping Wang. Gate decorator: Global filter pruning method for accelerating deep convolutional neural networks. In *NeurIPS*, pp. 2130–2141, 2019.

Kun Yuan, Quanquan Li, Jing Shao, and Junjie Yan. Learning connectivity of neural networks from a topological perspective. In *ECCV*. Springer, 2020.

Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *ECCV (1)*, volume 8689 of *Lecture Notes in Computer Science*, pp. 818–833. Springer, 2014.

Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, pp. 8697–8710. IEEE Computer Society, 2018.

6 APPENDIX

6.1 MODEL SETTINGS IN IMAGENET CLASSIFICATION

Setup for MobileNetV2. We train the networks for 200 epochs with a half-period-cosine shaped learning rate decay. The initial learning rate is 0.4 with a warmup phase of 5 epochs. The weight decay is set to 4e-5. To prevent overfitting, we use label smoothing (Szegedy et al., 2017) with a coefficient of 0.1 and dropout (Srivastava et al., 2014) before the last layer with rate of 0.2. For building DY-Graph, the *Inverted Bottleneck* block is represented as a node.

Setup for ResNets and ResNeXt. The networks are trained for 100 epochs with a half-period-cosine shaped learning rate decay. The initial learning rate is 0.4 with a warmup phase of 5 epochs. The weight decay is set to e-4. We use label smoothing with a coefficient of 0.1. Other details of the training procedure are the same as (Goyal et al., 2017). To form DY-Graph, the *BasicBlock* of ResNet18 and *Bottleneck* block of ResNet50/101 are denoted as nodes. For ResNeXt, the *Aggregated Bottleneck* block is set to be a node.

Setup for RegNets. The networks are trained for 100 epochs with a half-period-cosine shaped learning rate decay. The initial learning rate is 0.8 with a warmup phase of 5 epochs. The weight decay is set to 5e-5. We use label smoothing with a coefficient of 0.1. Other details are the same as (Radosavovic et al., 2020). For DY-Graph, the *Regular Bottleneck* is transformed to be a node.

6.2 LOCATION OF ROUTERS

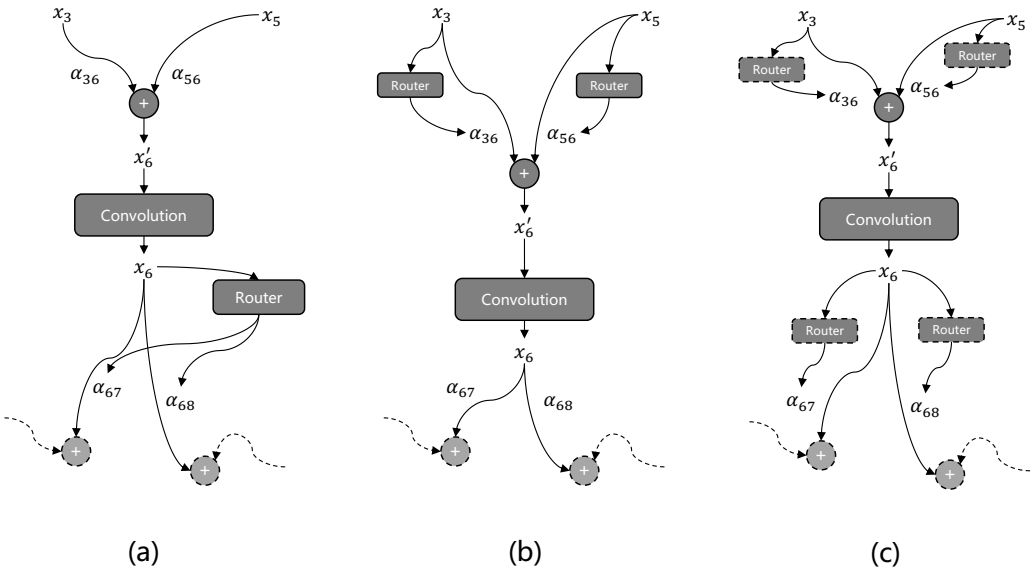

Figure 3: Different routing methods with different locations of routers.

In this paper, the connectivity is determined by the weights of edges which are predicted using extra conditional module routers. Within a node, there are two ways to obtain the weights, respectively predicting the weights of the output edges (as shown in Fig. 3 (a)) or predicting the weights of the input edges (as shown in Fig. 3 (b)). For the output type, as described in section 3.2, the router receives the transformed features as input and generates attention over output edges connected with posterior nodes. For the input type, the number of routers is related to the in-degree of the current node. Each router receives features from the connected preceding node and predicts weight for each input edge independently.

Although the form seems to be different, the two methods are equivalent under the routing function we designed. The router module consists of a global average pooling, a fully-connected layer and a

sigmoid activation. The router in Fig. 3 (a) can be split into the type of Fig. 3 (c). It can be noted as:

$$\boldsymbol{\pi}^i = \sigma(\mathbf{W}^T \times \boldsymbol{x}) = [\sigma(\mathbf{W}_{:,0}^T \times \boldsymbol{x}), \cdots, \sigma(\mathbf{W}_{:,j}^T \times \boldsymbol{x})] \tag{6}$$

where $\boldsymbol{x}$ is the feature vector after global average pooling, $\mathbf{W}$ is the weight of fully-connected layer, $\mathbf{W}_{:,j}$ is the weight of independent fully-connected layer after splitting, and $(\times)$ denotes the matrix multiplication. The prediction for the output edge of the current node is equal to the prediction for the input edge of the next node. Therefore, the two methods are equivalent. To simplify, we select the first type in implementation.

6.3 Ablation Study

We conduct ablation study on different connectivity methods to reflect the effectiveness of the proposed DY-Graph. The experiments are performed in ImageNet and follow the training setting in section 4.1. For a fair comparison, we select ResNet-50/101 as the backbone structure. The symbol α denotes assigning learnable parameters to the edge directly, which learns fixed connectivity for all samples. The symbol α_b denotes the type of DY-Graph, which learns instance-aware connectivity. The experimental results are given in Table 6. In this way, ResNet-50 with α_b still outperforms one with α by 1.00% in top-1 accuracy. And ResNet-101 is the same. This demonstrates that due to the enlarged optimization space, dynamic connectivity is better than static connectivity in these networks.

Table 6: Ablation study on different connectivity methods. Experiments are conducted in ImageNet and Top-1/5 accuracies are reported. And the numbers in subscript represent incremental improvements.

Backbone	α	α_b	Top-1	Top-5
ResNet-50			76.70	93.39
	✓		$77.00_{+0.30}$	93.39
		✓	$78.00_{+1.00}$	94.04
ResNet-101			78.29	94.09
	✓		$78.64_{+0.35}$	94.32
		✓	$79.68_{+1.04}$	94.73