NAS-FPN: Learning Scalable Feature Pyramid Architecture for Object Detection

Golnaz Ghiasi Tsung-Yi Lin Quoc V. Le
Google Brain

Abstract

Current state-of-the-art convolutional architectures for object detection are manually designed. Here we aim to learn a better architecture of feature pyramid network for object detection. We adopt Neural Architecture Search and discover a new feature pyramid architecture in a novel scalable search space covering all cross-scale connections. The discovered architecture, named NAS-FPN, consists of a combination of top-down and bottom-up connections to fuse features across scales. NAS-FPN, combined with various backbone models in the RetinaNet framework, achieves better accuracy and latency tradeoff compared to state-of-the-art object detection models. NAS-FPN improves mobile detection accuracy by 2 AP compared to state-of-the-art SSDLite with MobileNetV2 model in [32] and achieves 48.3 AP which surpasses Mask R-CNN [10] detection accuracy with less computation time.

1. Introduction

Learning visual feature representations is a fundamental problem in computer vision. In the past few years, great progress has been made on designing the model architecture of deep convolutional networks (ConvNets) for image classification [12, 15, 35] and object detection [21, 22]. Unlike image classification which predicts class probability for an image, object detection has its own challenge to detect and localize multiple objects across a wide range of scales and locations. To address this issue, the pyramidal feature representations, which represent an image with multiscale feature layers, are commonly used by many modern object detectors [11, 23, 26].

Feature Pyramid Network (FPN) [22] is one of the representative model architectures to generate pyramidal feature representations for object detection. It adopts a backbone model, typically designed for image classification, and builds feature pyramid by sequentially combining two adjacent layers in feature hierarchy in backbone model with top-down and lateral connections. The high-level features, which are semantically strong but lower resolution, are up-

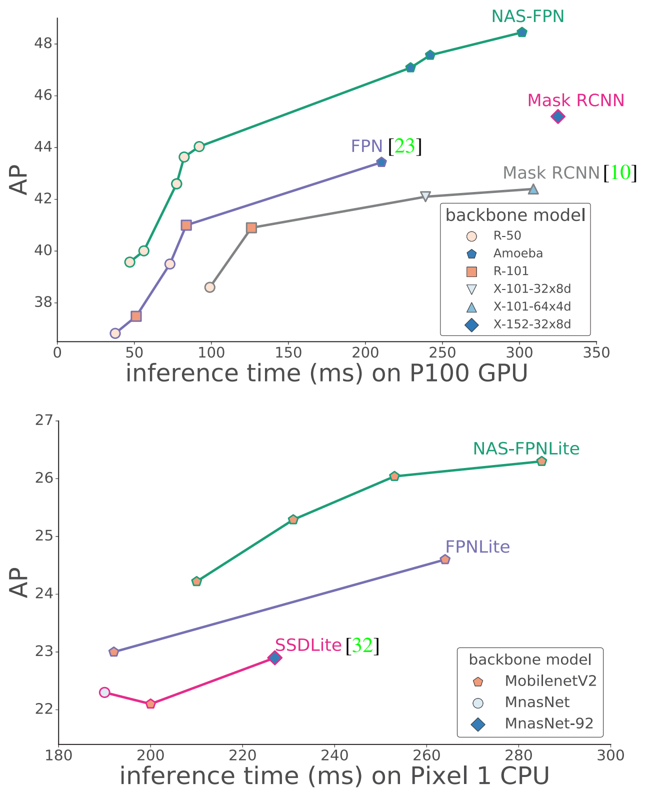


Figure 1: Average Precision vs. inference time per image across accurate models (top) and fast models (bottom) on mobile device. The green curve highlights results of NAS-FPN combined with RetinaNet. Please refer to Figure 9 for details.

sampled and combined with higher resolution features to generate feature representations that are both high resolution and semantically strong. Although FPN is simple and effective, it may not be the optimal architecture design. Recently, PANet [25] shows adding an extra bottom-up pathway on FPN features improves feature representations for lower resolution features. Many recent works [7, 16, 17, 34, 38, 39, 40, 43, 41] propose various cross-scale connections or operations to combine features to generate pyramidal feature representations.

The challenge of designing feature pyramid architecture is in its huge design space. The number of possible connections to combine features from different scales grow exponentially with the number of layers. Recently, Neural Architecture Search algorithm [44] demonstrates promising results on efficiently discovering top-performing architectures for image classification in a huge search space. To achieve their results, Zoph et al. [45] propose a modularized architecture that can be repeated and stacked into a scalable architecture. Inspired by [45], we propose the search space of scalable architecture that generates pyramidal representations. The key contribution of our work is in designing the search space that covers all possible *cross-scale* connections to generate multiscale feature representations. During the search, we aims to discover an atomic architecture that has identical input and output feature levels and can be applied repeatedly. The modular search space makes searching pyramidal architectures manageable. Another benefit of modular pyramidal architecture is the ability for anytime object detection (or "early exit"). Although such early exit approach has been attempted [14], manually designing such architecture with this constraint in mind is quite difficult.

The discovered architecture, named NAS-FPN, offers great flexibility in building object detection architecture. NAS-FPN works well with various backbone model, such as MobileNet [32], ResNet [12], and AmoebaNet [29]. It offers better tradeoff of speed and accuracy for both fast mobile model and accurate model. Combined with MobileNetV2 backbone in RetinaNet framework, it outperforms state-of-the-art mobile detection model of SSDLite with MobilenetV2 [32] by 2 AP given the same inference time. With strong AmoebaNet-D backbone model, NAS-FPN achieves 48.3 AP single model accuracy with single testing scale. The detection accuracy surpasses Mask R-CNN reported in [10] with even less inference time. A summary of our results is shown in Figure 1.

2. Related Works

2.1. Architecture for Pyramidal Representations

Feature pyramid representations are the basis of solutions for many computer vision applications required multiscale processing [1]. However, using Deep ConvNets to generate pyramidal representations by featurizing image pyramid imposes large computation burden. To address this issue, recent works on human pose estimation, image segmentation, and object detection [8, 11, 22, 28, 31] introduce cross-scale connections in ConvNets that connect internal feature layers in different scales. Such connections effectively enhance feature representations such that they are not only semantically strong but also contain high resolution information. Many works have studied how to improve mutliscale feature presentations. Liu et.al [25] propose an additional bottom-up pathway based on FPN [22]. Recently, Zhao et al. [42] extends the idea to build stronger feature pyramid representations by employing multiple U-shape modules after a backbone model. Kong et al. [16] first combine features at all scales and generate features at each scale by a global attention operation on the combined features. Despite it is an active research area, most architecture designs of cross-scale connections remain shallow compared to the backbone model. In addition to manually design the cross-scale connections, [5, 27] propose to learn the connections through gating mechanism for visual counting and dense label predictions.

In our work, instead of manually designing architectures for pyramidal representations, we use a combination of scalable search space and Neural Architecture Search algorithm to overcome the large search space of pyramidal architectures. We constrain the search to find an architecture that can be applied repeatedly. The architecture can therefore be used for anytime object detection (or "early exit"). Such early exit idea is related to [3, 37], especially in image classification [14].

2.2. Neural Architecture Search

Our work is closely related to the work on Neural Architecture Search [44, 2, 45, 29]. Most notably, Zoph et al. [45] use a reinforcement learning with a controller RNN to design a cell (or a layer) to obtain a network, called NASNet which achieves state-of-the-art accuracy on ImageNet. The efficiency of the search process is further improved by [24] to design a network called PNASNet, with similar accuracy to NASNet. Similarly, an evolution method [29] has also been used to design AmoebaNets that improve upon NASNet and PNASNet. Since reinforcement learning and evolution controllers perform similarly well, we only experiment with a Reinforcement Learning controller in this paper. Our method has two major differences compared to [44]: (1) the outputs of our method are multiscale features whereas output of [44] is single scale features for classification; (2) our method specifically searches cross-scale connections, while [44] only focuses on discovering connections within the same feature resolution. Beyond image classification, Neural Architecture Search has also been used to improve image segmentation networks [4]. To the best of our knowledge, our work is the first to report success of applying Neural Architecture Search for pyramidal architecture in object detection. For a broader overview of related methods for Neural Architecture Search, please see [6].

3. Method

Our method is based on the RetinaNet framework [23] because it is simple and efficient. The RetinaNet framework has two main components: a backbone network (often state-of-the-art image classification network) and a feature pyramid network (FPN). The goal of the proposed algorithm is to discover a better FPN architecture for RetinaNet. Figure 2 shows the RetinaNet architecture.

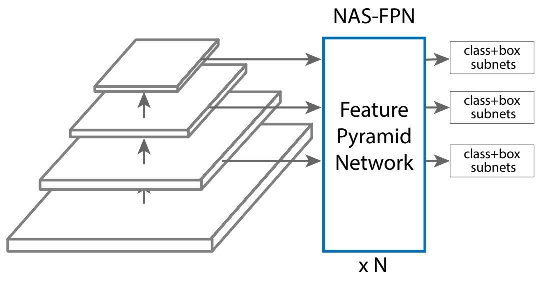


Figure 2: RetinaNet with NAS-FPN. In our proposal, feature pyramid network is to be searched by a neural architecture search algorithm. The backbone model and the subnets for class and box predictions follow the original design in RetinaNet [23]. The architecture of FPN can be stacked N times for better accuracy.

To discover a better FPN, we make use of the Neural Architecture Search framework proposed by [44]. The Neural Architecture Search trains a controller to select best model architectures in a given search space using reinforcement learning. The controller uses the accuracy of a child model in the search space as the reward signal to update its parameters. Thus through trial and error the controller learns to generate better architectures over time. As it has been identified by previous works [36, 44, 45], the search space plays a crucial role in the success of architecture search.

In the next section, we design a search space for FPN to generate feature pyramid representations. For scalability of the FPN (i.e., so that an FPN architecture can be stacked repeatedly within RetinaNet), during the search, we also force the the FPN to repeat itself N times and then concatenated into a large architecture. We call our feature pyramid architecture NAS-FPN.

3.1. Architecture Search Space

In our search space, a feature pyramid network consists a number of "merging cells" that combine a number of input layers into representations for RetinaNet. In the following, we will describe the inputs into the Feature Pyramid Network, and how each merging cell is constructed.

Feature Pyramid Network. A feature pyramid network takes multiscale feature layers as inputs and generate output feature layers in the identical scales as shown in Figure 2. We follow the design by RetinaNet [23] which uses the last layer in each group of feature layers as the inputs to the first pyramid network. The output of the first pyramid network are the input to the next pyramid network. We use as inputs features in 5 scales $\{C_3, C_4, C_5, C_6, C_7\}$ with corresponding feature stride of $\{8, 16, 32, 64, 128\}$ pixels. The C_6 and C_7 are created by simply applying stride 2 and stride 4 max pooling to C_5. The input features are then passed to a pyramid network consisting of a series of *merging cells* (see below) that introduce cross-scale connections. The pyramid network then outputs augmented multiscale feature representations $\{P_3, P_4, P_5, P_6, P_7\}$. Since both inputs and outputs of a pyramid network are feature layers in the identical scales, the architecture of the FPN can be *stacked* repeatedly for better accuracy. In Section 4, we show controlling the number of pyramid networks is one simple way to tradeoff detection speed and accuracy.

Merging cell. An important observation in previous works in object detection is that it is necessary to "merge" features at different scales. The cross-scale connections allow model to combine high-level features with strong semantics and low-level features with high resolution.

We propose *merging cell*, which is a fundamental building block of a FPN, to merge any two input feature layers into a output feature layer. In our implementation, each merging cell takes two input feature layers (could be from different scales), applies processing operations and then combines them to produce one output feature layer of a desired scale. A FPN consists of N different merging cells, where N is given during search. In a merging cell, all feature layers have the same number of filters. The process of constructing a merging cell is shown in Figure 3.

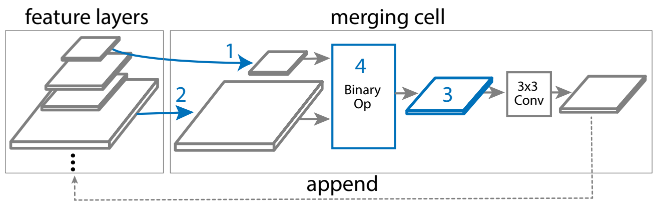


Figure 3: Four prediction steps required in a merging cell. Note the output feature layer is pushed back into the stack of candidate feature layers and available for selection for the next merging cell.

The decisions of how to construct the merging cell are made by a controller RNN. The RNN controller selects any two candidate feature layers and a binary operation to combine them into a new feature layer, where all feature layers may have different resolution. Each merging cell has 4 prediction steps made by distinct softmax classifiers:

Step 1. Select a feature layer h_i from candidates.

Step 2. Select another feature layer h_j from candidates without replacement.

Step 3. Select the output feature resolution.

Step 4. Select a binary op to combine h_i and h_j selected in Step 1 and Step 2 and generate a feature layer with the resolution selected in Step 3.

In step 4, we design two binary operations, *sum* and *global pooling*, in our search space as shown in Figure 4. These two operations are chosen for their simplicity and efficiency. They do not add any extra trainable parameters. The sum operation is commonly used for combining features [22]. The design of global pooling operation is inspired by [20]. We follow Pyramid Attention Networks [20] except removing convolution layers in the original design. The input feature layers are adjusted to the output resolution by nearest neighbor upsampling or max pooling if needed before applying the binary operation. The merged feature layer is always followed by a ReLU, a 3x3 convolution, and a batch normalization layer.

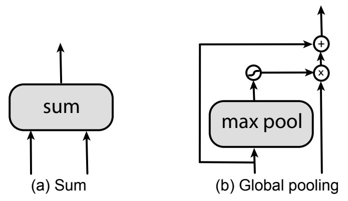


Figure 4: Binary operations.

The input feature layers to a pyramid network form the initial list of input candidates of a merging cell. In Step 5, the newly-generated feature layer is appended to the list of existing input candidates and becomes a new candidate for the next merging cell. There can be multiple candidate features share the same resolution during architecture search. To reduce computation in discovered architecture, we avoid selecting stride 8 feature in Step 3 for intermediate merging cells. In the end, the last 5 merging cells are designed to outputs feature pyramid $\{P_3, P_4, P_5, P_6, P_7\}$. The order of output feature levels is predicted by the controller. Each output feature layer is then generated by repeating the step 1, 2, 4 until the output feature pyramid is fully generated. Similar to [44], we take all feature layers that have not been connected to any of output layer and sum them to the output layer that has the corresponding resolution.

3.2. Deeply supervised Anytime Object Detection

One advantage of scaling NAS-FPN with stacked pyramid networks is that the feature pyramid representations can be obtained at output of any given pyramid network. This property enables *anytime detection* which can generate detection results with early exit. Inspired by [19, 13], we can attach classifier and box regression heads after all intermediate pyramid networks and train it with deep supervision [19]. During inference, the model does not need to finish the forward pass for all pyramid networks. Instead, it can stop at the output of any pyramid network and generate detection results. This can be a desirable property when computation resource or latency is a concern and provides a solution that can dynamically decide how much computation resource to allocate for generating detections. In Appendix A, we show NAS-FPN can be used for anytime detection.

4. Experiments

In this section, we first describe our experiments of Neural Architecture Search to learn a RNN controller to discover the NAS-FPN architecture. Then we demonstrate the discovered NAS-FPN works well with different backbone models and image sizes. The capacity of NAS-FPN can be easily adjusted by changing the number of stacking layers and the feature dimension in pyramid network. We show how to build accurate and fast architectures in the experiments.

4.1. Implementation Details

We use the open-source implementation of RetinaNet[1] for experiments. The models are trained on TPUs with 64 images in a batch. During training, we apply multiscale training with a random scale between [0.8, 1.2] to the output image size. The batch normalization layers are applied after all convolution layers. We use $\alpha = 0.25$ and $\gamma = 1.5$ for focal loss. We use a weight decay of 0.0001 and a momentum of 0.9. The model is trained using 50 epochs. The initial learning rate 0.08 is applied for first 30 epochs and decayed 0.1 at 30 and 40 epochs. For experiments with DropBlock [9], we use a longer training schedule of 150 epochs with first decay at 120 and the second decay at 140 epochs. The step-wise learning rate schedule was not stable for training our model with AmoebaNet backbone on image size of 1280x1280 and for this case we use cosine learning rate schedule. The model is trained on COCO train2017 and evaluated on COCO val2017 for most experiments. In Table 1, we report test-dev accuracy to compare with existing methods.

4.2. Architecture Search for NAS-FPN

Proxy task. To speed up the training of the RNN controller we need a proxy task [45] that has a short training time and also correlates with the real task. The proxy task can then be used during the search to identify a good FPN architecture. We find that we can simply shorten the training of target task and use it as the proxy task. We only train the proxy task for 10 epochs, instead of 50 epochs that we use to train RetinaNet to converge. To further speed up training proxy task, we use a small backbone architecture of

[1]https://github.com/tensorflow/tpu/tree/master/models/official/retinanet

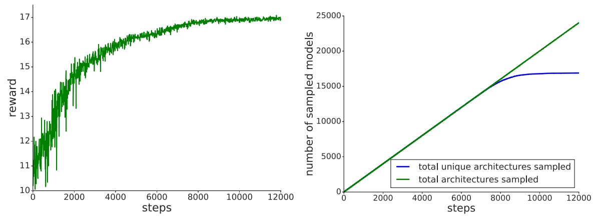


Figure 5: Left: Rewards over RL training. The reward is computed as the AP of sampled architectures on the proxy task. Right: The number of sampled unique architectures to the total number of sampled architectures. As controller converges, more identical architectures are sampled by the controller.

ResNet-10 with input 512 $\times$ 512 image size. With these reductions, the training time is 1hr for a proxy task on TPUs. We repeat the pyramid networks 3 times in our proxy task. The initial learning rate 0.08 is applied for first 8 epochs and decayed by the factor of 0.1 at epoch 8. We reserve a randomly selected 7392 images from the COCO train2017 set as the validation set, which we use to obtain rewards.

Controller. Similar to [44] our controller is a recurrent neural network (RNN) and it is trained using the Proximal Policy Optimization (PPO) [33] algorithm. The controller samples child networks with different architectures. These architectures are trained on a proxy task using a pool of workers. The workqueue in our experiments consisted of 100 Tensor Processing Units (TPUs). The resulting detection accuracy in average precision (AP) on a held-out validation set is used as the reward to update the controller. Figure 5-Left shows the AP of the sampled architectures for different iterations of training. As it can be seen the controller generated better architectures over time. Figure 5-Right shows total number of sampled architectures and also the total number of unique architectures generated by the RNN controller. The number of unique architectures converged after about 8000 steps. We use the architecture with the highest AP from all sampled architectures during RL training in our experiments. This architecture is first sampled at 8000 step and sampled many times after that. Figure 6 shows the details of this architecture.

Discovered feature pyramid architectures. What makes a good feature pyramid architecture? We hope to shed lights on this question by visualizing the discovered architectures. In Figure 7(b-f), we plot NAS-FPN architectures with progressively higher reward during RL training. We find the RNN controller can quickly pick up some important cross-scale connections in the early learning stage. For example, it discovers the connection between high resolution input and output feature layers, which is critical to generate high resolution features for detecting small objects. As the controller converges, the controller discovers architectures that have both top-down and bottom-up connections which is different from vanilla FPN in Figure 7(a). We also find better feature reuse as the controller converges. Instead of randomly picking any two input layers from the candidate pool, the controller learns to build connections on newly-generated layers to reuse previously computed feature representations.

4.3. Scalable Feature Pyramid Architecture

In this section, we show how to control the model capacity by adjusting (1) backbone model, (2) the number of repeated pyramid networks, and (3) the number of dimension in pyramid network. We discuss how these adjustments tradeoff computational time and speed. We define a simple notation to indicate backbone model and NAS-FPN capacity. For example, R-50, 5 @ 256 indicate a model using ResNet-50 backbone model, 5 stacked NAS-FPN pyramid networks, and 256 feature dimension.

Stacking pyramid networks. Our pyramid network has a nice property that it can be scaled into a larger architecture by stacking multiple repeated architectures. In Figure 8a, we show that stacking the vanilla FPN architecture does not always improve performance whereas stacking NAS-FPN improves accuracy significantly. This result highlights our search algorithm can find scalable architectures, which may be hard to design manually. Interestingly, although we only apply 3 pyramid networks for the proxy task during the architecture search phase, the performance still improves with up to 7 pyramid networks applied.

Adopting different backbone architectures. One common way to tradeoff accuracy and speed for object detection architectures is altering the backbone architecture. Although the pyramid network in NAS-FPN was discovered by using a light-weight ResNet-10 backbone architecture, we show that it can be transferred well across different backbone architectures. Figure 8b shows the performance of NAS-FPN on top of different backbones, from a lighter weight architecture such as MobilenetV2 to a very high capacity architecture such as AmoebaNet-D [29]. When we apply NAS-FPN with MobilenetV2 on the image size of 640 $\times$ 640, we get 36.6 AP with $160B$ FLOPs. Using state-of-the-art image classification architecture of AmoebaNet-D [29] as the backbone increases the FLOPs to $390B$ but also adds about 5 AP. NAS-FPN with both light and heavy backbone architectures benefits from stacking more pyramid networks.

Adjusting feature dimension of feature pyramid networks. Another way to increase the capacity of a model is

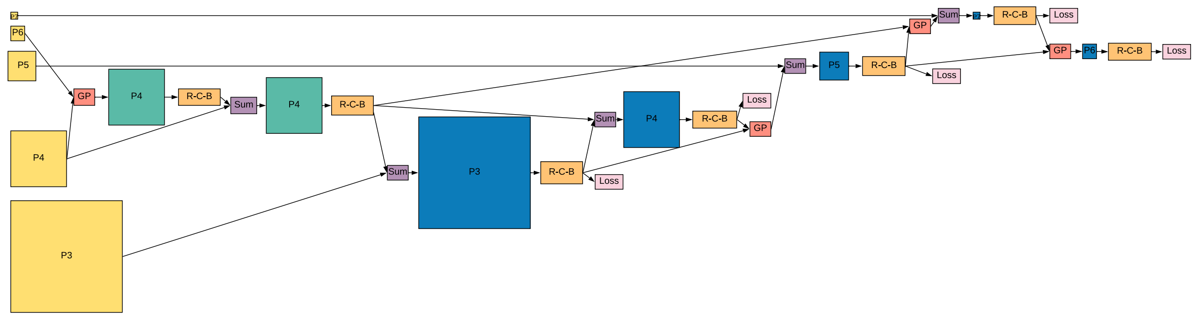


Figure 6: Architecture of the discovered 7-merging-cell pyramid network in NAS-FPN with 5 input layers (yellow) and 5 output feature layers (blue). GP and R-C-B are stands for Global Pooling and ReLU-Conv-BatchNorm, respectively.

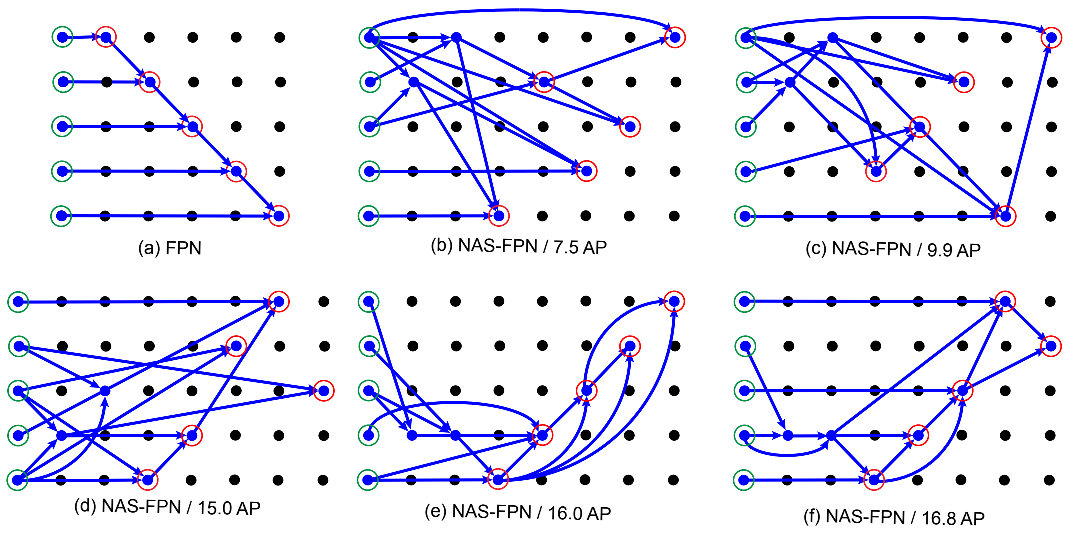
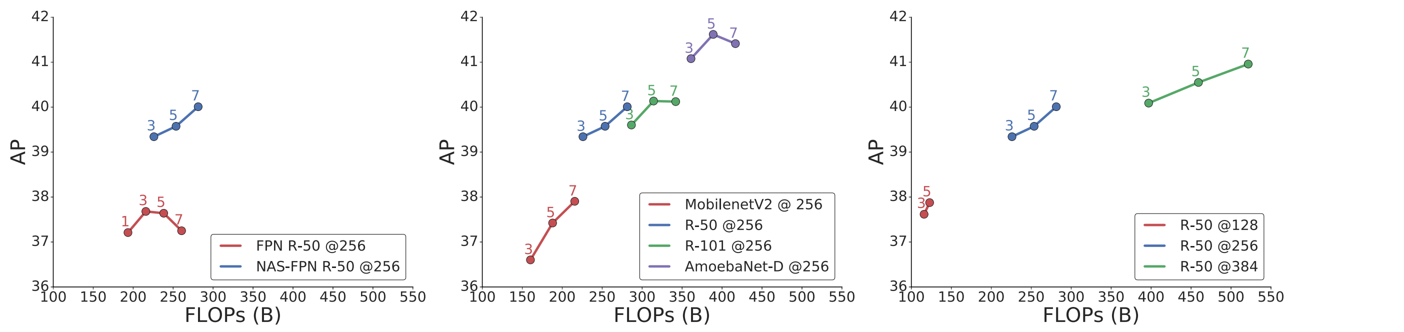


Figure 7: Architecture graph of NAS-FPN. Each dot represents a feature layer. Feature layers in the same row have identical resolution. The resolution decreases in the bottom-up direction. The arrows indicate the connections between internal layers. The graph is constructed such that an input layer is on the left side. The inputs to a pyramid network are marked with green circles and outputs are marked with red circles. (a) The baseline FPN architecture. (b-f) The 7-cell NAS-FPN architectures discovered by Neural Architecture Search over training of the RNN controller. The discovered architectures converged as the reward (AP) of the proxy task progressively improves. (f) The final NAS-FPN that we used in our experiments.

to increase the feature dimension of feature layers in NAS-FPN. Figure 8c shows results of 128, 256, and 384 feature dimension in NAS-FPN with a ResNet-50 backbone architecture. Not surprisingly, increasing the feature dimension improves detection performance but it may not be an efficient way to improve the performance. In Figure 8c, R-50 7 @ 256, with much less FLOPs, achieves similar AP compared to R-50 3 @ 384. Increasing feature dimension would require model regularization technique. In Section 4.4, we discuss using DropBlock [9] to regularize the model.

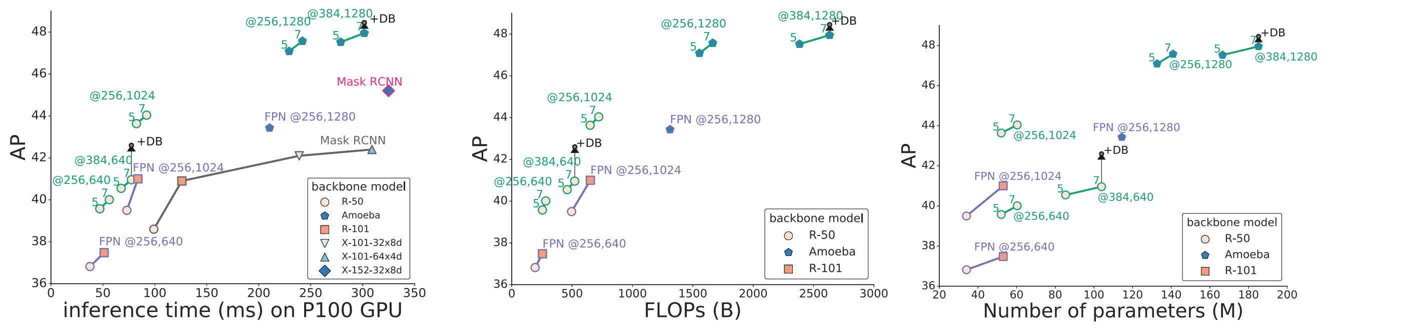
Architectures for high detection accuracy. With the scalable NAS-FPN architecture, we discuss how to build an accurate model while remaining efficient. In Figure 9a, we first show that NAS-FPN R-50 5 @256 model has comparable FLOPs to the R-101 FPN baseline but with 2.5 AP gain. This shows using NA S-FPN is more effective than replacing the backbone with a higher capacity model. Going for a higher accuracy model, one can use a heavier backbone model or higher feature dimensions. Figure 9a shows that NAS-FPN architectures are in the upper left part in the accuracy to inference time figure compared to existing methods. The NAS-FPN is as accurate as to the state-of-the-art Mask R-CNN model with less computation time.

Architectures for fast inference. Designing object detector with low latency and limited computation budget is an active research topic. Here, we introduce NAS-FPNLite

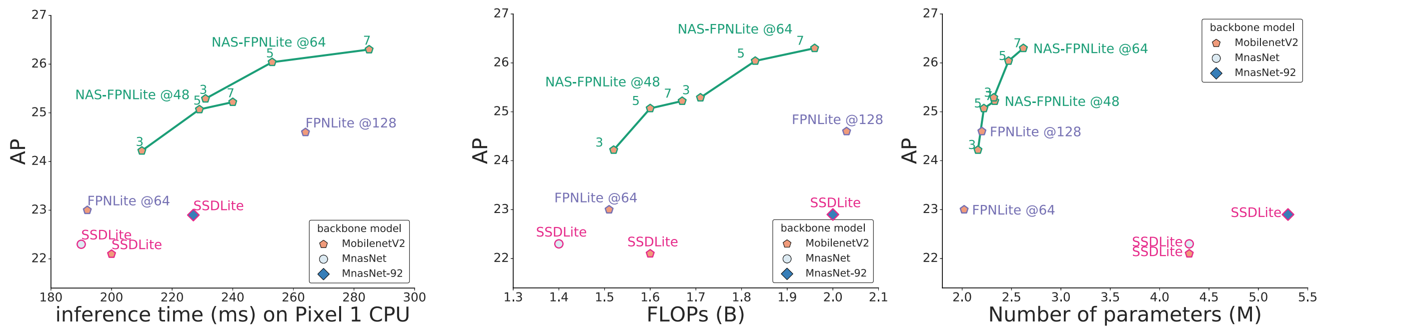


(a) Number of pyramid networks (b) Backbone architectures (c) Feature dimension

Figure 8: The model capacity of NAS-FPN can be controlled with (a) stacking pyramid networks, (b) changing the backbone architecture, and (c) increasing feature dimension in pyramid networks. All models are trained/tested on the image size of 640x640. Number above the marker indicates number of pyramid networks in NAS-FPN.



(a) Accurate models



(b) Fast models

Figure 9: Detection accuracy to inference time (left), FLOPs (middle), and parameters (right). (a) We compare to other high accuracy models. The inference time of all models are computed on a machine with P100 GPU. The green curves highlights results for NAS-FPN with different backbone architectures. The number above the marker indicates the number of repeats of pyramid networks in NAS-FPN. The feature dimension of NAS-FPN/FPN and input image size are mentioned next to each data point. (b) We compare to other fast models. The input image size of all models is 320x320 and the inference times are computed on Pixel 1 CPU. Our model are trained with light-weight model of MobileNetV2.

for mobile object detection. The major difference of NAS-FPNLite and NAS-FPN is that we search a pyramid network that has outputs from P_3 to P_6. Also we follow SSDLite [32] and replace convolution with depth-wise separable convolution in NAS-FPN. We discover a 15-cell architecture which yields good performance and use it in our experiments. We combine NAS-FPNLite and MobileNetV2 [32] in RetinaNet framework. For a fair comparison, we create a FPNLite baseline, which follows the original FPN structure and replaces all convolution layers

model	image size	# FLOPs	# params	inference time (ms)	test-dev AP
YOLOv3 DarkNet-53 [30]	320 × 320	38.97 B	-	22 (Titan X)	28.2
MobileNetV2 + SSDLite [36]	320 × 320	1.6B	4.3M	200 (Pixel 1 CPU)	22.1
MnasNet + SSDLite [36]	320 × 320	1.4B	4.3M	190 (Pixel 1 CPU)	22.3
MnasNet-92 + SSDLite [36]	320 × 320	2.0B	5.3M	227 (Pixel 1 CPU)	22.9
FPNLite MobileNetV2 @ 64	320 × 320	1.51B	2.02M	192 (Pixel 1 CPU)	22.7
FPNLite MobileNetV2 @ 128	320 × 320	2.03B	2.20M	264 (Pixel 1 CPU)	24.3
NAS-FPNLite MobileNetV2 (3 @ 48)	320 × 320	1.52 B	2.16 M	210 (Pixel 1 CPU)	24.2
NAS-FPNLite MobileNetV2 (7 @ 64)	320 × 320	1.96 B	2.62 M	285 (Pixel 1 CPU)	25.7
YOLOv3 DarkNet-53 [30]	608 × 608	140.69 B	-	51 (Titan X)	33.0
CornerNet Hourglass [18]	512 × 512	-	-	244 (Titan X)	40.5
Mask R-CNN X-152-32x8d [11]	1280 × 800	-	-	325 (P100)	45.2
RefineDet R-101 [41]	832 × 500	-	-	90 (Titan X)	34.4
FPN R-50 @256 [23]	640 × 640	193.6B	34.0M	37.5 (P100)	37.0
FPN R-101 @256 [23]	640 × 640	254.2B	53.0M	51.1 (P100)	37.8
FPN R-50 @256 [23]	1024 × 1024	495.8B	34.0M	73.0 (P100)	40.1
FPN R-101 @256 [23]	1024 × 1024	651.1B	53.0M	83.7 (P100)	41.1
FPN AmoebaNet @256 [23]	1280 × 1280	1311 B	114.4 M	210.4 (P100)	43.4
NAS-FPN R-50 (7 @ 256)	640 × 640	281.3B	60.3M	56.1 (P100)	39.9
NAS-FPN R-50 (7 @ 256)	1024 × 1024	720.4B	60.3M	92.1 (P100)	44.2
NAS-FPN R-50 (7 @ 256)	1280 × 1280	1125.5B	60.3M	131.9 (P100)	44.8
NAS-FPN R-50 (7 @ 384)	1280 × 1280	2086.3B	103.9 M	192.3 (P100)	45.4
NAS-FPN R-50 (7 @ 384) + DropBlock	1280 × 1280	2086.3B	103.9M	192.3 (P100)	46.6
NAS-FPN AmoebaNet (7 @ 384)	1280 × 1280	2633 B	166.5 M	278.9 (P100)	48.0
NAS-FPN AmoebaNet (7 @ 384) + DropBlock	1280 × 1280	2633 B	166.5 M	278.9 (P100)	48.3

Table 1: Performance of RetinaNet with NAS-FPN and other state-of-the-art detectors on test-dev set of COCO.

with depth-wise separable convolution. Following [36, 32], we train NAS-FPNLite and FPNLite using an open-source object detection API.[2] In Figure 9b, we control the feature dimension of NAS-FPN to be 48 or 64 so that it has similar FLOPs and CPU runtime on Pixel 1 as baseline methods and show that NAS-FPNLite outperforms both SSDLite [32] and FPNLite.

4.4. Further Improvements with DropBlock

Due to the increased number of new layers introduced in NAS-FPN architecture, a proper model regularization is needed to prevent overfitting. Following the technique in [9], we apply DropBlock with block size 3x3 after batch normalization layers in the the NAS-FPN layers. Figure 10 shows DropBlock improves the performance of NAS-FPN. Especially, it improves more for architecture that has more newly introduced filters. Note that by default we do not apply DropBlock in previous experiments for the fair comparison to existing works.

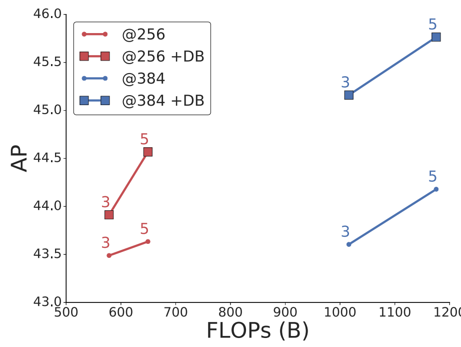


Figure 10: Performance comparison of NAS-FPN with feature dimension of 256 or 384 when it is trained with and without DropBlock (DB). Models are trained with backbone of ResNet-50 on image size of 1024x1024. Adding DropBlock is more important when we increase feature dimension in pyramid networks.

5. Conclusion

In this paper, we proposed to use Neural Architecture Search to further optimize the process of designing Feature Pyramid Networks for Object Detection. Our experiments on the COCO dataset showed that the discovered architecture, named NAS-FPN, is flexible and performant for building accurate detection model. On a wide range of accuracy and speed tradeoff, NAS-FPN produces significant improvements upon many backbone architectures.

[2]https://github.com/tensorflow/models/tree/master/research/object_detection

References

[1] E. H. Adelson, C. H. Anderson, J. R. Bergen, P. J. Burt, and J. M. Ogden. Pyramid methods in image processinh. *RCA engineer*, 1984. 2

[2] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. In *ICLR*, 2016. 2

[3] T. Bolukbasi, J. Wang, O. Dekel, and V. Saligrama. Adaptive neural networks for efficient inference. In *ICML*, 2017. 2

[4] L.-C. Chen, M. D. Collins, Y. Zhu, G. Papandreou, B. Zoph, F. Schroff, H. Adam, and J. Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *NIPS*, 2018. 2

[5] R. J. L.-S. D. Ooro-Rubio, M. Niepert. Learning short-cut connections for object counting. *BMVC*, 2018. 2

[6] T. Elsken, J. H. Metzen, and F. Hutter. Neural architecture search: A survey. *arXiv preprint arXiv:1808.05377*, 2018. 2

[7] C. Fu, W. Liu, A. Ranga, A. Tyagi, and A. C. Berg. DSSD : Deconvolutional single shot detector. *CoRR*, abs/1701.06659, 2017. 1

[8] G. Ghiasi and C. C. Fowlkes. Laplacian pyramid reconstruction and refinement for semantic segmentation. In *ECCV*, 2016. 2

[9] G. Ghiasi, T. Lin, and Q. V. Le. DropBlock: A regularization method for convolutional networks. *NIPS*, 2018. 4, 6, 8

[10] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, and K. He. Detectron. `https://github.com/facebookresearch/detectron`, 2018. 1, 2

[11] K. He, G. Gkioxari, P. Dollár, and R. Girshick. Mask R-CNN. In *ICCV*, 2017. 1, 2, 8

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, 2016. 1, 2

[13] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Weinberger. Multi-scale dense networks for resource efficient image classification. In *ICLR*, 2018. 4

[14] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger. Multi-scale dense networks for resource efficient image classification. In *ICLR*, 2017. 2

[15] G. Huang, Z. Liu, and K. Q. Weinberger. Densely connected convolutional networks. In *CVPR*, 2017. 1

[16] T. Kong, F. Sun, W. Huang, and H. Liu. Deep feature pyramid reconfiguration for object detection. In *ECCV*, 2018. 1, 2

[17] T. Kong, F. Sun, A. Yao, H. Liu, M. Lu, and Y. Chen. RON: reverse connection with objectness prior networks for object detection. In *CVPR*, 2017. 1

[18] H. Law and J. Deng. Cornernet: Detecting objects as paired keypoints. In *ECCV*, 2018. 8

[19] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply-supervised nets. In *AISTATS*, 2015. 4

[20] H. Li, P. Xiong, J. An, and L. Wang. Pyramid attention network for semantic segmentation. *BMVC*, 2018. 4

[21] Z. Li, C. Peng, G. Yu, X. Zhang, Y. Deng, and J. Sun. Detnet: A backbone network for object detection. In *ECCV*, 2018. 1

[22] T.-Y. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie. Feature pyramid networks for object detection. In *CVPR*, 2017. 1, 2, 4

[23] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár. Focal loss for dense object detection. In *ICCV*, 2017. 1, 2, 3, 8

[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. In *ECCV*, 2017. 2

[25] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia. Path aggregation network for instance segmentation. In *CVPR*, 2018. 1, 2

[26] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg. SSD: single shot multibox detector. In *ECCV*, 2016. 1

[27] N. D. B. B. Md Amirul Islam, Mrigank Rochan and Y. Wang. Gated feedback refinement network for dense image labeling. *CVPR*, 2017. 2

[28] A. Newell, K. Yang, and J. Deng. Stacked hourglass networks for human pose estimation. In *ECCV*, 2016. 2

[29] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le. Regularized evolution for image classifier architecture search. In *AAAI*, 2018. 2, 5

[30] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018. 8

[31] O. Ronneberger, P. Fischer, and T. Brox. U-Net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention*, 2015. 2

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen. MobileNetV2: inverted residuals and linear bottl. *CVPR*, 2019. 1, 2, 7, 8

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 5

[34] J.-Y. S. M.-C. K. S.-J. K. Seung-Wook Kim, Hyong-Keun Kook. Parallel feature pyramid network for object detection. *ECCV*, 2018. 1

[35] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Deep residual learning for image recognition. In *CVPR*, 2015. 1

[36] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *arXiv preprint arXiv:1807.11626*, 2018. 3, 8

[37] S. Teerapittayanon, B. McDanel, and H. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *ICPR*, pages 2464–2469. IEEE, 2016. 2

[38] S. Woo, S. Hwang, and I. S. Kweon. StairNet: top-down semantic aggregation for accurate one shot detection. In *WACV*, 2018. 1

[39] D. K. Yonghyun Kim, Bong-Nam Kang. San: Learning relationship between convolutional features for multi-scale object detection. *ECCV*, 2018. 1

[40] F. Yu, D. Wang, E. Shelhamer, and T. Darrell. Deep layer aggregation. In *CVPR*, 2018. 1

[41] S. Zhang, L. Wen, X. Bian, Z. Lei, and S. Z. Li. Single-shot refinement neural network for object detection. In *CVPR*, 2018. 1, 8

[42] Q. Zhao, T. Sheng, Y. Wang, Z. Tang, Y. Chen, L. Cai, and H. Ling. M2det: A single-shot object detector based on multi-level feature pyramid network. *AAAI*, 2019. 2

[43] P. Zhou, B. Ni, C. Geng, J. Hu, and Y. Xu. Scale-transferrable object detection. In *CVPR*, 2018. 1

[44] B. Zoph and Q. V. Le. Neural architecture search with reinforcement learning. In *ICLR*, 2017. 2, 3, 4, 5

[45] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le. Learning transferable architectures for scalable image recognition. In *CVPR*, 2018. 2, 3, 4