# Pyramidal Convolution: Rethinking Convolutional Neural Networks for Visual Recognition

**Ionut Cosmin Duta, Li Liu, Fan Zhu, Ling Shao**
Inception Institute of Artificial Intelligence (IIAI)
{icduta}@gmail.com

## Abstract

This work introduces pyramidal convolution (PyConv), which is capable of processing the input at multiple filter scales. PyConv contains a pyramid of kernels, where each level involves different types of filters with varying size and depth, which are able to capture different levels of details in the scene. On top of these improved recognition capabilities, PyConv is also efficient and, with our formulation, it does not increase the computational cost and parameters compared to standard convolution. Moreover, it is very flexible and extensible, providing a large space of potential network architectures for different applications. PyConv has the potential to impact nearly every computer vision task and, in this work, we present different architectures based on PyConv for four main tasks on visual recognition: image classification, video action classification/recognition, object detection and semantic image segmentation/parsing. Our approach shows significant improvements over all these core tasks in comparison with the baselines. For instance, on image recognition, our 50-layers network outperforms in terms of recognition performance on ImageNet dataset its counterpart baseline ResNet with 152 layers, while having 2.39 times less parameters, 2.52 times lower computational complexity and more than 3 times less layers. On image segmentation, our novel framework sets a new state-of-the-art on the challenging ADE20K benchmark for scene parsing.
Code is available at: https://github.com/iduta/pyconv

## 1 Introduction

Convolutional Neural Networks (CNNs) [1], [2] represent the workhorses of the most current computer vision applications. Nearly every recent state-of-the-art architecture for different tasks on visual recognition is based on CNNs [3]–[18]. At the core of a CNN there is the convolution, which learns spatial kernels/filters for visual recognition. Most CNNs use a relatively small kernel size, usually $3\times3$, forced by the fact that increasing the size comes with significant costs in terms of number of parameters and computational complexity. To cope with the fact that a small kernel size cannot cover a large region of the input, CNNs use a chain of convolutions with small kernel size and downsampling layers, to gradually reduce the size of the input and to increase the receptive field of the network. However, there are two issues that can appear. First, even though for many of current CNNs the theoretical receptive field can cover a big part of the input or even the whole input, in [19] it is shown that the empirical receptive field is much smaller than the theoretical one, even more than 2.7 times smaller in the higher layers of the network. Second, downsampling the input without previously having access to enough context information (especially in complex scenes as in Fig. 1) may affect the learning process and the recognition performance of the network, as useful details are lost since the receptive filed is not large enough to capture different dependencies in the scene before performing the downsampling.

Natural images can contain extremely complicated scenes. Two examples (an outdoor and an indoor scenes in the wild) are presented in Fig. 1 on the left, while on the right we have the semantic label of each pixel in the scene (taken from the ADE20K dataset [20]). Parsing these images and providing a semantic category of each pixel is very challenging, however, it is one of the holy grail goals of the computer vision field. We can notice in these examples a big number of class categories in the same scene, some partially occluded, and different object scales.

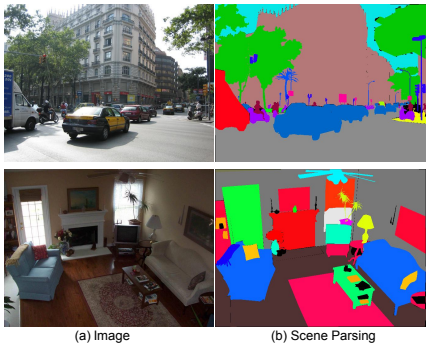


Figure 1: Scene parsing examples. An outdoor and indoor images with their associated pixel-level semantic category.

We can see in Fig. 1 that some class categories can have a very large spatial representations (e.g. buildings, trees or sofas) while other categories can have significantly smaller representations in the image (e.g. persons, books or the bottle). Furthermore, the same object category can appear at different scales in the same image. For instance, the scales of cars in Fig. 1 vary significantly, from being one of the biggest objects in the image to cover just a very tiny portion of the scene. To be able to capture such a diversity of categories and such a variability in their scales, the use of a single type of kernel (as in standard convolution) with a single spatial size, may not be an optimal solution for such complexity. One of the long standing goals of computer vision is the ability to process the input at multiple scales for capturing detailed information about the content of the scene. One of the most notorious example in the hand-crafted features era is SIFT [21], which extracts the descriptors at different scales. However, in the current deep learning era with learned features, the standard convolution is not implicitly equipped with the ability to process the input at multiple scales, and contains a single type of kernel with a single spatial size and depth.

To address the aforementioned challenges, this work provides the following main contributions: **(1)** We introduce pyramidal convolution (PyConv), which contains different levels of kernels with varying size and depth. Besides enlarging the receptive field, PyConv can process the input using increasing kernel sizes in parallel, to capture different levels of details. On top of these advantages, PyConv is very efficient and, with our formulation, it can maintain a similar number of parameters and computational costs as the standard convolution. PyConv is very flexible and extendable, opening the door for a large variety of network architectures for numerous tasks of computer vision (see Section 3). **(2)** We propose two network architectures for image classification task that outperform the baselines by a significant margin. Moreover, they are efficient in terms of number of parameters and computational costs and can outperform other more complex architectures (see Section 4). **(3)** We propose a new framework for semantic segmentation. Our novel head for parsing the output provided by a backbone can capture different levels of context information from local to global. It provides state-of-the-art results on scene parsing (see Section 5). **(4)** We present network architectures based on PyConv for object detection and video classification tasks, where we report significant improvements in recognition performance over the baseline (see Appendix).

## 2 Related Work

Among the various methods employed for image recognition, the residual networks (ResNets) family [7], [8], [16] represents one of the most influential and widely used. By using a shortcut connection, it facilitates the learning process of the network. These networks are used as backbones for various complex tasks, such as object detection and instance segmentation [7], [13]–[18]. We use ResNets as baselines and make use of such architectures when building our different networks.

The seminal work [3] uses a form of grouped convolution to distribute the computation of the convolution over two GPUs for overcoming the limitations of computational resources (especially memory). Furthermore, also [16] uses grouped convolution but with the aim of improving the recognition performance in the ResNeXt architectures. We also make use of grouped convolution but in a different architecture. The works [17] and [22] propose squeeze-and-excitation and non-local blocks to capture context information. However, these are additional blocks that need to be inserted into the CNN; therefore, these approaches still need to use a spatial convolution in their overall CNN architecture (thus, they can be complementary to our approach). Furthermore, these blocks significantly increase the model and computational complexity.

On the challenging task of semantic segmentation, a very powerful network architecture is PSP-Net [23], which uses a pyramid pooling module (PPM) head on top of a backbone in order to parse the scene for extracting different levels of details. Another powerful architecture is presented in [24], which uses atrous spatial pyramid pooling (ASPP) head on top of a backbone. In contrast to these competitive works, we propose a novel head for parsing the feature maps provided by a backbone, using a local multi-scale context aggregation module and a global multi-scale context aggregation block for efficient parsing of the input. Our novel framework for image segmentation is not only very competitive in terms of recognition performance but is also significantly more efficient in terms of model and computational complexity than these strong architectures.

## 3 Pyramidal Convolution

The standard convolution, illustrated in Fig. 2(a), contains a single type of kernel: with a single spatial size $K_1{}^2$ (in the case of square kernels, e.g., height×width: $3\times3 = 3^2$, $K_1 = 3$) and the depth equal to the number of input feature maps $FM_i$. The result of applying a number of $FM_o$ kernels (all having the same spatial resolution and the same depth) over $FM_i$ input feature maps is a number of $FM_o$ output feature maps (with spatial height $H$ and width $W$). Thus, the number of parameters and FLOPs (floating point operations) required for the standard convolution are: $parameters = K_1{}^2 \cdot FM_i \cdot FM_o$; $FLOPs = K_1{}^2 \cdot FM_i \cdot FM_o \cdot (W \cdot H)$.

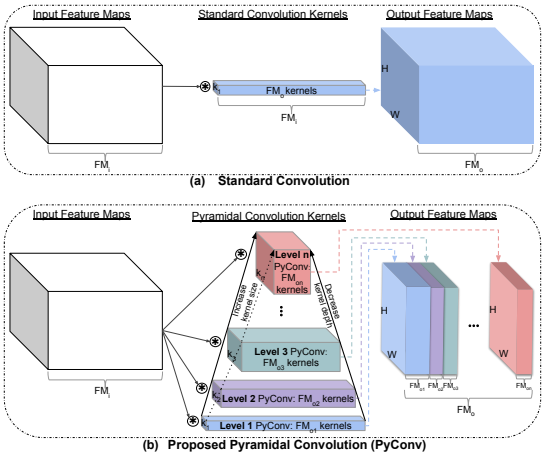


Figure 2: (a) Standard conv; (b) Proposed PyConv.

The proposed pyramidal convolution (PyConv), illustrated in Fig. 2(b), contains a pyramid with $n$ levels of different types of kernels. The goal of the proposed PyConv is to process the input at different kernel scales without increasing the computational cost or the model complexity (in terms of parameters). At each level of the PyConv, the kernel contains a different spatial size, increasing kernel size from the bottom of the pyramid (level 1 of PyConv) to the top (level $n$ of PyConv). Simultaneously with increasing the spatial size, the depth of the kernel is decreased from level 1 to level $n$. Therefore, as shown in Fig. 2(b), this results in two interconnected pyramids, facing opposite directions. One pyramid has the base at the bottom (evolving to the top by decreasing the kernel depth) and the other pyramid has the base on top, where the convolution kernel has the largest spatial size (evolving to the bottom by decreasing the spatial size of the kernel).

To be able to use different depths of the kernels at each level of PyConv, the input feature maps are split into different groups, and apply the kernels independently for each input feature maps group. This is called grouped convolution, an illustration is presented in Fig. 3 where we show three examples (the color encodes the group assignment). In these examples, there are eight input and output feature maps. Fig. 3(a) shows the case comprising a single group of input feature maps, this is the standard convolution, where the depth of the kernels is equal to the number of input feature maps. In this case, each output feature map is connected to all input feature maps. Fig. 3(b) shows the case when the input feature maps are split into two groups, where the kernels are applied independently over each group, therefore, the depth of the kernels is reduced by two. As shown in Fig. 3, when the number of groups is increased, the connectivity (and thus the depth of the kernels) decreases. As a result, the number of parameters and the computational cost of a convolution is reduced by a factor equal to the number of groups.

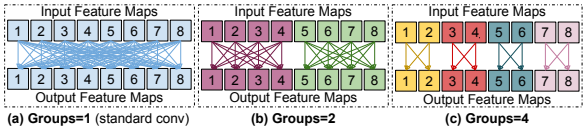


Figure 3: Grouped Convolution.

As illustrated in Fig. 2(b), for the input feature maps $FM_i$, each level of the PyConv $\{1, 2, 3, ..., n\}$ applies different kernels with a different spatial size for each level $\{K_1{}^2, K_2{}^2, K_3{}^2, ..., K_n{}^2\}$ and with different kernel depths $\{FM_i, \frac{FM_i}{(\frac{K_2{}^2}{K_1{}^2})}, \frac{FM_i}{(\frac{K_3{}^2}{K_1{}^2})}, ..., \frac{FM_i}{(\frac{K_n{}^2}{K_1{}^2})}\}$, which outputs a different number

of output feature maps $\{FM_{o_1}, FM_{o_2}, FM_{o_3}, ..., FM_{o_n}\}$ (with height $H$ and width $W$). Therefore, the number of parameters and the computational cost (in terms of FLOPs) for PyConv are:

$$\left|\begin{array}{l} parameters = \\ {K_n}^2 \cdot \frac{FM_i}{(\frac{{K_n}^2}{{K_1}^2})} \cdot FM_{o_n} + \\ \vdots \\ {K_3}^2 \cdot \frac{FM_i}{(\frac{{K_3}^2}{{K_1}^2})} \cdot FM_{o_3} + \\ {K_2}^2 \cdot \frac{FM_i}{(\frac{{K_2}^2}{{K_1}^2})} \cdot FM_{o_2} + \\ {K_1}^2 \cdot FM_i \cdot FM_{o_1}; \end{array}\right| \left|\begin{array}{l} FLOPs = \\ {K_n}^2 \cdot \frac{FM_i}{(\frac{{K_n}^2}{{K_1}^2})} \cdot FM_{o_n} \cdot (W \cdot H) + \\ \vdots \\ {K_3}^2 \cdot \frac{FM_i}{(\frac{{K_3}^2}{{K_1}^2})} \cdot FM_{o_3} \cdot (W \cdot H) + \\ {K_2}^2 \cdot \frac{FM_i}{(\frac{{K_2}^2}{{K_1}^2})} \cdot FM_{o_2} \cdot (W \cdot H) + \\ {K_1}^2 \cdot FM_i \cdot FM_{o_1} \cdot (W \cdot H), \end{array}\right| \quad (1)$$

where $FM_{o_n} + \cdots + FM_{o_3} + FM_{o_2} + FM_{o_1} = FM_o$. Each row in these equations represents the number of parameters and computational cost for a level in PyConv. If each level of PyConv outputs an equal number of feature maps, then the number of parameters and the computational cost of PyConv are distributed evenly along each pyramid level. With this formulation, regardless of the number of levels of PyConv and the continuously increasing kernel spatial sizes from ${K_1}^2$ to ${K_n}^2$, the computational cost and the number of parameters is similar to the standard convolution with a single kernel size ${K_1}^2$. To link the illustration in Fig. 3 with Equations 1, the denominator of $FM_i$ in Equations 1 refers to the number of groups (G) that the input feature maps $FM_i$ are split in Fig. 3.

除数 分母

In practice, when building a PyConv there are several additional rules. The denominator of $FM_i$ at each level of the pyramid in Equations 1, should be a divisor of $FM_i$. In other words, at each pyramid level, the number of feature maps from each created group should be equal. Therefore, as an approximation, when choosing the number of groups for each level of the pyramid (and thus the depth of the kernel), we can take the closest number to the denominator of $FM_i$ from the list of possible divisors of $FM_i$. Furthermore, the number of groups for each level should be also a divisor for the number of output feature maps of each level of PyConv. To be able to easily create different network architectures with PyConv, it is recommended that the number of input feature maps, the groups for each level of pyramid, and the number of output feature maps for each level of PyConv, to be numbers of power of 2. Next sections show practical examples.

The main advantages of the proposed PyConv are: (1) **Multi-Scale Processing.** Besides the fact that, compared to the standard convolution, PyConv can enlarge the receptive field of the kernel without additional costs, it also applies in parallel different types of kernels, having different spatial resolutions and depths. Therefore, PyConv parses the input at multiple scales capturing more detailed information. This double-oriented pyramid of kernels types, where on one side the kernel sizes are increasing and on the other side the kernel depths (connectivity) are decreasing (and vice-versa), allows PyConv to provide a very diverse pool of combinations of different kernel types that the network can explore during learning. The network can explore from large receptive fields of the kernels with lower connectivity to smaller receptive fields with higher connectivity. These different types of kernels of PyConv bring complementary information and help boosting the recognition performance of the network. The kernels with smaller receptive field can focus on details, capturing information about smaller objects and/or parts of the objects, while increasing the kernels size provides more reliable details about larger objects and/or context information.

(2) **Efficiency.** In comparison with the standard convolution, PyConv maintains, by default, a similar number of model parameters and requirements in computational resources, as shown in Equation 1. Furthermore, PyConv offers a high degree of parallelism due to the fact that the pyramid levels can be independently computed in parallel. Thus, PyConv can also offer the possibility of customizable heavy network architectures (in the case where the architecture cannot fit into the memory of a computing unit and/or it is too expensive in terms of FLOPs), where the levels of PyConv can be executed independently on different computing units and then the outputs can be merged.

(3) **Flexibility.** PyConv opens the door for a great variety of network architectures. The user has the flexibility to choose the number of layers of the pyramid, the kernel sizes and depths at each PyConv level, without paying the price of increasing the number of parameters or the computational costs. Furthermore, the number of output feature maps can be different at each level. For instance, for a particular final task it may be more useful to have less output feature maps from the kernels with small receptive fields and more output feature maps from the kernels with bigger receptive fields. Also, the PyConv settings can be different along the network, thus, at each layer of the network we can have different PyConv settings. For example, we can start with several layers for PyConv, and based on the resolution of the input feature maps at each layer of the network, we can decrease the levels of PyConv as the resolution decreases along the network. That being said, we can now build architectures using PyConv for different visual recognition tasks.

## 4 PyConv Networks for Image Classification

For our PyConv network architectures on image classification, we use a residual bottleneck building block similar to the one reported in [7]. Fig. 4 shows an example of a building block used on the first stage of our network. First, it applies a $1\times1$ conv to reduce the input feature maps to 64, then we use our proposed PyConv with four levels of different kernels sizes: $9\times9$, $7\times7$, $5\times5$, $3\times3$. Also the depth of the kernels varies along each level, from 16 groups to full depth/connectivity. Each level outputs 16 feature maps, which sums a 64 output feature maps for the PyConv. Then a $1\times1$ conv is applied to regain the initial number of feature maps. As common, batch normalization [6] and ReLU activation function [25] follow a conv block. Finally, there is a shortcut connection that can help with the identity mapping.

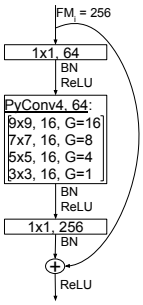


Figure 4: PyConv bottleneck building block.

Our proposed network for image classification, **PyConvResNet**, is illustrated in Table 1. For direct comparison we place aside also the baseline architecture ResNet [7]. Table 1 presents the case for a 50-layers deep network, for the other depths, the number of layers are increased as in [7]. Along the network we can identify four main stages, based on the spatial size of the feature maps. For PyConvResNet, we start with a PyConv with four levels. Since the spatial size of the feature maps decreases at each stage, we reduce also the PyConv levels. On the last main stage, the network ends up with only one level for PyConv, which is basically the standard convolution. This is appropriate because the spatial size of the feature maps is only $7\times7$, thus, three successive convolutions of size $3\times3$ cover well the feature maps. Regarding the efficiency, PyConvResNet provides also a slight decrease in FLOPs.

As we highlighted, flexibility is a strong point of PyConv, Table 1 presents another architecture based on PyConv, **PyConvHGResNet**, which uses a higher grouping for each level. For this architecture we set a minimum of 32 groups and a maximum of 64 in the PyConv. The number of feature maps for the spatial convolutions is doubled to provide better capabilities on learning spatial filters. Note that for stage one of the network, it is not possible to increase the number of groups more than 32 since this is the number of input and output feature maps for each level. Thus, PyConvHGResNet produces a slight increase in FLOPs.

Table 1: PyConvResNet and PyConvHGResNet.

| stage | output | ResNet-50 | PyConvResNet-50 | PyConvHGResNet-50 |
|---|---|---|---|---|
| | 112×112 | 7×7, 64, s=2 | 7×7, 64, s=2 | 7×7, 64, s=2 |
| | | 3×3 max pool,s=2 | | |
| 1 | 56×56 | [1×1, 64; 3×3, 64; 1×1, 256] ×3 | [1×1, 64; PyConv4, 64: [9×9, 16, G=16; 7×7, 16, G=8; 5×5, 16, G=4; 3×3, 16, G=1]; 1×1, 256] ×3 | [1×1, 128; PyConv4, 128: [9×9, 32, G=32; 7×7, 32, G=32; 5×5, 32, G=32; 3×3, 32, G=32]; 1×1, 256] ×3 |
| 2 | 28×28 | [1×1, 128; 3×3, 128; 1×1, 512] ×4 | [1×1, 128; PyConv3, 128: [7×7, 64, G=8; 5×5, 32, G=4; 3×3, 32, G=1]; 1×1, 512] ×4 | [1×1, 256; PyConv3, 256: [7×7, 128, G=64; 5×5, 64, G=64; 3×3, 64, G=32]; 1×1, 512] ×4 |
| 3 | 14×14 | [1×1, 256; 3×3, 256; 1×1, 1024] ×6 | [1×1, 256; PyConv2, 256: [5×5, 128, G=4; 3×3, 128, G=1]; 1×1, 1024] ×6 | [1×1, 512; PyConv2, 512: [5×5, 256, G=64; 3×3, 256, G=32]; 1×1, 1024] ×6 |
| 4 | 7×7 | [1×1, 512; 3×3, 512; 1×1, 2048] ×3 | [1×1, 512; PyConv1, 512: [3×3, 512, G=1]; 1×1, 2048] ×3 | [1×1, 1024; PyConv1, 1024: [3×3, 1024, G=32]; 1×1, 2048] ×3 |
| | 1×1 | global avg pool<br>1000-d fc | global avg pool<br>1000-d fc | global avg pool<br>1000-d fc |
| # params | | **25.56** × $10^6$ | **24.85** × $10^6$ | **25.23** × $10^6$ |
| FLOPs | | **4.14** × $10^9$ | **3.88** × $10^9$ | **4.61** × $10^9$ |

As our networks contain different levels of kernels, it can perform the downsampling of the feature maps using different kernel sizes. This is important as downsampling produces loss of spatial resolution and therefore loss of details, but having different kernel sizes to perform the downsampling can take into account different levels of spatial context dependencies to perform the dowsampling in parallel. As can be seen in Table 1, the original ResNet [7] uses a max pooling layer before the first stage of the network to downsample the feature maps and to get the translation invariance. Different from the original ResNet, we move the max pooling on the first projection shortcut, just before the $1\times1$ conv (usually, the first shortcut of a stage contains a projection $1\times1$ conv to adapt the number of feature maps and their spatial resolution for the summation with the output of the block). This is similar to projection shortcut in [26]. Therefore, for the original ResNet, the downsampling is not performed by the first stage (as the max pooling performs this before), the next three main stages perform the downsampling on their first block. In our networks, all four main stages perform the downsampling in their first block.

This change does not increase the number of parameters of the network and does not affect significantly the computational costs (as can be seen in Table 1, as the first block uses the spatial convolutions with the stride 2), providing advantages in recognition performance for our networks. Moving the max pooling to the shortcut gives our approach the opportunity to have access to larger spatial resolution of the feature maps in the first block of the first stage, to downsample the input using multiple kernel scales and, at the same time, to benefit from the translation invariance provided by max pooling. The results show that our networks provide improved recognition capabilities.

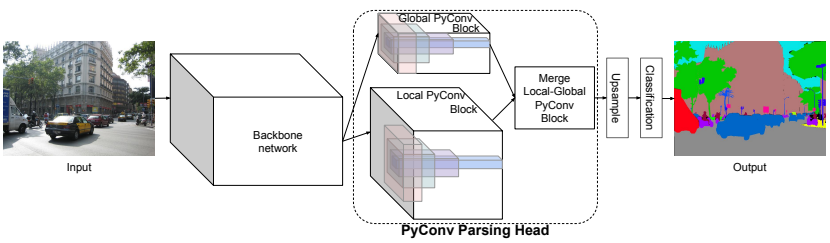


Figure 5: PyConvSegNet framework for image segmentation.

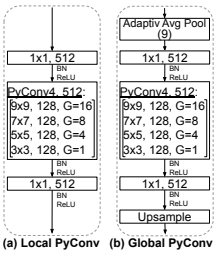


Figure 6: PyConv blocks.

## 5 PyConv Network on Semantic Segmentation

Our proposed framework for scene parsing (image segmentation) is illustrated in Fig. 5. To build an effective pipeline for scene parsing, it is necessary to create a head that can parse the feature maps provided by the backbone and obtain not only local but also global information. The head should be able to deal with fine details and, at the same time, take into account the context information. We propose a novel head for scene parsing (image segmentation) task, PyConv Parsing Head (PyConvPH). The proposed PyConvPH is able to deal with both local and global information at multiple scales.

PyConvPH contains three main components: (1) **Local PyConv block** (LocalPyConv), which is mostly responsible for smaller objects and capturing local fine details at multiple scales, as shown in Fig. 5. It also applies different type of kernels, with different spatial sizes and depths, which can also be seen as a local multi-scale context aggregation module. The detailed information about each component of LocalPyConv is represented in Fig. 6(a). LocalPyConv takes the output feature maps from the backbone and applies a $1\times1$ conv to reduce the number of feature maps to 512. Then, it performs a PyConv with four layers to capture different local details at four scales of the kernel $9\times9$, $7\times7$, $5\times5$, and $3\times3$. Additionally, the kernels have different connectivity, represented by the number of groups (G). Finally, it applies a $1\times1$ conv to combine the information extracted at different kernel sizes and depths. As is standard, all convolution blocks are followed by a batch normalization layer [6] and a ReLU activation function [25].

(2) **Global PyConv block** (GlobalPyConv) is responsible for capturing global details about the scene, and for dealing with very large objects. It is a multi-scale global aggregation module. The components of GlobalPyConv are represented in Fig. 6(b). As the input image size can vary, to ensure that we can capture full global information we keep the largest spatial size dimension as 9. We apply an adaptive average pooling that reduces the spatial size of the feature maps to $9\times9$ (in the case of square images), which still maintains reasonable spatial resolution. Then we apply a $1\times1$ conv to reduce the number of feature maps to 512. We use a PyConv with four layers similarly as in the LocalPyConv. However, as now we have decreased the spatial size of the feature maps to $9\times9$, the PyConv kernels can cover very large parts of the input, ultimately, the layer with a $9\times9$ convolution covers the whole input and captures full global information, as illustrated also in Fig. 5. Then we apply a $1\times1$ conv to fuse the information from different scales. Finally, we upsample the feature maps to the initial size before the adaptive average pooling, using a bilinear interpolation.

(3) **Merge Local-Global PyConv block** performs first the concatenation of the output feature maps from the LocalPyConv and GlobalPyConv blocks. Over the resulting 1024 feature maps, it applies a PyConv with one level, which is basically a standard $3\times3$ conv that outputs 256 feature maps. We use here a single level for PyConv because the previous layers already captured all levels of context information and it is more important at this point to focus on merging this information (thus, to use full connectivity of the kernels) as it approaches the final classification stage. To provide the final output, the framework continues with an upsample layer (using also bilinear interpolation) to restore the feature maps to the initial input image size; finally, there is a classification layer which contains a $1\times1$ conv, to provide the output with a dimension equal to the number of classes. As illustrated in Fig. 5, our proposed framework is able to capture local and global information at multiple scales of kernels, parsing the image and providing a strong representation. Furthermore, our framework is also very efficient, and in the following we provide the exact numbers and the comparison with other state-of-the-art frameworks.

# 6 Experiments

**Experimental setup.** For image classification task we perform our experiments on the commonly used ImageNet dataset [27]. It consists of 1000 classes, 1.28 million training images and 50K validation images. We report both top-1 and top-5 error rates. We follow the settings in [7], [8], [28] and use the SGD optimizer with a standard momentum of 0.9, and weight decay of 0.0001. We train the model for 90 epochs, starting with a learning rate of 0.1 and reducing it by 1/10 at the 30-th, 60-th and 80-th epochs, similarly to [7], [28]. The models are trained using 8 GPUs V100. We use the standard 256 training mini-batch size and data augmentation as in [5], [28], training/testing on $224\times224$ image crop. For image segmentation we use ADE20K benchmark [20], which is one of the most challenging datasets for image segmentation/parsing. It contains 150 classes and a high level of scenes diversity, containing both object and stuff classes. It is divided in 20K/2K/3K images for training, validation and testing. As standard, we report both pixel-wise accuracy (pAcc.) and mean of class-wise intersection over union (mIoU). We train for 100 epochs with a mini-batch size of 16 over 8 GPUs V100, using train/test image crop size of $473\times473$. We follow the training settings as in [23], including the auxiliary loss, with the weight 0.4.

**PyConv results on image recognition**. We present in Table 2 the ablation experiments results of the proposed PyConv for image recognition task on the ImageNet dataset where, using the network with 50 layers, we vary the number of levels of PyConv. We first provide a direct comparison to the baseline ResNet [7] without any additional changes, just replacing the standard 3x3 conv with our PyConv. The column "PyConv levels" points to the number of levels used at each of the four main stages of the network. The PyConv levels (1, 1, 1, 1) represent the case when we use a single level for PyConv on all four stages, which is basically the baseline ResNet. Remarkably, just increasing the number of PyConv levels to two provides a significant improvement in recognition performance, improving the top-1 error rate from 23.88 to 23.12. At the same time it requires less number of parameters and FLOPS than the baseline. Note that by just using two levels for PyConv ($5\times5$ and $3\times3$ kernels), it has already significantly increased the receptive field at each stage of the network. Gradually increasing the levels of PyConv at each level brings further improvement, for the PyConv levels (4, 3, 2, 1) it brings the top-1 error rate to 22.97 with even lower number of parameters and FLOPs. We also run the experiment using only the top level of the PyConv at each main stage network, basically the opposite case of the baseline which uses only the bottom level. Therefore, top(4, 3, 2, 1) refers to the case when using only the fourth level of the PyConv for the stage 1 ($9\times9$ kernel), third level for stage 2 ($7\times7$ kernel), second level for stage 3 ($5\times5$ kernel) and first level of stage 4 ($3\times3$ kernel). This configuration also provides significant improvements in recognition performance compared to the baseline while having a lower number of parameters and FLOPs, showing that our formulation of increasing the kernel sizes for building the network is beneficial in many possible configurations.

Table 2: ImageNet ablation experiments of PyConvResNet.

| PyConv levels | top-1(%) | top-5(%) | params | GFLOPs |
|---|---|---|---|---|
| (1, 1, 1, 1)baseline | 23.88 | 7.06 | 25.56 | 4.14 |
| (2, 2, 2, 1) | 23.12 | 6.58 | 24.91 | 3.91 |
| (3, 3, 2, 1) | 22.98 | 6.62 | 24.85 | 3.85 |
| (4, 3, 2, 1) | 22.97 | 6.56 | 24.85 | 3.84 |
| top(4, 3, 2, 1) | 23.18 | 6.60 | 24.24 | 3.63 |
| (5, 4, 3, 2) | 23.03 | 6.56 | 23.45 | 3.71 |
| (4, 3, 2, 1) max | 22.46 | 6.24 | 24.85 | 3.88 |
| (4, 3, 2, 1) final | 22.12 | 6.20 | 24.85 | 3.88 |

We also add one more layer to PyConv for each stage of the network, (5, 4, 3, 2) case, where the fifth level has a $11\times11$ kernel, but we do not notice further improvements in recognition performance. In the rest of the paper we use (4, 3, 2, 1) levels of PyConv for image classification task. However, we find this configuration reasonably good for this task with the input image resolution ($224\times224$), however, if, for instance, the input image resolution is increased, then other settings of PyConv may provide even further improvements. Moving the max pooling to the shortcut, which provides access for PyConv to perform the downsampling at multiple kernel sizes, improves further the top-1 error rate to 22.46. To further benefit from the translation invariance and to address the fact that a $1\times1$ conv lacks the spatial resolution for performing downsampling, we maintain a max pooling on the projection shortcut in the first block of each following stages. Our final network result is 22.12 top-1 error requiring only 24.85 million parameters and 3.88 GFLOPs. The conclusion is that regardless of the settings of PyConv, using our formulation, it consistently provides better results than the baseline.

Fig. 7 shows the training and validation curves for comparing our networks, PyConvResNet and PyConvHGResNet, with baseline ResNet over 50, 101 and 152 layers, where we can notice that our networks significantly improve the learning convergence. For instance, on 50 depth, on first interval (first 30 epochs, before the first reduction of the learning rate), our PyConvResNet needs

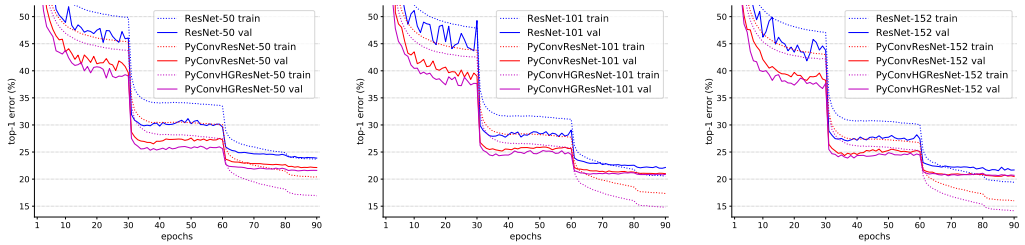


Figure 7: ImageNet training curves for ResNet and PyConvResNet on 50, 101 and 152 layers.

Table 3: Validation error rates comparison results of PyConv on ImageNet with other architectures.

| Network | network depth: 50 | | | | network depth: 101 | | | | network depth: 152 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | top-1 | top-5 | params | GFLOPs | top-1 | top-5 | params | GFLOPs | top-1 | top-5 | params | GFLOPs |
| ResNet (baseline)[7] | 23.88 | 7.06 | 25.56 | 4.14 | 22.00 | 6.10 | 44.55 | 7.88 | 21.55 | 5.74 | 60.19 | 11.62 |
| pre-act. ResNet [8] | 23.77 | 7.04 | 25.56 | 4.14 | 22.11 | 6.26 | 44.55 | 7.88 | 21.41 | 5.78 | 60.19 | 11.62 |
| iResNet [26] | 22.69 | 6.46 | 25.56 | 4.18 | 21.36 | 5.63 | 44.55 | 7.92 | 20.66 | 5.43 | 60.19 | 11.65 |
| NL-ResNet [22] | 22.91 | 6.42 | 36.72 | 6.18 | 21.40 | 5.83 | 55.71 | 9.91 | 21.91 | 6.11 | 71.35 | 13.66 |
| SE-ResNet [17] | 22.74 | 6.37 | 28.07 | 4.15 | 21.31 | 5.79 | 49.29 | 7.90 | 21.38 | 5.80 | 66.77 | 11.65 |
| ResNeXt [16] | 22.44 | 6.25 | 25.03 | 4.30 | 21.03 | 5.66 | 44.18 | 8.07 | 20.98 | 5.48 | 59.95 | 11.84 |
| PyConvHGResNet | **21.52** | **5.94** | 25.23 | 4.61 | **20.78** | **5.57** | 44.63 | 8.42 | **20.64** | **5.27** | 60.66 | 12.29 |
| PyConvResNet | **22.12** | **6.20** | 24.85 | 3.88 | **20.99** | **5.53** | 42.31 | 7.31 | **20.48** | **5.27** | 56.64 | 10.72 |

less than 10 epochs to outperform the best results of ResNet on all first 30 epochs. Thus, because of improved learning capabilities, our PyConvResNet can require significantly less epochs for training to outperform the baseline. Table 3 presents the comparison results of our proposed networks with other state-of-the-art networks on 50, 101, 152 layers. Our networks outperform the baseline ResNet [7] by a large margin on all depths. For instance, our PyConvResNet improves the top-1 error rate from 23.88 to 22.12 on 50 layers, while having lower number of parameters and FLOPs. Remarkably, our PyConvHGResNet with 50 layers outperforms ResNet with 152 layers on top-1 error rate. Besides providing better results than pre-activation ResNet [8] and ResNeXt [16], our networks outperform more complex architectures, like SE-ResNet [17], despite that it uses an additional squeeze-and-excitation block, which increases model complexity.

The above results mainly aim to show the advantages of our PyConv over the standard convolution by running all the networks with the same standard training settings for a fair comparison. Note that there are other works which report better results on ImageNet, such as [29]–[31]. However, the improvements are mainly due to the training settings. For instance, [30] uses very complex training settings, such as, complex data augmentation (autoAugment [32]) with different regularization techniques (dropout [33]), stochastic depth [34], the training is performed on a powerful Google TPU computational architecture over 350 epochs with a large batch of 2048. The works [29], [31], besides using a strong computational architecture with many GPUs, take advantage of a large dataset of 3.5B images collected from Instagram (this dataset is not publicly available). Therefore, these resources are not handy to everyone. However, the results show that PyConv is superior to standard convolution and combining it with [29]–[31] can bring further improvements. While on ImageNet we do not have access to such scale of computational and data resources to directly compete with state-of-the-art, we do push further and show that our proposed framework obtains state-of-the-art results on challenging task of image segmentation.

To support our claim, that our networks can be easily improved using more complex training settings, we integrate an additional data augmentation, CutMix [35]. As CutMix requires more epochs to converge, we increase the training epochs to 300 and use a cosine scheduler [36] for learning rate decaying. To speed-up the training, we increase the batch size to 1024 and use mixed precision [37]. Table 4 presents the comparison results of PyConvResNet for the baseline training settings and with the CutMix data augmentation. For both depths, 50- and 101-layers, just adding these simple additional training settings improve significantly the results. For the same trained models, in addition to the standard test crop size of $224\times224$ we also run the testing on $320\times320$ crop size. This results show that there is still room for improvement if more complex training settings are included (as the training settings from [30]) and/or additional data used for training (as in [29], [31]), however, this requires significantly more computational and data resources, which are not easily available.

Table 4: Validation error rates comparison results of PyConvResNet on ImageNet with different training settings, for network depth 50 and 101 ($^{\dagger}$on the already trained model with 224×224 crop, just perform the test on 320×320).

| Network | test crop: 224×224 | | | test crop: 320×320$^{\dagger}$ | | | params |
|---|---|---|---|---|---|---|---|
| | top-1 | top-5 | GFLOPs | top-1 | top-5 | GFLOPs | |
| PyConvResNet-50 | 22.12 | 6.20 | 3.88 | 21.10 | 5.55 | 7.91 | 24.85 |
| PyConvResNet-50 + augment | **20.56** | **5.31** | 3.88 | **19.41** | **4.75** | 7.91 | 24.85 |
| PyConvResNet-101 | 20.99 | 5.53 | 7.31 | 20.03 | 4.82 | 14.92 | 42.31 |
| PyConvResNet-101 + augment | **19.42** | **4.87** | 7.31 | **18.51** | **4.28** | 14.92 | 42.31 |

Table 5: Head-to-Head comparison on image segmentation (ResNet-50 as backbone) on ADE20K.

| Head | output stride backbone: 8 | | | | output stride backbone: 16 | | | |
|---|---|---|---|---|---|---|---|---|
| | mean IoU | pixel Acc. | params | GFLOPs | mean IoU | pixel Acc. | params | GFLOPs |
| baseline [23]: 3×3 conv | 37.87 | 78.17 | 35.42 | 131.37 | 36.84 | 77.84 | 35.42 | 39.52 |
| DeepLabv3 [24]: ASPP | 40.91 | 79.92 | 41.48 | 151.17 | 40.34 | 79.44 | 41.48 | 44.47 |
| PSPNet [23]: PPM | 41.24 | 80.01 | 49.06 | 165.42 | 39.75 | 79.17 | 49.06 | 48.08 |
| PyConvSegNet: PyConvPH | **41.54** | **80.18** | 34.40 | 116.84 | **40.43** | **79.45** | 34.40 | 36.08 |

**PyConv results on semantic segmentation.** We compare our proposed framework, PyConvSegNet, with two of the most powerful architectures for semantic segmentation [23] and [24]. Table 5 presents head-to-head comparison of our method with state-of-the-art heads on image segmentation: PSPNet with Pyramid Pooling Module (PPM) head, and DeepLabv3 with Atrous Spatial Pyramid Pooling (ASPP). The baseline is constructed as in [23], which as head, it basically applies a 3×3 conv over the output feature maps provided by the backbone. For a fair and direct comparison, all methods use the same auxiliary loss (deep supervision) exactly as in [23]. For a comprehensive view, the reports in terms of number of parameters and FLOPs include the auxiliary loss components. As [23] uses an output stride for the backbone of 8 and [24] uses 16, we report the experiments for both cases. We run these experiments using the ResNet with 50 layers as backbone. Table 5 shows that our proposed head is not only more accurate than the other methods, but it is also more efficient, requiring significantly smaller number of parameters and FLOPs than [23] and [24]. We can also see that without a strong head on top of the backbone, the baseline reports significantly worse results.

Table 6: PyConvSegNet results with different backbones.

| Backbone | mean IoU(%) | | pixel Acc.(%) | | params | GFLOPs |
|---|---|---|---|---|---|---|
| | SS | MS | SS | MS | | |
| ResNet-50 | 41.54 | 42.88 | 80.18 | 80.97 | 34.40 | 116.84 |
| PyConvResNet-50 | 42.08 | 43.31 | 80.31 | 81.18 | 33.69 | 114.18 |
| ResNet-101 | 42.88 | 44.39 | 80.75 | 81.60 | 53.39 | 185.47 |
| PyConvResNet-101 | 42.93 | 44.58 | 80.91 | 81.77 | 51.15 | 177.29 |
| ResNet-152 | 44.04 | 45.28 | 81.18 | 81.89 | 69.03 | 242.00 |
| PyConvResNet-152 | **44.36** | **45.64** | **81.54** | **82.36** | 65.48 | 229.11 |

Table 6 shows the results of PyConvSegNet using different depths of the backbones ResNet and PyConvResNet. Besides the single-scale (SS) inference results, we show also the results using multi-scale inference (MS) (scales equal to {0.5, 0.75, 1, 1.25, 1.5, 1.75}). Table 7 presents the comparisons of our approach with the state-of-the-art on both validation and testing sets. Notably, our approach PyConvSegNet, with 152 layers for backbone, outperforms PSPNet [23] with its 269-layers heavy backbone, which also requires significantly more parameters and FLOPs for their PPM head.

Table 7: State-of-the-art comparison on ADE20K (single model). ($^{\dagger}$ increase the crop size just for inference from 473×473 to 617×617; ♣ just increase training epochs from 100 to 120 and train over training+validation sets; the results on testing set are provided by the official evaluation server, as the labels are not publicly available. The score is the average of mean IoU and pixel Acc. results.)

| Method | Validation set | | Testing set | | |
|---|---|---|---|---|---|
| | mIoU | pAcc. | mIoU | pAcc. | Score |
| FCN [38] | 29.39 | 71.32 | - | - | 44.80 |
| DilatedNet [39] | 32.31 | 73.55 | - | - | 45.67 |
| SegNet [40] | 21.64 | 71.00 | - | - | 40.79 |
| RefineNet [41] | 40.70 | - | - | - | - |
| UperNet [42] | 41.22 | 79.98 | - | - | - |
| PSANet [43] | 43.77 | 81.51 | - | - | - |
| KE-GAN [44] | 37.10 | 80.50 | - | - | - |
| CFNet [45] | 44.89 | - | - | - | - |
| CiSS-Net [46] | 42.56 | 80.77 | - | - | - |
| EncNet [47] | 44.65 | 81.69 | - | - | 55.67 |
| PSPNet-152 [23] | 43.51 | 81.38 | - | - | - |
| PSPNet-269 [23] | 44.94 | 81.69 | - | - | 55.38 |
| PyConvSegNet-152 | 45.64 | 82.36 | 37.75 | 73.61 | 55.68 |
| PyConvSegNet-152 $^{\dagger}$ | **45.99** | **82.49** | - | - | - |
| PyConvSegNet-152 ♣ | - | - | **39.13** | **73.91** | **56.52** |

## 7 Conclusion

In this paper we proposed pyramidal convolution (PyConv), which contains several levels of kernels with varying scales. PyConv shows significant improvements for different visual recognition tasks and, at the same time, it is also efficient and flexible, providing a very large pool of potential network architectures. Our novel framework for image segmentation provides state-of-the-art results. In addition to a broad range of visual recognition tasks, PyConv can have a significant impact in many other directions, such as image restoration, completion/inpainting, noise/artifact removal, enhancement and image/video super-resolution.

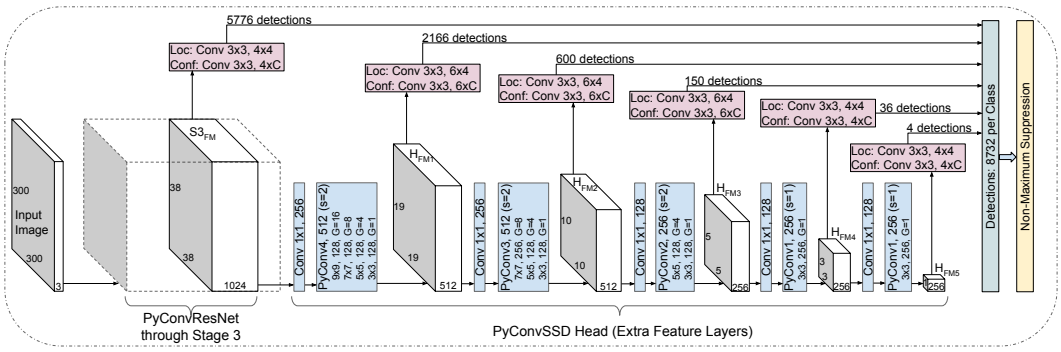


Figure 8: PyConvSSD framework for object detection.

# A Appendix

In this Appendix we present additional experiments, architectures details and/or analysis. It contains three main sections: Section A.1 presents the details of our architecture for object detection; Section A.2 presents the details for the video classification pipeline; Finally, Section A.3 shows some visual examples on image segmentation.

## A.1 PyConv on object detection

As we already presented in the the main paper the final result on object detection, that we outperform the baseline by a significant margin (see main contribution (4) in the main paper), this section provides the details of our architecture on object detection and the exact numbers of the results.

As our proposed PyConv uses different levels of kernel sizes in parallel, it can provide significant benefits for object detection task, where the objects can appear in the image at different scales. For object detection, we integrate our PyConv in a powerful approach, Single Shot Detector (SSD) [48]. SSD is a very efficient single stage framework for object detection, which performs the detection at multiple feature maps resolutions. Our proposed framework for object detection, PyConvSSD, is illustrated in Fig. 8. The framework contains two main parts:

(1) **PyConvResNet Backbone.** In our framework we use the proposed PyConvResNet as backbone, which was previously pre-trained on ImageNet dataset [27]. To maintain a high efficiency of the framework, and also to heave a similar number of output feature maps as in the backbone used in [48], we remove from our PyConvResNet backbone all layers after the third stage. We also set all strides in the stage 3 of the backbone network to 1. With this, PyConvResNet provides (as output of the stage 3) 1024 output feature maps ($S3_{FM}$) with the spatial resolution $38\times38$ (for an input image size of $300\times300$).

(2) **PyConvSSD Head.** Our PyConvSSD head illustrated in Fig. 8 uses the proposed PyConv to further extract different features using different kernel sizes in parallel. Over the resulted feature maps for the third stage of the backbone we apply a PyConv with four levels (kernel sizes: $9\times9$, $7\times7$, $5\times5$, $3\times3$). Also PyConv performs the downsampling (stride $s=2$) of the feature maps using these multiple kernel sizes in parallel. As the feature maps resolution decreases we also decrease the levels of the pyramid for PyConv. The last two PyConv contains only one level (which is basically the standard $3\times3$) as the spatial resolution of the feature maps is very small. Note that the last two PyConvs use a stride $s=1$ and the spatial resolution is decreased just by not using padding. Thus, the head decreases the spatial resolution of the feature maps from $38\times38$ to $1\times1$. All the output feature maps from the PyConvs in the head are used for detections.

For each of the six output feature maps selected for detection $\{S3_{FM}, H_{FM1}, H_{FM2}, H_{FM3}, H_{FM4}, H_{FM5}\}$ the framework performs the detection using a coresponding number of default boxes (anchor boxes) $\{4, 6, 6, 6, 4, 4\}$ for each spatial location. For instance, for ($S3_{FM}$) output feature maps with the spatial resolution $38\times38$, using the four default boxes on each location results in 5776 detections. For localizing each bounding box, there are four values that network should predict (loc: $\Delta(cx, cy, w, h)$, where $cx$ and $cy$ represent the center point of the bounding box, $w$ and $h$ the width and height of the bounding box). This bounding box offset output values are measured relative to a

Table 8: PyConvSSD with $300\times300$ input image size (results on COCO val2017).

| Architecture | Avg. Precision, IoU: | | | Avg. Precision, Area: | | | Avg. Recall, #Dets: | | | Avg. Recall, Area: | | | params | GFLOPs |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0.5:0.95 | 0.5 | 0.75 | S | M | L | 1 | 10 | 100 | S | M | L | | |
| Baseline SSD-50 | 26.20 | 43.97 | 26.96 | 8.12 | 28.22 | 42.64 | 24.50 | 35.41 | 37.07 | 12.61 | 40.76 | 57.25 | 22.89 | 20.92 |
| PyConvSSD-50 | **29.16** | **47.26** | **30.24** | **9.31** | **31.21** | **47.79** | **26.14** | **37.81** | **39.61** | **13.79** | **43.87** | **60.98** | 21.55 | 19.71 |
| Baseline SSD-101 | 29.58 | 47.69 | 30.80 | 9.38 | 31.96 | 47.64 | 26.47 | 38.00 | 39.64 | 14.09 | 43.54 | 61.03 | 41.89 | 48.45 |
| PyConvSSD-101 | **31.27** | **50.00** | **32.67** | **10.65** | **33.76** | **51.75** | **27.33** | **39.33** | **41.07** | **15.48** | **45.53** | **63.44** | 39.01 | 45.02 |

default box position, relative to each feature maps location. Also, for each bounding box, the network should output the confidences for each class category (in total $C$ class categories). For providing the detections the framework uses a classifier which is represented by a $3\times3$ convolution, that outputs for each bounding box the confidences for all class categories ($C$). For localization the framework uses also a $3\times3$ convolution to output the four localization values for each regressed default bounding box. In total, the framework outputs 8732 detections (for $300\times300$ input image size), which pass through a non-maximum suppression to provide the final detections.

Different from the original SSD framework [48], for a fair and direct comparison, in the baseline SSD, we replaced the VGG backbone [4] with ResNet [7], as ResNet is far superior to VGG in terms of recognition performance and computational costs as shown in [7]. Therefore, as main differences from our PyConvSSD, the baseline SSD in this work uses ResNet [7] as backbone and the SSD head uses standard $3\times3$ conv (instead of PyConv) as in the original framework [48]. For showing the exact numbers to compare our PyConvSSD with the baseline on object detection, we use COCO dataset [49], which contains 81 categories. We use for training COCO train2017 (118K images) and for testing COCO val2017 (5K images). We train for 130 epochs using 8 GPUs with 32 batch size each, resulting in 60K training iterations. We use for training SGD optimiser with momentum 0.9, weight decay 0.0005, with the learning rate 0.02 (reduced by 1/10 before 86-th and 108-th epoch). We also use a linear warmup in the first epoch [28]. For data augmentation, we perform random crop as in [48], color jitter and horizontal flip. We use an input image size of $300\times300$ and report the metrics as in [48].

Table 8 shows the comparison results of PyConvSSD with the baseline, over 50- and 101-layers backbones. While being more efficient in terms of number of parameters and FLOPs, the proposed PyConvSSD reports significant improvements over the baseline over all metrics. Notably, PyConvSSD with 50 layers backbone is even competitive with the baseline using 101 layers as backbone. This results show a grasp of the benefits for PyConv on object detection task.

### A.2 PyConv on video classification

In the main paper we introduced the main result for video classification, that we report significant results over the baseline (see main contribution (4)). This section presents the details of the architecture and the exact numbers. PyConv can show significant benefits on video related tasks as it can enlarge the receptive field and process the input using multiple kernels scales in parallel not only spatially but also in the temporal dimension. Extending the networks from image recognition to video involves extending the 2D spatial convolution to 3D spatio-temporal convolution. Table 9 presents the baseline network ResNet3D and our proposed network PyConvResNet3D, which are the initial 2D networks extended to work with video input. The input for the network is represented by 16-frame input clips, with spatial size is $224\times224$. As the temporal size is smaller than spatial dimensions, for our PyConv we do not need to use equally large size on the upper layers of the pyramid. In the first stage of the network, our PyConv with four layers contains kernel sizes of: $7\times9\times9$, $5\times7\times7$, $3\times5\times5$ and $3\times3\times3$ (the temporal dimension comes first).

For video classification, we perform the experiments on Kinetics-400 [50], which is a large-scale video recognition dataset that contains $\sim$246k training videos and 20k validation videos, with 400 action classes. Similar to image recognition, use the SGD optimizer with a standard momentum of 0.9 and weight decay of 0.0001, we train the model for 90 epochs, starting with a learning rate of 0.1 and reducing it by 1/10 at the 30-th, 60-th and 80-th epochs, similar to [7], [28]. The models are trained from scratch, using the weights initialization of [51] for all convolutional layers; for training we use a minibatch of 64 clips over 8 GPUs. Data augmentation is similar to [4], [22]. For training, we randomly select 16-frame input clips from the video. We also skip four frames to cover a longer video period within a clip. The spatial size is $224\times224$, randomly cropped from a scaled video, where

Table 9: Video recognition networks.

| stage | output | ResNet3D-50 | PyConvResNet3D-50 |
|---|---|---|---|
| | 16×112×112 | 5×7×7, 64<br>stride (1,2,2) | 5×7×7, 64<br>stride (1,2,2) |
| | | 1×3×3 max pool<br>stride (1,2,2) | |
| 1 | 16×56×56 | [1×1×1, 64<br>3×3×3, 64<br>1×1×1, 256] ×3 | [1×1×1, 64<br>PyConv4, 64:<br>[7×9×9, 16, G=16<br>5×7×7, 16, G=8<br>3×5×5, 16, G=4<br>3×3×3, 16, G=1]<br>1×1×1, 256] ×3 |
| 2 | 16×28×28 | [1×1×1, 128<br>3×3×3, 128<br>1×1×1, 512] ×4 | [1×1×1, 128<br>PyConv3, 128:<br>[5×7×7, 64, G=8<br>3×5×5, 32, G=4<br>3×3×3, 32, G=1]<br>1×1×1, 512] ×4 |
| 3 | 8×14×14 | [1×1×1, 256<br>3×3×3, 256<br>1×1×1, 1024] ×6 | [1×1×1, 256<br>PyConv2, 256:<br>[3×5×5, 128, G=4<br>3×3×3, 128, G=1]<br>1×1×1, 1024] ×6 |
| 4 | 4×7×7 | [1×1×1, 512<br>3×3×3, 512<br>1×1×1, 2048] ×3 | [1×1×1, 512<br>PyConv1, 512:<br>[3×3×3, 512, G=1]<br>1×1×1, 2048] ×3 |
| | 1×1×1 | global avg pool<br>400-d fc | global avg pool<br>400-d fc |
| # params | | **47.00** × $10^6$ | **44.91** × $10^6$ |
| FLOPs | | **93.26** × $10^9$ | **91.81** × $10^9$ |

Table 10: Video recognition error rates (%).

| Architecture | top-1 | top-5 | params | GFLOPs |
|---|---|---|---|---|
| ResNet3D-50 [7] | 37.01 | 15.41 | 47.00 | 93.26 |
| PyConvResNet3D-50 | **34.56** | **13.34** | 44.91 | 91.81 |

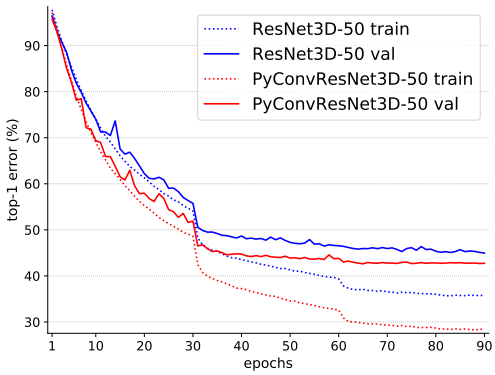


Figure 9: Training and validation curves on Kinetics-400 dataset (these results are computed during training over independent clips).

the shorter side is randomly selected from the interval [256, 320], similar to [4], [22]. As the networks on video data are prone to overfitting due to the increase in number of parameters, we use dropout [52] after the global average pooling layer, with a 0.5 dropout ratio. For the final validation, following common practice, we uniformly select a maximum of 10 clips per video. Each clip is scaled to 256 pixels for the shorter spatial side. We take 3 spatial crops to cover the spatial dimensions. In total, this results in a maximum of 30 clips per video, for each of which we obtain a prediction. To get the final prediction for a video, we average the softmax scores. We report both, top-1 and top-5 error rates.

Table 10 presents the result comparing our network, PyConvResNet3D, with the baseline over 50-layers depth. PyConvResNet3D improves significantly the results over baseline, for top-1 error, from 37.01% to 34.56%. In the same time our network requires less number of parameters and FLOPs than the baseline. Fig. 9 shows the training and validation curves where we can see that our network improves significantly the training convergence. This results show the potential of PyConv on video related tasks.

### A.3 Qualitative examples on image segmentation

Fig. 10 shows some qualitative examples for visually comparing our proposed approach for image segmentation, PyConSegNet, with state-of-the-art approaches PSPNet [23] and DeepLabv3 [24]. For the numeric results, refer to Table 4 in the main paper (for the output stride backbone 8). This examples show the visual comparison results between our proposed head, PyConvPH (PyConv parsing head), with ASPP (Atrous Spatial Pyramid Pooling) of [24] and PPM head (Pyramid Pooling Module) of [23].

Very suggestive is the last row example of Fig. 10, where we can clearly notice the difference in segmentation details. It is remarkable that our proposed head can compete at a high level with other state-of-the art approaches for image segmentation while having significantly less requirements in terms of number of parameters and computational complexity. For instance, in comparison with our PyConSegNet, PSPNet [23] requires over 40% more parameters and FLOPs, while DeepLabv3 [24] requires over 20% more parameters and close to 30% more FLOPs.

In the second row example of Fig. 10 we can also notice a failure case of our approach, which confuses the door with a window. However, this case is quite difficult and confusing even for a human eye. Fig. 11 shows some visual results of our approach, PyConSegNet, using 50-, 101-, 152-layers for the PyConvResNet backbone. For the exact number, refer to Table 5 in the main paper (multi-scale inference). Note in the second row of Fig. 11 how the quality of the segmentation for the fan (ceiling mount air fan) is improving while increasing the depth of our PyConvResNet backbone.

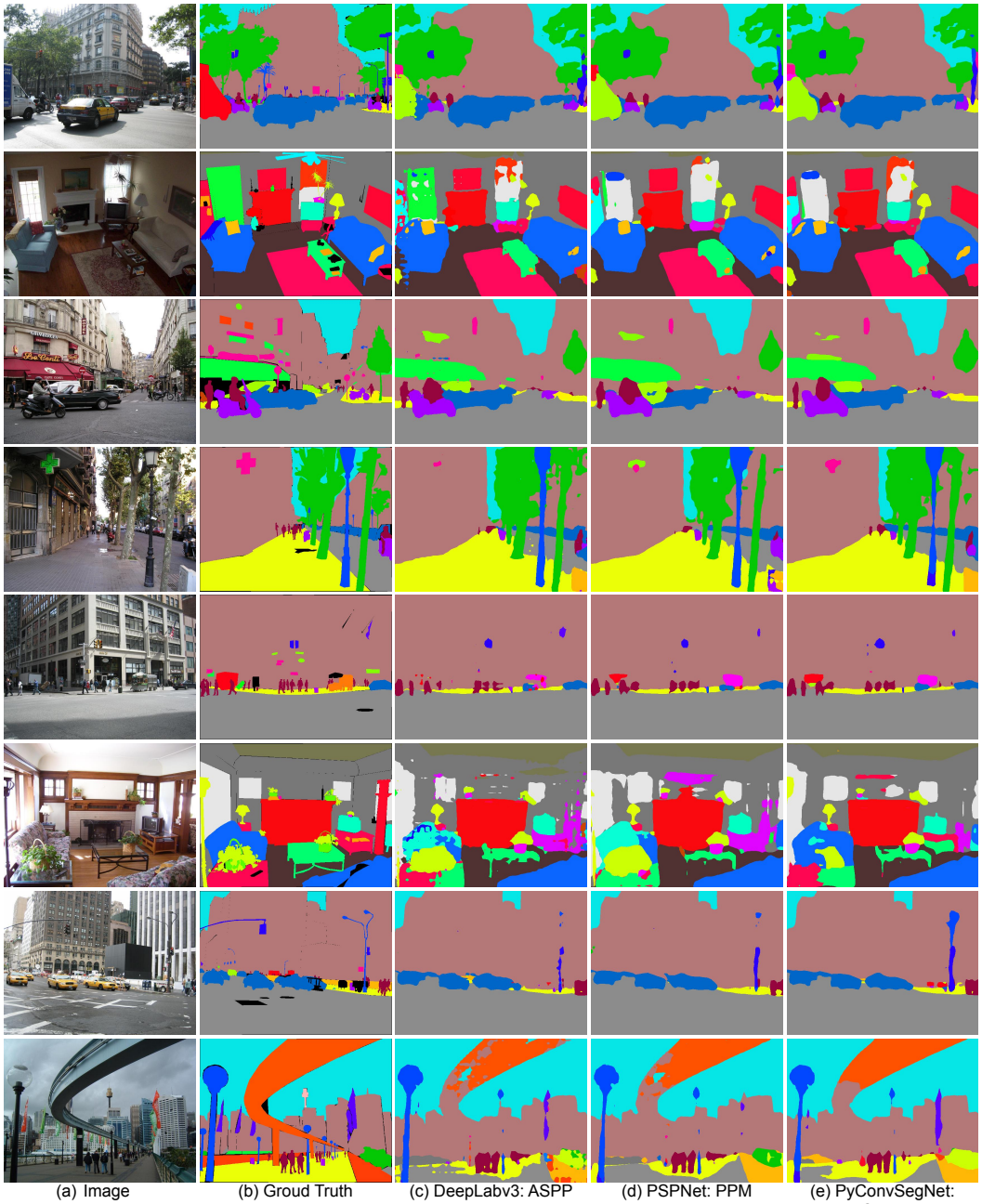


Figure 10: Visual comparison results of our approach PyConSegNet (with PyConvPH head) with state-of-the-art approaches: PSPNet [23] (with PPM head) and DeepLabv3 [24] (with ASPP head). The images are from ADE20K dataset [20] validation.

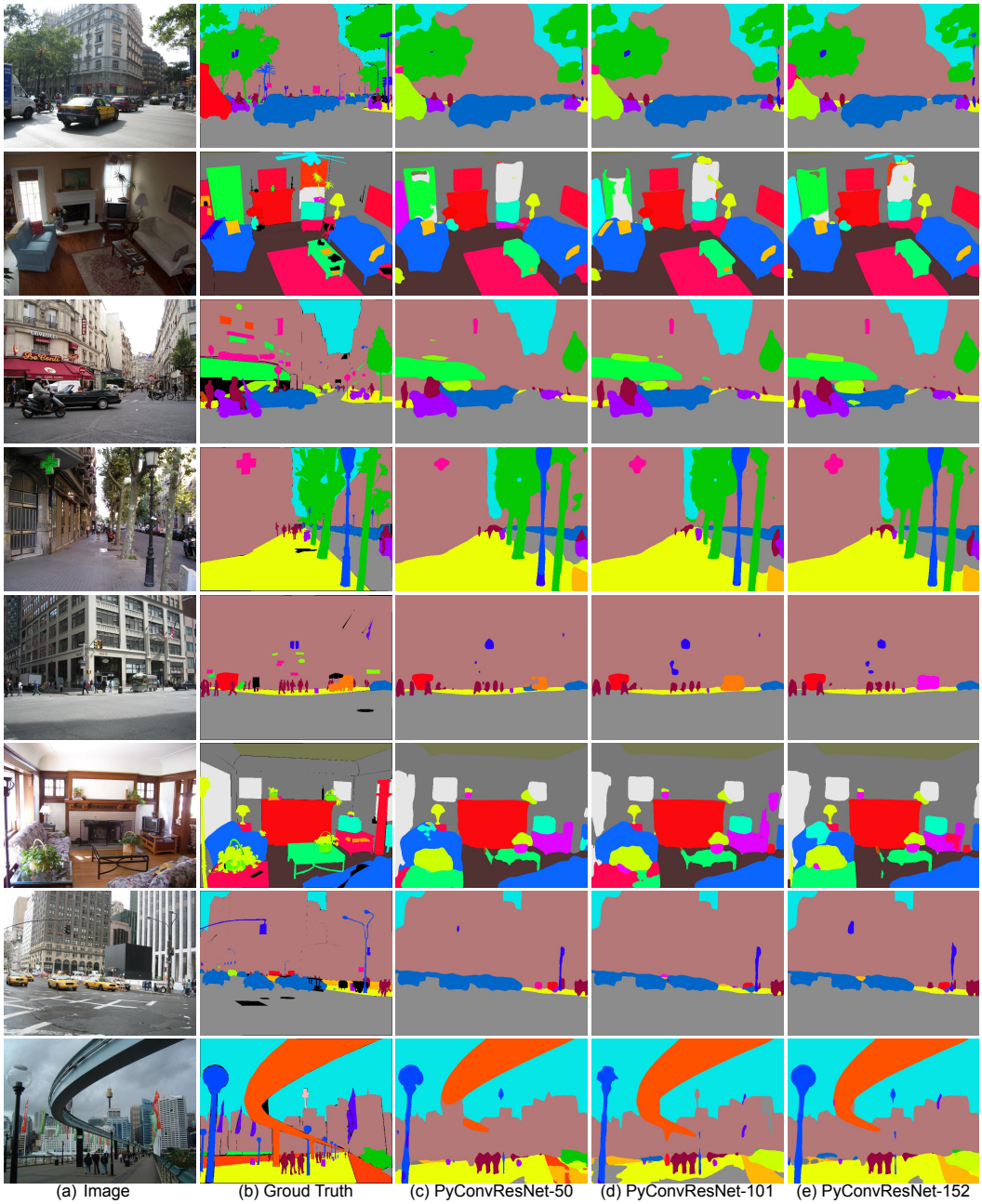


Figure 11: Visual results of our approach, PyConSegNet, on 50-, 101-, 152-layers deep backbone PyConvResNet. The images are from ADE20K dataset [20] validation set.

## References


[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural computation*, vol. 1, no. 4, pp. 541–551, 1989.

[2] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS*, 2012.

[4] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *ArXiv:1409.1556*, 2014.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *CVPR*, 2015.

[6] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *ICML*, 2015.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR*, 2016.

[8] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *ECCV*, 2016.

[9] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *CVPR*, 2017.

[10] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *AAAI*, 2017.

[11] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *CVPR*, 2017.

[12] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *CVPR*, 2018.

[13] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *ICCV*, 2017.

[14] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," in *ICCV*, 2017.

[15] T.-Y. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, and S. Belongie, "Feature pyramid networks for object detection," in *CVPR*, 2017.

[16] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *CVPR*, 2017.

[17] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," in *CVPR*, 2018.

[18] Y. Wu and K. He, "Group normalization," in *ECCV*, 2018.

[19] B. Zhou, A. Khosla, A. Lapedriza, A. Oliva, and A. Torralba, "Object detectors emerge in deep scene cnns," in *ICLR*, 2015.

[20] B. Zhou, H. Zhao, X. Puig, T. Xiao, S. Fidler, A. Barriuso, and A. Torralba, "Semantic understanding of scenes through the ade20k dataset," *IJCV*, vol. 127, no. 3, pp. 302–321, 2019.

[21] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *IJCV*, vol. 60, no. 2, pp. 91–110, 2004.

[22] X. Wang, R. Girshick, A. Gupta, and K. He, "Non-local neural networks," in *CVPR*, 2018.

[23] H. Zhao, J. Shi, X. Qi, X. Wang, and J. Jia, "Pyramid scene parsing network," in *CVPR*, 2017.

[24] L.-C. Chen, G. Papandreou, F. Schroff, and H. Adam, "Rethinking atrous convolution for semantic image segmentation," *ArXiv:1706.05587*, 2017.

[25] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *ICML*, 2010.

[26] I. C. Duta, L. Liu, F. Zhu, and L. Shao, "Improved residual networks for image and video recognition," *ArXiv:2004.04989*, 2020.

[27] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[28] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," *ArXiv:1706.02677*, 2017.

[29] D. Mahajan, R. Girshick, V. Ramanathan, K. He, M. Paluri, Y. Li, A. Bharambe, and L. van der Maaten, "Exploring the limits of weakly supervised pretraining," in *ECCV*, 2018.

[30] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *ICML*, 2019.

[31] H. Touvron, A. Vedaldi, M. Douze, and H. Jégou, "Fixing the train-test resolution discrepancy," in *NeurIPS*, 2019.

[32] E. D. Cubuk, B. Zoph, D. Mane, V. Vasudevan, and Q. V. Le, "Autoaugment: Learning augmentation strategies from data," in *CVPR*, 2019.

[33] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *JMLR*, vol. 15, no. 1, pp. 1929–1958, 2014.

[34] G. Huang, Y. Sun, Z. Liu, D. Sedra, and K. Q. Weinberger, "Deep networks with stochastic depth," in *ECCV*, 2016.

[35] S. Yun, D. Han, S. J. Oh, S. Chun, J. Choe, and Y. Yoo, "Cutmix: Regularization strategy to train strong classifiers with localizable features," in *ICCV*, 2019.

[36] I. Loshchilov and F. Hutter, "Sgdr: Stochastic gradient descent with warm restarts," in *ICLR*, 2017.

[37] P. Micikevicius, S. Narang, J. Alben, G. Diamos, E. Elsen, D. Garcia, B. Ginsburg, M. Houston, O. Kuchaiev, G. Venkatesh, *et al.*, "Mixed precision training," *ArXiv:1710.03740*, 2017.

[38] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *CVPR*, 2015.

[39] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," in *ICLR*, 2016.

[40] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *TPAMI*, vol. 39, no. 12, pp. 2481–2495, 2017.

[41] G. Lin, A. Milan, C. Shen, and I. Reid, "Refinenet: Multi-path refinement networks for high-resolution semantic segmentation," in *CVPR*, 2017.

[42] T. Xiao, Y. Liu, B. Zhou, Y. Jiang, and J. Sun, "Unified perceptual parsing for scene understanding," in *ECCV*, 2018.

[43] H. Zhao, Y. Zhang, S. Liu, J. Shi, C. Change Loy, D. Lin, and J. Jia, "Psanet: Point-wise spatial attention network for scene parsing," in *ECCV*, 2018.

[44] M. Qi, Y. Wang, J. Qin, and A. Li, "Ke-gan: Knowledge embedded generative adversarial networks for semi-supervised scene parsing," in *CVPR*, 2019.

[45] H. Zhang, H. Zhang, C. Wang, and J. Xie, "Co-occurrent features in semantic segmentation," in *CVPR*, 2019.

[46] Y. Zhou, X. Sun, Z.-J. Zha, and W. Zeng, "Context-reinforced semantic segmentation," in *CVPR*, 2019.

[47] H. Zhang, K. Dana, J. Shi, Z. Zhang, X. Wang, A. Tyagi, and A. Agrawal, "Context encoding for semantic segmentation," in *CVPR*, 2018.

[48] 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.

[49] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft coco: Common objects in context," in *ECCV*, 2014.

[50] W. Kay, J. Carreira, K. Simonyan, B. Zhang, C. Hillier, S. Vijayanarasimhan, F. Viola, T. Green, T. Back, P. Natsev, *et al.*, "The kinetics human action video dataset," *ArXiv:1705.06950*, 2017.

[51] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *ICCV*, 2015.

[52] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *ArXiv:1207.0580*, 2012.