CSPNet: A New Backbone that can Enhance Learning Capability of CNN

Chien-Yao Wang¹, Hong-Yuan Mark Liao¹,², Yueh-Hua Wu¹,³, Ping-Yang Chen⁴, Jun-Wei Hsieh⁵, and I-Hau Yeh⁶

¹Institute of Information Science, Academia Sinica, Taiwan
²Department of Computer Science and Information Engineering, Providence University, Taiwan
³Department of Computer Science and Information Engineering, National Taiwan University, Taiwan
⁴Department of Computer Science, National Chiao Tung University, Taiwan
⁵College of Artificial Intelligence and Green Energy, National Chiao Tung University, Taiwan
⁶Elan Microelectronics Corporation, Taiwan

kinyiu@iis.sinica.edu.tw, liao@iis.sinica.edu.tw, kriswu@iis.sinica.edu.tw, pingyang.cs08g@nctu.edu.tw, jwhsieh@nctu.edu.tw, and ihyeh@emc.com.tw

Abstract

Neural networks have enabled state-of-the-art approaches to achieve incredible results on computer vision tasks such as object detection. However, such success greatly relies on costly computation resources, which hinders people with cheap devices from appreciating the advanced technology. In this paper, we propose Cross Stage Partial Network (CSPNet) to mitigate the problem that previous works require heavy inference computations from the network architecture perspective. We attribute the problem to the duplicate gradient information within network optimization. The proposed networks respect the variability of the gradients by integrating feature maps from the beginning and the end of a network stage, which, in our experiments, reduces computations by 20% with equivalent or even superior accuracy on the ImageNet dataset, and significantly outperforms state-of-the-art approaches in terms of AP₅₀ on the MS COCO object detection dataset. The CSPNet is easy to implement and general enough to cope with architectures based on ResNet, ResNeXt, and DenseNet.

1. Introduction

Neural networks have been shown to be especially powerful when it gets deeper [8, 37, 11] and wider [38]. However, extending the architecture of neural networks usually brings up a lot more computations, which makes computationally heavy tasks such as object detection unaffordable for most people. Light-weight computing has gradually received stronger attention since real-world applications usually require short inference time on small devices, which poses a serious challenge for computer vision algorithms. Although some approaches were designed exclusively for mobile CPU [10, 30, 9, 32, 41, 23], the depth-wise convolution they adopted is usually not compatible with industrial IC design such as Application-Specific Integrated Circuit (ASIC) for edge-computing systems. In this work, we investigate the computational burden in state-of-the-art ap-
proaches such as ResNet, ResNeXt, and DenseNet. We further develop computationally efficient components that enable the mentioned networks to be deployed on both CPUs and mobile GPUs without sacrificing the performance.

In this study, we introduce Cross Stage Partial Network (CSPNet). The main purpose of designing CSPNet is to enable this architecture to achieve a richer gradient combination while reducing the amount of computation. This aim is achieved by partitioning feature map of the base layer into two parts and then merging them through a proposed cross-stage hierarchy. Our main concept is to make the gradient flow propagate through different network paths by splitting the gradient flow. In this way, we have confirmed that the propagated gradient information can have a large correlation difference by switching concatenation and transition steps. In addition, CSPNet can greatly reduce the amount of computation, and improve inference speed as well as accuracy, as illustrated in Fig 1. The proposed CSPNet-based object detector deals with the following three problems:

1) Strengthening learning ability of a CNN The accuracy of existing CNN is greatly degraded after lightweighting, so we hope to strengthen CNN’s learning ability, so that it can maintain sufficient accuracy while being lightweighting. The proposed CSPNet can be easily applied to ResNet, ResNeXt, and DenseNet. After applying CSPNet on the above mentioned networks, the computation effort can be reduced from 10% to 20%, but it outperforms ResNet [8], ResNeXt [37], DenseNet [11], HarDNet [2], Elastic [34], and Res2Net [5], in terms of accuracy, in conducting image classification task on ImageNet [3].

2) Removing computational bottlenecks Too high a computational bottleneck will result in more cycles to complete the inference process, or some arithmetic units will often idle. Therefore, we hope we can evenly distribute the amount of computation at each layer in CNN so that we can effectively upgrade the utilization rate of each computation unit and thus reduce unnecessary energy consumption. It is noted that the proposed CSPNet makes the computational bottlenecks of PeleeNet [35] cut into half. Moreover, in the MS COCO [17] dataset-based object detection experiments, our proposed model can effectively reduce 80% computational bottleneck when tested on YOLOv3-based models.

3) Reducing memory costs The wafer fabrication cost of Dynamic Random-Access Memory (DRAM) is very expensive, and it also takes up a lot of space. If one can effectively reduce the memory cost, he/she will greatly reduce the cost of ASIC. In addition, a small area wafer can be used in a variety of edge computing devices. We adopt cross-channel pooling [6] to compress the feature maps during the feature pyramid generating process. In this way, the proposed CSPNet with the proposed detector can cut down 75% memory usage on PeleeNet when generating feature pyramids.

Since CSPNet is able to promote the learning capability of a CNN, we thus use smaller models to achieve 50% COCO AP\textsubscript{50} at 109 fps on GTX 1080ti. Since CSPNet can effectively cut down a significant amount of memory traffic, our proposed method can achieve 40% COCO AP\textsubscript{50} at 52 fps on Intel Core i9-9900K. In addition, since CSPNet can significantly lower down the computational bottleneck and Exact Fusion Model (EFM) can effectively cut down the required memory bandwidth, our proposed method can achieve 42% COCO AP\textsubscript{50} at 49 fps on Nvidia Jetson TX2.

2. Related work

CNN architectures design. In ResNeXt [37], Xie et al. first demonstrate that cardinality can be more effective than the dimensions of width and depth. DenseNet [11] can significantly reduce the number of parameters and computations due to the strategy of adopting a large number of reuse features. And it concatenates the output features of all preceding layers as the next input, which can be considered as the way to maximize cardinality. SparseNet [44] adjusts dense connection to exponentially spaced connection can effectively improve parameter utilization and thus result in better outcomes. Wang et al. further explain why high cardinality and sparse connection can improve the learning ability of the network by the concept of gradient combination and developed the partial ResNet (PRN) [33]. For improving the inference speed of CNN, Ma et al. [23] introduce four guidelines to be followed and design ShuffleNet-v2. Chao et al. [2] proposed a low memory traffic CNN called Harmonic DenseNet (HarDNet) and a metric Convolutional Input/Output (CIO) which is an approximation of DRAM traffic proportional to the real DRAM traffic measurement.
Real-time object detector. The most famous two real-time object detectors are YOLOv3 [28] and SSD [20]. Based on SSD, LRF [36] and RFBNet [18] can achieve state-of-the-art real-time object detection performance on GPU. Recently, anchor-free based object detector [4, 43, 13, 14, 40] has become mainstream object detection system. Two object detectors of this sort are CenterNet [43] and CornerNet-Lite [14], and they both perform very well in terms of efficiency and efficacy. For real-time object detection on CPU or mobile GPU, SSD-based Pelee [35], YOLOv3-based PRN [35], and Light-Head RCNN [16]-based ThunderNet [25] all receive excellent performance on object detection.

3. Method

3.1. Cross Stage Partial Network

Cross Stage Partial Network. The mainstream CNN architectures, such as ResNet [8], ResNeXt [37], DenseNet [11], their output is usually a linear or non-linear combination of the outputs of intermediate layers. Therefore, the output of a k-layer CNN can be expressed as follows:

\[ y = F(x_0) = x_k = H_k(x_{k-1}, H_{k-1}(x_{k-2}), H_{k-2}(x_{k-3}), \ldots, H_1(x_0), x_0) \]  

where \( F \) is the mapping function from input \( x_0 \) to target \( y \), which is also the model of the entire CNN. As for \( H_k \), it is the operation function of the \( k \)th layer of the CNN. Usually, \( H_k \) is composed of a set of convolutional layers and a non-linear activation function. If we use ResNet and DenseNet as examples, they can be represented by Equation 2 and Equation 3 respectively as follows:

\[ x_k = R_k(x_{k-1}) + x_{k-1} = R_k(x_{k-1}) + R_{k-1}(x_{k-2}) + \ldots + R_1(x_0) + x_0 \]  

(2)

\[ x_k = [D_k(x_{k-1}), x_{k-1}] = [D_k(x_{k-1}), D_{k-1}(x_{k-2}), \ldots, D_1(x_0), x_0] \]  

(3)

In the above two equations, \( R \) and \( D \) respectively represent the computation operators of the residual layer and dense layer, and these operators often composed of 2~3 convolutional layers.

From the above two equations, whether it is a residual layer or a dense layer, the input of each convolutional layer that composes them receives the outputs from all the previous layers. Under these circumstances, the length of gradient path can be minimized and makes gradient flow propagation more efficient in the back propagation process. However, we also know that this architecture design will make the \( k \)th layer pass the gradient to all \( k-1, k-2, \ldots, 1 \) layers and use it to update the weights, which will cause repeated learning redundant information.

Recently, some studies have tried to use the input of screened \( H_k(\cdot) \) to improve learning ability and parameter utilization. For example, SparseNet [44] uses exponentially spaced connection to make \( H_k \) directly related to \( H_{k-1}, H_{k-2}, H_{k-4}, \ldots, H_{k-2}, \ldots \) only. ShuffleNetV2 [23] use split channels to make \( H_k \) directly related to only half of \( H_{k-1} \) channels, and its equation can be expressed as \( S(H_k(x_{k-1}[1 : c/2]), x_{k-1}[c/2 + 1 : c]) \), where \( S \) represents the shuffle operation, and \( x_{k-1}[1 : c/2] \) represents the first to the \( c/2 \) channels of \( x_{k-1} \). As for the PyramidNet [7] and PRN [33], they all use feature maps with unequal number of channels to build ResNet to achieve the effect of gradient shunting.

The state-of-the-art methods put their emphasis on optimizing the \( H_i \) function at each layer, and we propose that CSPNet directly optimizes the \( F \) function as follows:

\[ y = M \left( [x_{0}^{\prime}, T(F(x_{0}^{\prime})] \right) \]  

(4)

where \( x_0 \) is split into two parts along channel and it can be represented as \( x_0 = [x_0^{\prime}, x_0^{\prime\prime}] \). \( T \) is the transition function used to truncate the gradient flows of \( H_1, H_2, \ldots, H_k \), and \( M \) is the transition function used to mix the two segmented parts. Next, we will show examples of how to integrate CSPNet into DenseNet and explain how to solve the problem of learning duplicate information in CNN.

DenseNet. Figure 2 (a) shows the detailed structure of one-stage of the DenseNet proposed by Huang et al. [11]. Each stage of a DenseNet contains a dense block and a transition layer, and each dense block is composed of \( k \) dense layers. The output of the \( i \)th dense layer will be concatenated with the input of the \( i \)th dense layer, and the concatenated outcome will become the input of the \( (i + 1) \)th dense layer. The equations showing the above-mentioned mechanism can be expressed as:

\[ \begin{align*}
    x_1 & = w_1 \ast x_0 \\
    x_2 & = w_2 \ast [x_0, x_1] \\
    \vdots \\
    x_k & = w_k \ast [x_0, x_1, \ldots, x_k-1]
\end{align*} \]  

(5)

where \( \ast \) represents the convolution operator, and \([x_0, x_1, \ldots, x_k]\) means to concatenate \( x_0, x_1, \ldots \), and \( w_i \) and \( x_i \) are the weights and output of the \( i \)th dense layer, respectively.

If one makes use of a backpropagation to update weights, the equations of weight updating can be written as:

\[ \begin{align*}
    w_1' &= f_1(w_1, [g_1]) \\
    w_2' &= f_2(w_2, [g_2, g_1]) \\
    \vdots \\
    w_k' &= f_k(w_k, [g_k, g_{k-1}])
\end{align*} \]  

(6)

where \( f_i \) is the function of weight updating of \( i \)th dense layer, and \( g_i \) represents the gradient propagated to the \( i \)th dense layer. We can find that large amount of gradient information are reused for updating weights of different dense layers. This will result in different dense layers repeatedly learn copied gradient information.
Cross Stage Partial DenseNet. The architecture of one-stage of the proposed CSPDenseNet is shown in Figure 2 (b). A stage of CSPDenseNet is composed of a partial dense block and a partial transition layer. In a partial dense block, the feature maps of the base layer in a stage are split into two parts through channel $x_0 = [x_0', x_0'']$. Between $x_0'$ and $x_0''$, the former is directly linked to the end of the stage, and the latter will go through a dense block. All steps involved in a partial transition layer are as follows: First, the output of dense layers, $[x_0', x_1, ..., x_k]$, will undergo a transition layer. Second, the output of this transition layer, $x_T$, will be concatenated with $x_0''$ and undergo another transition layer, and then generate output $x_U$. The equations of feed-forward pass and weight updating of CSPDenseNet are shown in Equations 7 and 8, respectively.

$$x_K = W_K * [x_0', x_1, ..., x_K']$$
$$x_T = W_T * [x_0'', x_1, ..., x_T']$$
$$x_T'' = W_T'' * [x_0'', x_T'']$$
$$W'_K = f_K(W_K, \{g_0', g_1, ..., g_K\})$$
$$W''_T = f_T(W_T, \{g_0'', g_1, ..., g_T\})$$

We can see that the gradients coming from the dense layers are separately integrated. On the other hand, the feature map $x_0'$ that did not go through the dense layers is also separately integrated. As to the gradient information for updating weights, both sides do not contain duplicate gradient information that belongs to other sides.

Overall speaking, the proposed CSPDenseNet preserves the advantages of DenseNet’s feature reuse characteristics, but at the same time prevents an excessively amount of duplicate gradient information by truncating the gradient flow. This idea is realized by designing a hierarchical feature fusion mechanism, which uses the strategy of truncating the gradient flow to prevent distinct layers from learning duplicate gradient information. Here we design two variations of CSPDenseNet to show how this sort of gradient flow truncating affects the learning ability of a network. Figure 3 (c) and Figure 3 (d) show two different fusion strategies. CSP (fusion first) means to concatenate the feature maps generated by two parts, and then do transition operation. If this strategy is adopted, a large amount of gradient information will be reused. As to the CSP (fusion last) strategy, the output from the dense block will go through the transition layer and then do concatenation with the feature map coming from part 1. If one goes with the CSP (fusion last) strategy, the gradient information will not be reused since the gradient flow is truncated. If we use the four architectures shown in 3 to perform image classification, the corresponding results are shown in Figure 4. It can be seen that if one adopts the CSP (fusion last) strategy to perform image classification, the computation cost is significantly dropped, but the top-1 accuracy only drop 0.1%. On the other hand, the CSP (fusion first) strategy does help the significant drop in computation cost, but the top-1 accu-
racy significantly drops 1.5%. By using the split and merge strategy across stages, we are able to effectively reduce the possibility of duplication during the information integration process. From the results shown in Figure 4, it is obvious that if one can effectively reduce the repeated gradient information, the learning ability of a network will be greatly improved.

![Diagram](image)

Figure 5: Applying CSPNet to ResNe(X)t.

**Apply CSPNet to Other Architectures.** CSPNet can also be applied to ResNet and ResNeXt, the architectures are shown in Figure 5. Since only half of the feature channels are going through Res(X)Blocks, there is no need to introduce the bottleneck layer anymore. This makes the theoretical lower bound of the Memory Access Cost (MAC) when the Floa6-16ting-point OPerations (FLOPs) is fixed.

### 3.2. Exact Fusion Model

**Looking Exactly to predict perfectly.** We propose EFM that captures an appropriate receptive field for each anchor, which enhances the accuracy of the one-stage object detector. For segmentation tasks, since pixel-level labels usually do not contain global information, it is usually more preferable to consider larger patches for better information retrieval [21]. However, for tasks like image classification and object detection, some critical information can be obscure when observed from image-level and bounding box-level labels. Li et al. [15] found that CNN can be often distracted when it learns from image-level labels and concluded that it is one of the main reasons that two-stage object detectors outperform one-stage object detectors.

**Aggregate Feature Pyramid.** The proposed EFM is able to better aggregate the initial feature pyramid. The EFM is based on YOLOv3 [28], which assigns exactly one bounding-box prior to each ground truth object. Each ground truth bounding box corresponds to one anchor box that surpasses the threshold IoU. If the size of an anchor box is equivalent to the receptive field of the grid cell, then for the grid cells of the $s^{th}$ scale, the corresponding bounding box will be lower bounded by the $(s - 1)^{th}$ scale and upper bounded by the $(s + 1)^{th}$ scale. Therefore, the EFM assembles features from the three scales.

**Balance Computation.** Since the concatenated feature maps from the feature pyramid are enormous, it introduces a great amount of memory and computation cost. To alleviate the problem, we incorporate the Maxout technique to compress the feature maps.

### 4. Experiments

We use ImageNet image classification dataset [3] to verify proposed CSPNet. And use the MS COCO object detection dataset [17] to verify the proposed CSPNet and EFM.

#### 4.1. Implementation Details

**ImageNet.** In ImageNet image classification experiments, all hyper-parameters such as training steps, learning rate schedule, optimizer, data augmentation, etc., we all follow the settings defined in Redmon et al. [28]. For ResNet-based models and ResNeXt-based models, we set 8,000,000 training steps. As to DenseNet-based models, we set 1,600,000 training steps. We set the initial learning rate 0.1 and adopt the polynomial decay learning rate scheduling strategy. The momentum and weight decay are respectively set as 0.9 and 0.005. All architectures use a single GPU to train universally in the batch size of 128. Finally, we use the validation set of ILSVRC 2012 to validate our method.

**MS COCO.** In MS COCO object detection experiments, all hyper-parameters also follow the settings defined in Redmon et al. [28]. Altogether we did 500,000 training steps. We adopt the step decay learning rate schedule and multiply with a factor 0.1 at the 400,000 steps and the 450,000 steps, respectively. The momentum and weight decay are respectively set as 0.9 and 0.0005. All architectures use a single GPU to execute multi-scale training in the batch size of 64. Finally, the COCO test-dev set is adopted to verify our method.

#### 4.2. Ablation Experiments

**Ablation study of CSPNet on ImageNet.** In the ablation experiments conducted on the CSPNet, we adopt PeleeNet [35] as the baseline, and the ImageNet is used to verify the performance of the CSPNet. We use different partial ratios $\gamma$ and feature fusion strategies for ablation study. Table 1 shows the results of ablation study on CSPNet. As to CSP (fusion first) and CSP (fusion last), they are proposed to validate the benefits of a partial transition.

From the experimental results of CSP (fusion last), the partial transition layer designed to reduce the learning of redundant information can achieve very good performance. For example, when the computation is cut down by 21%, the accuracy only degrades by 0.1%. One thing to be noted is that when $\gamma = 0.25$, the computation is cut down by 11%, but the accuracy is increased by 0.1%. Compared to the baseline PeleeNet, the proposed CSPPeleeNet achieves the best performance, it can cut down 13% computation, but at the same time upgrade the accuracy by 0.2%. If we adjust the partial ratio to $\gamma = 0.25$, we are able to upgrade the accuracy by 0.8% and cut down 3% computation.
As reflected in the experiment results, the proposed EFM is 2 fps slower than GFM, but its AP50 is significantly upgraded by 2.4%. The GloU can upgrade AP by 0.7%, but the AP50 is significantly degraded by 2.7%. For edge computing, what really matters is the number and locations of the objects. Therefore, we will not use GloU training in the subsequent models. SAM can get a better frame rate and AP than SPP, so we use EFM (SAM) as the final architecture.

4.3. ImageNet Image Classification

We apply the CSPNet to ResNet-10 [8], ResNeXt-50 [37], DenseNet-201 [11], PeleeNet [35], and DenseNet-201-Elastic [34] and compare with state-of-the-art methods. The experimental results are shown in Table 3.

Table 2: Ablation study of EFM on MS COCO.

<table>
<thead>
<tr>
<th>Head</th>
<th>global fusion</th>
<th>exact fusion</th>
<th>atten.</th>
<th>BFLOPs</th>
<th>FPS</th>
<th>AP</th>
<th>AP50</th>
<th>AP75</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRN [35]</td>
<td>✓</td>
<td></td>
<td></td>
<td>3.590</td>
<td>169</td>
<td>23.1</td>
<td>44.5</td>
<td>22.0</td>
</tr>
<tr>
<td>PRN-3l [33]</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.586</td>
<td>151</td>
<td>23.7</td>
<td>46.0</td>
<td>22.6</td>
</tr>
<tr>
<td>CEM [25]</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.049</td>
<td>148</td>
<td>23.8</td>
<td>45.4</td>
<td>22.6</td>
</tr>
<tr>
<td>CEM (SAM) [25]</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>4.165</td>
<td>144</td>
<td>24.1</td>
<td>46.0</td>
<td>23.1</td>
</tr>
<tr>
<td>GFM</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.605</td>
<td>134</td>
<td>24.3</td>
<td>46.2</td>
<td>23.3</td>
</tr>
<tr>
<td>EFM</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.868</td>
<td>132</td>
<td>26.4</td>
<td>48.6</td>
<td>26.3</td>
</tr>
<tr>
<td>EFM (GloU) [29]</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.868</td>
<td>132</td>
<td>27.1</td>
<td>45.9</td>
<td>28.2</td>
</tr>
<tr>
<td>EFM (SAM)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>5.068</td>
<td>129</td>
<td>26.8</td>
<td>49.0</td>
<td>26.7</td>
</tr>
<tr>
<td>EFM (SPP)</td>
<td>✓</td>
<td></td>
<td></td>
<td>4.863</td>
<td>128</td>
<td>26.2</td>
<td>48.5</td>
<td>25.7</td>
</tr>
</tbody>
</table>

It is confirmed by experimental results that no matter it is which architecture, when the concept of CSPNet is introduced, the computational load is reduced and the accuracy is either remain unchanged or upgraded, especially useful for the improvement of lightweight models. For example, compared to ResNet-10, CSPResNet-10 can improve accuracy by 1.8%. As to PeleeNet and DenseNet-201-Elastic, CSPPeleeNet and CSPDenseNet-201-Elastic can respectively cut down 13% and 19% computation, and either upgrade a little bit or maintain the accuracy. As to the case of ResNeXt-50, CSPResNeXt-50 can cut down 22% computation and upgrade top-1 accuracy to 77.9%.

Proposed CSPResNeXt-50 is also compared with ResNet-152 [8], DenseNet-264 [11], and HarDNet-138s [2], regardless of #parameter, BFLOPs, and top-1 accuracy, CSPResNeXt-50 all achieve the best result. As to the 10-crop test, CSPResNeXt-50 outperforms Res2NeXt-50 [5].
Table 4: Compare with state-of-the-art methods on MSCOCO Object Detection.

<table>
<thead>
<tr>
<th>Method</th>
<th>Backbone</th>
<th>Size</th>
<th>FPS</th>
<th>BFLOPs</th>
<th>#Parameter</th>
<th>AP</th>
<th>AP50</th>
<th>AP75</th>
<th>APs</th>
<th>AMP_5</th>
<th>AMP_7</th>
<th>AMP_9</th>
</tr>
</thead>
<tbody>
<tr>
<td>YOLOv3 [28]</td>
<td>DarkNet53 [23]</td>
<td>608×608</td>
<td>30</td>
<td>140.7</td>
<td>62.3M</td>
<td>33.0</td>
<td>57.9</td>
<td>34.4</td>
<td>18.3</td>
<td>35.4</td>
<td>41.9</td>
<td></td>
</tr>
<tr>
<td>YOLOv3 (SPP)</td>
<td>DarkNet53 [23]</td>
<td>608×608</td>
<td>30</td>
<td>141.5</td>
<td>62.9M</td>
<td>36.2</td>
<td>60.6</td>
<td>38.2</td>
<td>20.6</td>
<td>37.4</td>
<td>46.1</td>
<td></td>
</tr>
<tr>
<td>LRF [36]</td>
<td>ResNet101[8]</td>
<td>512×512</td>
<td>31</td>
<td>-</td>
<td>-</td>
<td>37.3</td>
<td>58.5</td>
<td>39.7</td>
<td>19.7</td>
<td>42.8</td>
<td>50.1</td>
<td></td>
</tr>
<tr>
<td>SSD [20]</td>
<td>HarDNet85 [2]</td>
<td>512×512</td>
<td>32</td>
<td>-</td>
<td>-</td>
<td>35.1</td>
<td>54.8</td>
<td>37.6</td>
<td>15.0</td>
<td>38.9</td>
<td>51.5</td>
<td></td>
</tr>
<tr>
<td>M2Det [42]</td>
<td>VGG16 [31]</td>
<td>320×320</td>
<td>33</td>
<td>-</td>
<td>-</td>
<td>33.5</td>
<td>52.4</td>
<td>35.6</td>
<td>14.4</td>
<td>37.6</td>
<td>47.6</td>
<td></td>
</tr>
<tr>
<td>PFPNet (R) [12]</td>
<td>VGG16 [31]</td>
<td>320×320</td>
<td>33</td>
<td>-</td>
<td>-</td>
<td>31.8</td>
<td>52.9</td>
<td>33.6</td>
<td>12.0</td>
<td>35.5</td>
<td>46.1</td>
<td></td>
</tr>
<tr>
<td>RFBNet [18]</td>
<td>VGG16 [31]</td>
<td>512×512</td>
<td>35</td>
<td>-</td>
<td>-</td>
<td>33.8</td>
<td>54.2</td>
<td>35.9</td>
<td>16.2</td>
<td>37.1</td>
<td>47.4</td>
<td></td>
</tr>
<tr>
<td>PANet (SPP) [19]</td>
<td>CSPResNeXt50</td>
<td>608×608</td>
<td>35</td>
<td>100.6</td>
<td>56.9M</td>
<td>38.4</td>
<td>60.6</td>
<td>41.6</td>
<td>22.1</td>
<td>41.8</td>
<td>47.6</td>
<td></td>
</tr>
<tr>
<td>SSD [20]</td>
<td>HarDNet68 [2]</td>
<td>512×512</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>31.7</td>
<td>51.0</td>
<td>33.8</td>
<td>12.5</td>
<td>35.1</td>
<td>47.9</td>
<td></td>
</tr>
<tr>
<td>LRF [36]</td>
<td>VGG16 [31]</td>
<td>512×512</td>
<td>38</td>
<td>-</td>
<td>-</td>
<td>36.2</td>
<td>56.6</td>
<td>38.7</td>
<td>19.0</td>
<td>39.9</td>
<td>48.8</td>
<td></td>
</tr>
<tr>
<td>PFPNet (S) [12]</td>
<td>VGG16 [31]</td>
<td>300×300</td>
<td>39</td>
<td>-</td>
<td>-</td>
<td>29.6</td>
<td>49.6</td>
<td>31.1</td>
<td>10.6</td>
<td>32.0</td>
<td>44.9</td>
<td></td>
</tr>
<tr>
<td>RetInDet [39]</td>
<td>VGG16 [31]</td>
<td>320×320</td>
<td>40</td>
<td>-</td>
<td>-</td>
<td>29.4</td>
<td>49.2</td>
<td>31.3</td>
<td>10.0</td>
<td>32.0</td>
<td>44.4</td>
<td></td>
</tr>
<tr>
<td>HSD [1]</td>
<td>VGG16 [31]</td>
<td>320×320</td>
<td>40</td>
<td>-</td>
<td>-</td>
<td>33.5</td>
<td>53.2</td>
<td>36.1</td>
<td>15.0</td>
<td>35.0</td>
<td>47.1</td>
<td></td>
</tr>
<tr>
<td>SSD [20]</td>
<td>VGG16 [31]</td>
<td>300×300</td>
<td>44</td>
<td>70.4</td>
<td>34.3M</td>
<td>25.7</td>
<td>43.9</td>
<td>26.2</td>
<td>6.9</td>
<td>27.7</td>
<td>42.6</td>
<td></td>
</tr>
<tr>
<td>PANet (SPP) [19]</td>
<td>CSPResNeXt50</td>
<td>512×512</td>
<td>44</td>
<td>71.3</td>
<td>56.9M</td>
<td>38.0</td>
<td>60.0</td>
<td>40.8</td>
<td>19.7</td>
<td>41.4</td>
<td>49.9</td>
<td></td>
</tr>
<tr>
<td>CenterNet [43]</td>
<td>ResNet101[8]</td>
<td>512×512</td>
<td>45</td>
<td>-</td>
<td>-</td>
<td>34.6</td>
<td>53.0</td>
<td>36.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YOLOv3 [28]</td>
<td>DarkNet53 [23]</td>
<td>416×416</td>
<td>46</td>
<td>65.9</td>
<td>62.3M</td>
<td>31.0</td>
<td>55.3</td>
<td>32.3</td>
<td>15.2</td>
<td>33.2</td>
<td>42.8</td>
<td></td>
</tr>
<tr>
<td>EFGRNet [24]</td>
<td>VGG16 [31]</td>
<td>320×320</td>
<td>48</td>
<td>-</td>
<td>-</td>
<td>33.2</td>
<td>53.4</td>
<td>35.4</td>
<td>13.4</td>
<td>37.1</td>
<td>47.9</td>
<td></td>
</tr>
<tr>
<td>PANet (SPP) [19]</td>
<td>CSPResNeXt50</td>
<td>416×416</td>
<td>53</td>
<td>47.1</td>
<td>56.9M</td>
<td>36.6</td>
<td>58.1</td>
<td>39.0</td>
<td>16.2</td>
<td>39.5</td>
<td>50.9</td>
<td></td>
</tr>
<tr>
<td>TTFNet [22]</td>
<td>DarkNet53 [23]</td>
<td>512×512</td>
<td>54</td>
<td>-</td>
<td>-</td>
<td>35.1</td>
<td>52.5</td>
<td>37.8</td>
<td>17.0</td>
<td>38.5</td>
<td>49.5</td>
<td></td>
</tr>
<tr>
<td>YOLOv3 [28]</td>
<td>DarkNet53 [23]</td>
<td>320×320</td>
<td>56</td>
<td>39.0</td>
<td>62.3M</td>
<td>28.2</td>
<td>51.5</td>
<td>29.7</td>
<td>11.9</td>
<td>30.6</td>
<td>43.4</td>
<td></td>
</tr>
<tr>
<td>PANet (SPP) [19]</td>
<td>CSPResNeXt50</td>
<td>320×320</td>
<td>58</td>
<td>27.9</td>
<td>56.9M</td>
<td>33.4</td>
<td>54.0</td>
<td>35.1</td>
<td>11.8</td>
<td>35.3</td>
<td>50.9</td>
<td></td>
</tr>
</tbody>
</table>

1 The table is separated into four parts, <100 fps, 100~200 fps, 200~300 fps, and >300 fps.
2 We mainly focus on FPS and AP50 since almost all applications need fast inference to locate and count objects.
3 Inference speed are tested on GTX 1080ti with batch size equals to 1 if possible, and our models are tested using Darknet [27].
4 All results are obtained by COCO test-dev set except for TTFNet [22] models which are verified on minival5k set.

4.4. MS COCO Object Detection

In the task of object detection, we aim at three targeted scenarios: (1) real-time on GPU: we adopt CSPResNeXt50 with PANet (SPP) [19]; (2) real-time on mobile GPU: we adopt CSPDenseNet-based models with the proposed EFM (SAM); and (3) real-time on CPU: we adopt CSPDenseNet-based models with PRN [33]. The comparisons between the above models and the state-of-the-art methods are listed in Table 4. As to the analysis on the inference speed of CPU and mobile GPU will be detailed in the next subsection.

If compared to object detectors running at 30~100 fps, CSPResNeXt50 with PANet (SPP) achieves the best performance in AP, AP50 and AP75. They receive, respectively, 38.4%, 60.6%, and 41.6% detection rates. If compared to LRF [36] under the input image size 512×512, CSPResNeXt50 with PANet (SPP) outperforms ResNet101 with LRF by 0.7% AP, 1.5% AP50 and 1.1% AP75. If compared to object detectors running at 100~200 fps, CSPPeleeNet with EFM (SAM) boosts 12.1% and 4.1% AP50 at the same speed as Pelee [35] and CenterNet [43], respectively.

If compared to very fast object detectors such as ThunderNet [25], YOLOv3-tiny [28], and YOLOv3-tiny-PRN [33], the proposed CSPDenseNet Reference with PRN is the fastest. It can reach 400 fps frame rate, i.e., 133 fps faster than ThunderNet with SNet49. Besides, it gets 0.5% higher on AP50. If compared to ThunderNet146, CSP-PeleeNet Reference with PRN (3l) increases the frame rate by 19 fps while maintaining the same level of AP50.
4.5. Analysis

Computational Bottleneck. Figure 7 shows the BLOPS of each layer of PeleeNet-YOLO, PeleeNet-PRN and proposed CSPPeleeNet-EFM. The computational bottleneck of PeleeNet-YOLO occurs when the head integrates the feature pyramid, while the computational bottleneck of PeleeNet-PRN occurs on the transition layers of the PeleeNet backbone. As to the proposed CSPPeleeNet-EFM, it can balance the overall computational bottleneck, which reduces the PeleeNet backbone 44% computational bottleneck and reduces PeleeNet-YOLO 80% computational bottleneck. Therefore, we can say that the proposed CSPNet can provide hardware with a higher utilization rate.

Figure 7: Computational bottleneck of PeleeNet-YOLO, PeleeNet-PRN and CSPPeleeNet-EFM.

Memory Traffic. Figure 8 shows the size of each layer of ResNeXt50 and the proposed CSPResNeXt50. The CIO of the proposed CSPResNeXt (32.6M) is lower than that of the original ResNeXt50 (34.4M). In addition, our CSPResNeXt50 removes the bottleneck layers in the ResXBlock and maintains the same numbers of the input channel and the output channel, which is shown in Ma et al. [23] that this will have the lowest MAC and the most efficient computation when FLOPs are fixed. The low CIO and FLOPs enable our CSPResNeXt50 to outperform the vanilla ResNeXt50 by 22% in terms of computations.

Figure 8: Input size and output size of ResNeXt and CSPResNeXt.

Inference Rate. We further evaluate whether the proposed methods are able to be deployed on real-time detectors with mobile GPU or CPU. Experiments are based on NVIDIA Jetson TX2 and Intel Core i9-9900K. The inference rate on CPU is evaluated with the OpenCV DNN module. We do not adopt model compression or quantization for fair comparisons. The results are shown in Table 5.

<table>
<thead>
<tr>
<th>Model</th>
<th>Size</th>
<th>GPU</th>
<th>CPU</th>
<th>mGPU</th>
<th>AP_{50}</th>
</tr>
</thead>
<tbody>
<tr>
<td>SNet46-Thunder [25]</td>
<td>320</td>
<td>248</td>
<td>32</td>
<td></td>
<td>40.2</td>
</tr>
<tr>
<td>SNet49-Thunder [25]</td>
<td>320</td>
<td>267</td>
<td>47</td>
<td></td>
<td>33.7</td>
</tr>
<tr>
<td>YOLOv3-tiny [28]</td>
<td>416</td>
<td>330</td>
<td>54</td>
<td>37</td>
<td>33.1</td>
</tr>
<tr>
<td>YOLOv3-tiny-PRN [33]</td>
<td>416</td>
<td>400</td>
<td>71</td>
<td>49</td>
<td>33.1</td>
</tr>
<tr>
<td>CSPPeleeNet Ref.-EFM (SAM)</td>
<td>320</td>
<td>205</td>
<td>-</td>
<td>41</td>
<td>44.6</td>
</tr>
<tr>
<td>CSPDenseNet Ref.-EFM (SAM)</td>
<td>320</td>
<td>235</td>
<td>-</td>
<td>49</td>
<td>42.2</td>
</tr>
<tr>
<td>CSPPeleeNet Ref.-PRN (3l)</td>
<td>320</td>
<td>267</td>
<td>52</td>
<td>38</td>
<td>40.0</td>
</tr>
<tr>
<td>CSPPeleeNet Ref.-PRN</td>
<td>320</td>
<td>306</td>
<td>75</td>
<td>52</td>
<td>38.5</td>
</tr>
<tr>
<td>CSPDenseNet Ref.-PRN</td>
<td>320</td>
<td>387</td>
<td>95</td>
<td>64</td>
<td>35.4</td>
</tr>
<tr>
<td>CSPDenseNetb Ref.-PRN</td>
<td>320</td>
<td>400</td>
<td>102</td>
<td>73</td>
<td>34.2</td>
</tr>
</tbody>
</table>

If we compare the inference speed executed on CPU, CSPDenseNetb Ref.-PRN receives higher AP_{50} than SNet49-ThunderNet, and it also outperforms SNet49-ThunderNet by 55 fps, in terms of frame rate. On the other hand, CSPPeleeNet Ref.-PRN (3l) reaches the same accuracy level as SNet4-16-ThunderNet but significantly upgrades the frame rate by 20 fps on CPU.

If we compare the inference speed executed on mobile GPU, our proposed EFM can greatly reduce the memory requirement when generating feature pyramids, it is definitely beneficial to function under the memory bandwidth restricted mobile environment. For example, CSPPeleeNet Ref.-EFM (SAM) can have a higher frame rate than YOLOv3-tiny, and its AP_{50} is 11.5% higher than YOLOv3-tiny. For the same CSPPeleeNet Ref. backbone, although EFM (SAM) is 62 fps slower than PRN (3l) on GTX 1080ti, it reaches 41 fps on Jetson TX2, 3 fps faster than PRN (3l), and with 4.6% AP_{50} increase.

5. Conclusion

We have proposed the CSPNet that enables state-of-the-art methods such as ResNet, ResNeXt, and DenseNet to be light-weighted for mobile GPUs or CPUs. One of the main contributions is that we have recognized the redundant gradient information problem that results in inefficient optimization and costly inference computations. We have proposed to utilize the cross-stage feature fusion strategy and the truncating gradient flow to enhance the variability of the learned features within different layers. In addition, we have proposed the EFM that incorporates the Maxout operation to compress the features maps generated from the feature pyramid, which largely reduces the required memory bandwidth and thus the inference is efficient enough to be compatible with edge computing devices. Experimentally, we have shown that the proposed CSPNet with the EFM significantly outperforms competitors in terms of accuracy and inference rate on mobile GPU and CPU for real-time object detection tasks.
References


