SpotPatch: Parameter-Efficient Transfer Learning for Mobile Object Detection

Keren Ye*[1][0000−0002−7349−7762], Adriana Kovashka[1][0000−0003−1901−9660], Mark Sandler[2][0000−0003−0352−6051], Menglong Zhu[2][0000−0003−4796−6235], Andrew Howard[2], and Marco Fornoni[2][0000−0001−5538−8012]

[1] University of Pittsburgh, Pittsburgh PA 15260, USA
[2] Google Research

Abstract. Deep learning based object detectors are commonly deployed on mobile devices to solve a variety of tasks. For maximum accuracy, each detector is usually trained to solve one single specific task, and comes with a completely independent set of parameters. While this guarantees high performance, it is also highly inefficient, as each model has to be separately downloaded and stored. In this paper we address the question: can task-specific detectors be trained and represented as a shared set of weights, plus a very small set of additional weights for each task? The main contributions of this paper are the following: 1) we perform the first systematic study of parameter-efficient transfer learning techniques for object detection problems; 2) we propose a technique to learn a model patch with a size that is dependent on the difficulty of the task to be learned, and validate our approach on 10 different object detection tasks. Our approach achieves similar accuracy as previously proposed approaches, while being significantly more compact.

1 Introduction

Mobile object detection models are fundamental building blocks for daily-used mobile applications. For example, face detectors are used for locking/unlocking the latest generation phones and for building social apps such as Snapchat. In the early years, most computer vision models were deployed on servers, which meant that images had to be sent from the device to the server and that users had to wait for the server responses. This process was sensitive to network outages, provided on-device latency that was often not tolerable, and burdened the server clusters with high loads of client requests.

With the advance of mobile hardware technology, on-device computation became more and more affordable. Meanwhile, advances in neural network architectures made model inference increasingly more efficient. On the one hand, MobileNets [1,2,3] optimize the network architecture by decomposing convolutions into more efficient operations. Such designs provide general and compact backbones for mobile inference. On the other hand, one-stage detection architectures such as SSD [4] and Yolo [5] provide mobile-friendly detection heads.

* Work partially done during an internship at Google.

Fig. 1: **The challenge of updating a mobile object detector.** Suppose a general-purpose object detector is already deployed on-device. In a naive setting, adding support for detecting new entities would require downloading a completely separate model, with large network costs. Our goal is to reduce the network costs by *"patching"* the existing model to also support the new entities.

Due to the combination of the above advancements, object detection models are now massively being moved from server-side to on-device. While this constitutes great progress, it also brings new challenges (see Fig. 1). Specifically, multiple isolated models are often downloaded to perform related tasks. Suppose that a well-performing mobile model is downloaded for general-purpose object detection, with 10MB data traffic costs. Suppose then that the user requests an additional functionality that requires detecting new entities, like faces or barcodes: this will naively require to download a new model, with an extra 10MB data cost. Each time a new task will need to be supported, the user and the network operator will incur an additional 10MB data cost. The question is then: can we instead "patch" the previously downloaded general-purpose object detector to solve also the new tasks? If so, how can we minimize the size of the "patch" while maintaining high accuracy on the new task? To answer these questions, we studied two experimental scenarios that mimic two practical use-cases for "patching" mobile detection models:

1. Adapting an object detector to solve a new task.
2. Updating an existing model, whenever additional training data is available.

To learn the model patch, we propose an approach simultaneously optimizing for accuracy and footprint (see Fig. 2 for an overview): 1) for each layer, we minimize the size of the patch by using a 1-bit representation of the weight residuals; 2) we employ a gating mechanism to selectively patch only important layers, while reusing the original weights for the remaining ones. We evaluate our problem on ten different object detection tasks, using an experimental setup similar to [6], which we refer to as "Detection Decathlon". We also showcase our method's ability to efficiently update the model when new data becomes available. To the best of our knowledge, this is the first systematic study of parameter-efficient transfer learning techniques on object detection tasks.

2 Related Work

The most relevant approaches for our work fall into three main categories: (1) Model footprint reduction, aimed at reducing the number of trainable param-

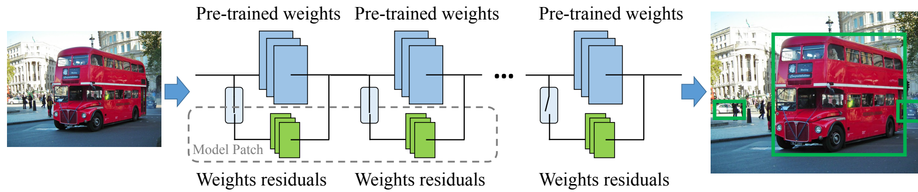


Fig. 2: **SpotPatch: demonstrating the gating mechanism.** During training, our model optimizes both the detection performance and the number of patched layers by opening, or closing per-layer gates. During deployment, only the *opened routes* constitute the model patch. We use a 1-bit representation for the weights residuals, significantly reducing the patch footprint.

eters, or the bit-size representation for each parameter. (2) Dynamic routing, adapting the network architecture at training time, based on well-designed cost functions. (3) Transfer learning and domain adaptation, ensuring representation transferability across tasks and domains.

Reducing the model footprint works involve various approaches that directly reduce the model size. Instead of fine-tuning the whole network [7] or the last few layers [8,9,10], [11] proposed to learn a set of parameters dispersed throughout the network. They experimented with updating batch normalization and depthwise convolution kernels, resulting in small model patches providing high classification accuracy. Similar findings were also published in [12].

Quantization is a technique commonly used in on-device models to reduce model size. Post-training quantization [13] can be applied to quantize a pre-trained floating-point precision model, while quantization-aware training [14,15,16,17] ensures that the quantization effects are already modeled at training time.

Ideas inspired by the quantization literature are used in transfer and multi-task learning to reduce the footprint of the model for the target tasks. For example, in [18], learned binary masks are element-wise multiplied to the source kernels to obtain the convolutional kernels to be used on the target task. In [19], the binary masks are augmented with floating-point scalars to produce an affine transformation of the kernel. Our method learns similar kernel masks, but we go one step further by automatically selecting the subset of layers to patch.

Dynamic routing works adapt the network structure to optimize a pre-defined objective. [20] trained a gating network selecting a sparse combination of experts based on input examples. [21] created a model allowing selective execution. In their setting, given an input, only a subset of neurons is executed. [22] reduced the number of ResNet [23] layers by bypassing residual blocks using a gating mechanism. [24] built a dynamic routing network to choose between using either the pre-trained frozen blocks, or the re-trained blocks.

Our approach differs from all these studies in that: (1) our dynamic model architecture is conditioned on dataset rather than on input; (2) we optimize (reduce) the number of patched layers; and (3) one of the route types, in our design, is specialized to use binary weights to further reduce footprint.

Transfer learning and domain adaptation works study the ability of models to transfer across tasks, in terms of achieving optimal accuracy in novel situations. In transfer learning, the target label set might differ from the source task. In domain adaptation, the classes may be the same, but have inherently different distributions. Some transfer and adaptation techniques minimize the discrepancy between tasks or domains in terms of the feature representation [25,26,27,28,29,30,31,32,33,34]. If the classes are the same, the deviation between classifier weights across domains can be minimized [35,36]. One of the most established recent benchmarks for transfer of classification networks is the Visual Decathlon Challenge [6], in which a decathlon-inspired scoring function is used to evaluate how well a single network can solve 10 different classification tasks. In the object detection literature, [37] proposes to use a source domain to generate boxes of different levels of class-specificity, to transfer to a target domain where only image-level labels are available. *In contrast to optimizing accuracy despite limited data in the target domain,* as is often the objective of domain adaptation, in this work we are concerned with preserving high accuracy while *minimizing the footprint* of the task-specific model patches.

3 Approach

To simplify notation, we assume a deep neural network $\mathcal{M}$ of depth N is composed of a set of layers represented by their weights $\boldsymbol{\theta} = \{\mathbf{W}_1, \dots, \mathbf{W}_N\}$, plus an activation function Φ. The transformation computed by the network is represented using Eq. 1, where $\boldsymbol{x}$ is the input and $\boldsymbol{z}_i$ denotes the i-th hidden state.

$$\begin{cases} \boldsymbol{z}_i = \Phi(\mathbf{W}_i \boldsymbol{z}_{i-1}), \quad \boldsymbol{z}_0 = \boldsymbol{x} \\ \mathcal{M}(\boldsymbol{x}) = \mathbf{W}_N \boldsymbol{z}_{N-1} \end{cases} \tag{1}$$

To adapt $\mathcal{M}$ to solve a new task, we seek a task-specific parameter $\boldsymbol{\theta}' = \{\mathbf{W}'_1, \dots, \mathbf{W}'_N\}$ that optimizes the loss on the target dataset. In addition, since we do not want to fully re-learn $\boldsymbol{\theta}'$, we look for a transformation with minimum cost (measured in terms of footprint) to convert the original $\boldsymbol{\theta}$ to $\boldsymbol{\theta}'$. Assume the transformation function can be expressed as $\boldsymbol{\theta}' = f(\boldsymbol{\theta}, \boldsymbol{\gamma})$ where $\boldsymbol{\gamma}$ is an additional set of parameters for the new task, and f is a function that combines the original parameters $\boldsymbol{\theta}$ with the parameter "patch" $\boldsymbol{\gamma}$ (as a very simple example, through addition). Our goal is to reduce the bit size of $\boldsymbol{\gamma}$. In our experiments, we use relative footprint $\frac{\text{bitsize}(\boldsymbol{\gamma})}{\text{bitsize}(\boldsymbol{\theta})}$ and patch size bitsize($\boldsymbol{\gamma}$) as footprint metrics.

We propose two approaches to compress the patch $\boldsymbol{\gamma}$, namely *task-specific weight transform* and *spot patching.* The former is inspired by the early Adaptive-SVM approaches such as [38,36], their Deep Neural Network counterparts such as [19], as well as quantization methods [15] and the low-rank representations [39]. The latter is inspired by the dynamic routing approaches such as [24] and channel pruning methods such as [40,41].

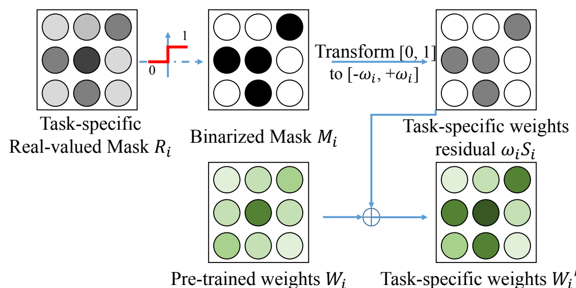


Fig. 3: **The weight patching model.** The pre-trained weights are augmented using a task-specific scaled sign matrix. For deployment, only the binary masks $\mathbf{M}_i$ and the scaling factors ω_i are stored in the model patch.

3.1 Task-specific weight transform

We denote the pre-trained weights by $\boldsymbol{\theta} = \{\mathbf{W}_1, \ldots, \mathbf{W}_N\}$ and define the task-specific weight trasformation as:

$$\mathbf{W}_i' = \mathbf{W}_i + \omega_i \mathbf{S}_i \tag{2}$$

where $\mathbf{S}_i$ is a 1-bit matrix of the same shape as $\mathbf{W}_i$ containing $\{-1, +1\}$ and ω_i is a scaling factor. For implementation convenience we use $\mathbf{M}_i$ as a 1-bit mask tensor with values in $\{0, 1\}$, and define $\mathbf{S}_i = \mathbf{1} - 2\mathbf{M}_i$. This formulation is equivalent to Eq. 2 in [19], with k_0 set to 1, and k_2 set to $-2k_1$. The reason we instead chose our formulation is that we empirically found the learned k_2 in [19] to be roughly distributed as $-2k_1$. We thus directly formulated Eq. 2 as learning a properly scaled zero-centered residual.

The incremental footprint of the model in Eq. 2 is $\{\omega_i, \mathbf{M}_i |\ i \in \{1, \ldots, N\}\}$, or roughly 1-bit per model weight, with a negligible additional cost for the per-layer scalar ω_i. To learn the 1-bit mask $\mathbf{M}_i$, we follow the same approach as [18] and [19]. We define the process in Eq. 3, and illustrate it in Fig. 3. During training, real-valued mask variables are maintained ($\mathbf{R}_i$ in Eq. 3) and differentiable binarization is applied to learn the masks. The binarization function is a hard thresholding function, and its gradient is set to that of the *sigmoid* function σ. After training, only the binarized masks and the per-layer scaling factors are used to deploy the model patch.

$$\mathbf{M}_i = \text{Binarize}(\mathbf{R}_i) \tag{3}$$

3.2 Spot patching

Assuming a 32-bit float representation, the task-specific weight transform produces patches 1/32 the size of the original network, regardless of the target dataset. However, our intuition is that the difficulty of adapting a model should also depend on the target dataset, so the footprint should vary for different tasks.

We design a gating mechanism to adapt the model complexity to different tasks. The process is defined in Eq. 4, and Fig. 2 shows the idea. Simply speaking, we add a *gate* g_i for each network layer. The layer uses the original pre-trained weights if the gate value is 0; otherwise, the layer uses weight transform to update the parameters. The benefit of using the gating indicator g_i is that it allows to search for a task-specific subset of layers to patch, *rather than patching all layers*. Compared to patching the whole network, it reduces the patch footprint to $\boldsymbol{\gamma} = \{\omega_i, \mathbf{M}_i | \ i \in \{1, \dots, N\}$ and $g_i = 1\}$.

$$\mathbf{W}'_i = \mathbf{W}_i + \underbrace{g_i}_{\text{gating}} \omega_i(\mathbf{1} - 2\mathbf{M}_i) \tag{4}$$

$$g_i = \text{Binarize}(f_i) \tag{5}$$

To learn g_i, we simply use the same differentiable binarization trick as for learning the $\mathbf{M}_i$. In Eq. 5, f_i is real-valued, and it is only used during training. To force the number of patched layers to be small, we minimize the number of patched layers $\sum_{i=1}^{N} g_i$ in the training loss (see next Section 3.3).

SpotPatch gating module design follows the same vein of dynamic routing approaches, especially [24]. The difference with respect to [24] lies in the fact that SpotPatch applies binary quantization to the tuning route (Sec. 3.1), greatly reducing the footprint. On the one hand, SpotPatch gating module can be seen as a simpler version of [24], in that we use the same differentiable binarization trick for both generating the binary masks, and directly optimizing the gating variables. On the other hand, our loss function explicitly minimizes the number of patched layers (see next Section 3.3), hence delivering a task-adaptive footprint, rather than a fix-sized one as in [24]. For example, [24] would provide a 300%, or 1000% footprint increase (they provided two models) to solve 10 classification tasks, while our method only requires an extra 35% footprint to solve 9 additional object detection tasks.

3.3 Our final task-adaptive detector

Our final model on a new task is similar to Eq. 1, with the parameters replaced by Eq. 4. During training, we use floating-point numbers and differentiable binarization (Eq. 3 and Eq. 5). During deployment, bit representations are used to efficiently encode the learned patch. Since the Batch Normalization layers did not constitute much of the footprint, we also trained task-specific Batch Normalization layers in addition to the convolutional weight residuals ([19] also patches BN layers). We use Eq. 6 to optimize the task-specific patch $\boldsymbol{\gamma}$:

$$L(\boldsymbol{\gamma}) = L_{det}(\boldsymbol{\gamma}) + \lambda_{sps} L_{sps}(\boldsymbol{\gamma}) + \lambda_{adp} L_{adp}(\boldsymbol{\gamma}) \tag{6}$$

where: $L_{det}(\boldsymbol{\gamma})$ is the detection loss optimizing both the class confidence scores and the box locations. $L_{sps}(\boldsymbol{\gamma}) = \sum_{i=1}^{N} g_i$ is the sparsity-inducing loss, pushing the number of patched layers to be small. Finally, $L_{adp}(\boldsymbol{\gamma}) = \sum_{i=1}^{N} \|\omega_i\|_2^2$ is the domain-adaptation loss forcing the scaling factors ω_i to be small, and thus $\boldsymbol{\theta}'$ to

be similar to $\boldsymbol{\theta}$. In this way, the pre-trained general-purpose source model serves as a strong prior for the task-specific target model. A similar loss to L_{adp} has been employed in prior domain adaptation works [36,38], to force the adapted weights to be small and train accurate models with limited data. We provide an ablation study for λ_{sps} in Sec. 4.4, while for λ_{adp} we use the constant value of 2E-5, as selected in preliminary experiments.

4 Experiments

We propose two scenarios for patching a mobile object detector, and design experiments to validate our model for both use-cases.

Adapting an object detector to solve a new task. For this scenario (Sec. 4.1), assume that we released a mature generic object detector to our users. However, if the user wants to perform a new unsupported detection task, we need to adapt the generic detector to solve the new task. For example, we may transform it into a product detector, or we may turn it into a pet detector. In this scenario, the challenge is to accurately and efficiently solve the new task.

Updating an existing detection model, whenever additional training data for the same task is available. For this use-case (Sec. 4.2), suppose we released an initial mobile model. We gather more training data after the model is released to the users. We then want to update the users models to improve the accuracy, while keeping the download byte size to be small. In this case, we assume that there is no significant shift in the data distribution, yet the initial model may be inaccurate because of the cold start.

In addition to the above settings, in Sec. 4.3 we consider the more practical 8-bit model quantization scenario. In Sec. 4.4 we study the effect of the sparsity constraint. In Sec. 4.5 we provide visualizations of the learned model patches.

Implementation details. Our experimental configuration is based on the SSD-FPNLite architecture [42,43], a practical mobile-friendly architecture. Slightly departing from the original configuration, we use the MobileNetV2 [3] architecture with 320×320 inputs.

Baselines. We compare with the following transfer learning baselines:

- Fine-Tuning [44]: This method fine-tunes the whole network. It provides a strong baseline in terms of accuracy, at the expense of a very large footprint.
- Tower Patch [44]: This method re-trains only the parameters in the detection head of the model. It is an adaptation and enhancement of the classifier last layer fine-tuning method, for object detection.
- Bn Patch [11], Dw Patch [11], Bn+Dw Patch [12,11]: These methods learn task-specific BatchNorm, Depthwise, or BatchNorm + Depthwise layers, respectively. They provide a patch with a tiny footprint.
- Piggyback [18]: Learns task-specific binary masks, and uses element-wise multiplication to apply the masks and obtain the task-specific convolutional kernels. Since the masks are binary, the 1-bit patch has a very low footprint.

Table 1: **Detection Decathlon datasets**. Number of samples and classes for each dataset used in our benchmark. Some datasets did not provide testing annotations; we thus evaluate on the held-out validation sets.

Name	#Trainval	#Eval	#Classes	Name	#Trainval	#Eval	#Classes
OID[45]	1,668,276	-	601	Face[46]	12,880	3,226	1
Birds[47]	3,000	3,033	200	Kitti[48]	6,981	500	2
Cars[49]	8,144	8,041	196	Pet[50]	3,180	500	37
COCO[51]	118,287	5,000	80	RPC[52]	5,400	600	200
Dogs[53]	12,000	8,580	120	VOC[54]	16,551	4,952	20

- WeightTrans [19]: This baseline also relies on binary masks. It applies affine transformations on the masks to get the task-specific kernels.

We reproduced all approaches using the SSD-FPNLite architecture for the following reasons: 1) most papers only report results for classification tasks, or for a few detection datasets; 2) implementations based on different network architectures make comparing the footprint challenging. In all our experiments we thus use our implementation of these methods. We did not re-implement and evaluate [24], as this approach creates weight residuals with the same bit-size as the original ones, i.e. full float kernels. It is thus not effective at significantly reducing the patch footprint, requiring as large as 3x or 10x (compared to the original model) additional footprints on the Visual Decathlon Challenge. As explained in next Section 4.1, we use finetuning as the baseline for parameter-*inefficient* transfer learning.

Metrics. To evaluate the detection performance, we use the mAP@0.5, which is the mean Average Precision over all classes, considering the generous threshold of IoU ≥ 0.5. For the footprint, we report the ratio $\frac{\text{bitsize}(\boldsymbol{\gamma})}{\text{bitsize}(\boldsymbol{\theta})}$ between the size of the additional parameters $\boldsymbol{\gamma}$ necessary to solve the new tasks, and the size of the original model $\boldsymbol{\theta}$. Sec. 4.1 and 4.2 consider 32-bit float models, in which the footprint metric is identical to that of Visual Decathlon Challenge [6]. In this case (32-bit float), the 1-bit binary masks (e.g., [18,19]) reduce the representation footprint by 32x. Sec. 4.3 considers instead the 8-bit representation, as it is more relevant for mobile applications.

4.1 Detection Decathlon

We use the OpenImages V4 [45] as the dataset for training the generic object detection model. This is a large-scale dataset featuring 1.74 million images of common objects from 600 different classes. Our fully-trained mobile model achieves 27.6% mAP@0.5 on the OpenImage V4 validation set. We then consider adapting the OpenImage V4 pre-trained model to nine additional detection tasks (see Tab. 1), and compare models on the basis of how well they solve all the problems (mAP@0.5), and how small is their footprint.

Below is a description of the detection datasets used in the Detection Decathlon problem. The Caltech-UCSD Birds [47] (Bird), Cars [49] (Car), and

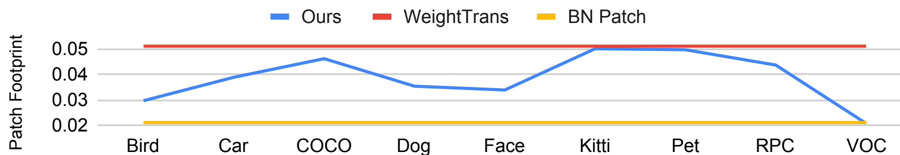


Fig. 4: **Per-dataset footprint.** SpotPatch footprint varies between that of Piggyback and that of BN Patch, depending on the complexity of the task.

Stanford Dogs [53] (Dog) are fine-grained categorization datasets. They provide center-view objects with bounding box annotations. The WiderFace [46] (Face) and Kitti [48] (Kitti) are human-related datasets. The former features human faces in different contexts and scales; the latter features vehicles and pedestrians for self-driving studies. The Oxford-IIIT Pet [50] (Pet) and Retail Product Checkout [52] (RPC) require both the fine-grained classification as well as localization. They involve many categories that appear in different locations and scales. Finally, the Pascal VOC [54] (VOC) and COCO [51] (COCO) are common object detection datasets. The class labels defined in them are subsets of those in OpenImage V4.

Different from the Visual Decathlon challenge, we assume the performance and footprint of the model on the original task (OpenImages V4) are unchanged, and only compare the models accuracy and footprint on the remaining nine tasks. We refer to the above problem as the *Detection Decathlon* problem. Similarly to [6] we also provide a decathlon-inspired scoring function to evaluate the performance of each method on the benchmark:

$$Score = 10000 \frac{1}{D} \sum_{d=1}^{D} \left(\frac{|s_d - b_d|^+}{1 - b_d} \right)^2 \tag{7}$$

where: the score s_d is the mAP of the considered approach on the d task; b_d is the score of a strong *baseline* on the d task; 10,000 is the maximum achievable score, and D is the total number of tasks to be solved. Similarly to [6], we select b_d to be the mAP of Fine-Tuning on task d, and normalize it so that its total score on the benchmark is 2,500. Specifically we set: $b_d = 2\ \text{mAP}_d(\text{Fine-Tuning}) - 1$.

To compare efficiency and effectiveness of different methods using one single metric, we finally report the *Score/Footprint* ratio [19]. For a given method, this metric practically measures the performance achieved for every Mb of footprint.

Tab. 2 and Fig. 4 shows our main results. Our first observation is that *patching an object detector is more challenging than patching a classifier.* Notably, none of the tested methods matches the Fine-Tuning Score, or mAP. The enhanced last layer fine-tuning method, Tower Patch, only achieves 33% of the Fine-Tuning score. Patching dispersed bottleneck layers provides reasonable improvements. For example in terms of Score, Bn Patch, DW Patch, and Bn+Dw Patch are 10.0%, 8.6%, and 22.4% relatively better than Tower Patch, and they all provide less than 0.50 footprints. However, the gap with

Table 2: **Detection Decathlon**. Footprint, per-dataset mAP, Average mAP, Score and Score / Footprint for each method. Best method (other than fine-tuning) in **bold**, second-best underlined. High score, low footprint, and high score/footprint ratio is good. Ours is the most parameter-efficient method in terms of Score/Footprint. It achieves mAP and Score comparable to the most accurate approach (WeightTrans 65.2%), with a 24% reduction in footprint.

Method	Footprint	Bird	Car	COCO	Dog	Face	Kitti	Pet	RPC	VOC	Average mAP	Score	Score / Footprint
Fine-Tuning	9.00	40.8	90.4	39.7	68.5	35.6	71.9	90.9	99.5	68.3	67.3	2500	278
Tower Patch	0.35	10.0	25.5	31.4	21.2	29.1	49.7	66.1	87.6	70.6	43.5	827	2362
Bn Patch	**0.19**	22.6	71.6	30.2	47.8	26.0	50.7	80.6	92.0	**71.1**	54.7	910	4789
Dw Patch	0.34	22.6	69.2	30.7	43.6	26.4	52.1	80.0	92.6	70.8	54.2	898	2642
Bn+Dw Patch	0.50	27.3	80.9	31.0	52.7	28.0	53.1	83.3	95.7	70.6	58.1	1012	2023
Piggyback	0.30	32.2	87.5	32.4	60.8	28.6	57.4	87.7	97.0	66.0	61.1	1353	4509
WeightTrans	0.46	**36.6**	**90.3**	**37.2**	**66.6**	**30.6**	**65.3**	**90.5**	98.7	70.7	**65.2**	**1987**	4319
Ours	0.35	35.8	89.8	36.6	63.3	30.1	64.0	90.3	**98.9**	70.6	64.4	1858	**5310**

respect to Fine-Tuning is still large. They only maintain less than 40.5% of the Fine-Tuning Score. The kernel quantization methods Piggyback and WeightTrans maintain at least 54.1% of the Fine-Tuning Score, while keeping the footprint below 0.46. SpotPatch achieves comparable performance to WeightTrans at only a 0.35 footprint. It also maintains 74.3% of the Fine-Tuning Score.

Our approach provides the best tradeoff by being parameter-efficient and yet accurate, as measured by the Score/Footprint ratio. This is achieved by learning patches with a task-adaptive footprint, resulting on average in a 24% footprint reduction with respect to WeightTrans*, with only minor loss in accuracy.*

4.2 Model updating

For this experiment, we use the COCO dataset [51]. First, we trained detection models (initialized from ImageNet[55] pre-trained model) on 10%, 20%, 40%, and 80% of the COCO training set. These models achieved 20.9%, 24.2%, 31.4%, and 35.7% mAP@0.5 on the COCO17 validation set, respectively. Then, we applied different patching approaches to update these imprecise models, using 100% of the COCO data. We then compared mAP@0.5 of the patched models, as well as the resulting patch footprints.

Tab. 3 shows the results. Similar to the Detection Decathlon, we observe that none of the tested approaches is able to achieve the same mAP as fine-tuning. Ours is the only method that can adapt the footprint according to the source model quality and the amount of new training data: At 10% training data, we achieve comparable mAP as WeightTrans (32.0% v.s. 32.7%) at a comparable footprint (5.04% v.s. 5.15%). However, when more data is available, the patch

Table 3: **Model updating**. Our method achieves comparable mAP and smaller footprint than the strongest baseline. The percentages (e.g. 10%) indicate the amount of data to train the source model. The best method is shown in **bold**, second-best underlined. Only one footprint number is shown for the baseline methods because they generate the same footprint regardless of the task.

Method	Footprint (%)				mAP (%)			
	10%	20%	40%	80%	10%	20%	40%	80%
Fine-Tuning	100.0				37.6	37.6	38.2	38.5
Tower Patch	3.85				24.5	26.7	32.4	35.8
Bn Patch	**2.08**				26.2	28.1	33.0	35.8
Dw Patch	3.76				25.9	27.8	33.1	**36.0**
Bn+Dw Patch	5.59				26.8	28.7	33.4	35.9
Piggyback	3.32				26.4	28.3	32.0	35.3
WeightTrans	5.15				**32.7**	**32.4**	**34.8**	35.9
Ours	5.04	4.98	4.21	**2.08**	32.0	31.4	34.2	35.8

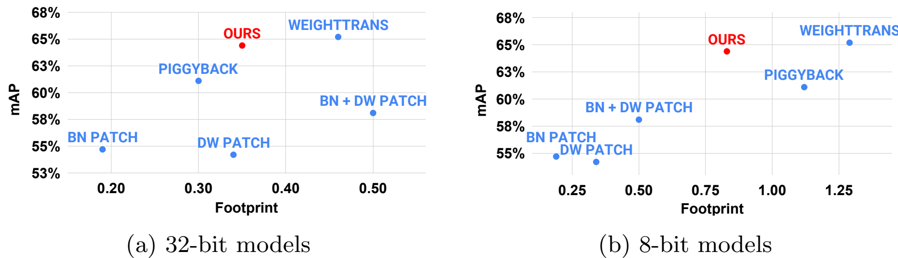
footprint generated by our approach is smaller than WeightTrans (2.08% v.s. 5.15%), while accuracy remains comparable (35.8% v.s. 35.9%).

To summarize, our method can effectively adapt the patch footprint to the amount of new data to be learned by the patch, while maintaining high accuracy.

4.3 Accuracy-footprint tradeoff in 8-bit models

To compute the footprint, both Sec. 4.1 and 4.2 account for the binary mask size as 1/32 of the float kernel size. This convention is widely accepted by the participants in the Visual Decathlon Challenge. However, in practical mobile applications, quantization-aware training [14,15,16,17] is often used to train a model using 8 bits per weight – i.e., reduce the model size by 4x, without losing accuracy. In the 8-bit model scenario, the relative footprint gains achieved by binary masks are thus 4x smaller than in the 32-bit model scenario. We estimate the footprint of 8-bit models in the Detection Decathlon and assume quantization-aware training does not significantly hurt the detection performance [15]. I.e., we did not train the 8-bit models but assume the mAP to be roughly the same as the 32-bit counterpart. To compare with the relative gains in the 32-bit scenarion, we show both of them side-by-side in Fig. 5.

As shown in Fig. 5, in the 8-bit scenario our method becomes more parameter-efficient than Piggyback and WeightTrans. The reason lies in the fact that ours is the only mask-based approach to explicitly minimize the number of masks in each patch. In Tab. 4, our model is thus as much as 26% and 36% more parameter-efficient than Piggyback and WeightTrans, respectively (0.83 v.s. 1.12, 1.29). Our method would save additional 0.9Mb in network costs compared to WeightTrans *per download*. Please note that while adding more tasks does not directly translate into mAP losses, footprint gains keep cumulating. In practical mobile application this effect would be amplified, as the same patch would need to be downloaded as many times as there are users. We thus argue that



(a) 32-bit models (b) 8-bit models

Fig. 5: **Detection Decathlon: mAP v.s. Footprint.** We show the object detection mAP versus footprint, for both 32-bit and 8-bit models. The x-axis denotes the patch footprint for the Detection Decathlon while the y-axis indicates the detection performance measured by mAP@0.5. We expect a good model to have both a high mAP and a low footprint.

Table 4: **Detection Decathlon: mAP and patch size**. The patch size is measured by the expected download bytes to solve the additional nine tasks. It depends on both the model backbone and the method. Excluding the final layer in the classification head, the size of a single Mobilenet-v2 SSD-FPNLite detector is 7.99Mb for a 32-bit model, and 2.00Mb for an 8-bit model.

	Type	Fine-Tuning	Tower Patch	Bn Patch	Dw Patch	Bn+Dw Patch	Piggy back	Weight Trans	Ours
mAP (%)	-	67.3	43.5	54.7	54.2	58.1	61.1	65.2	64.4
Footprint	32-bit	9.00	0.35	0.19	0.34	0.50	0.30	0.46	0.35
	8-bit						1.12	1.29	0.83
Patch size	32-bit	71.9Mb	2.77Mb	1.50Mb	2.71Mb	4.02Mb	2.39Mb	3.70Mb	2.79Mb
	8-bit	18.0Mb	692Kb	374Kb	677Kb	1.01Mb	2.25Mb	2.58Mb	1.66Mb

in practical mobile applications the 36% footprint reduction achieved by our method over WeightTrans, with only a 0.8% average mAP loss (64.4% v.s. 65.2%), constitutes a *significant improvement* over WeightTrans.

To summarize, in practical 8-bit scenarios our method can potentially reduce WeightTrans *footprint by 36%, with only a negligible loss in performance. It is also the most parameter-efficient mask-based method.*

4.4 Impact of the sparsity constraint

Next, we show the tradeoff between footprint and performance can further be selected by tuning λ_{sps}. We perform a study on the Detection Decathlon tasks: we vary λ_{sps} while keeping all other hyper-parameters the same.

Tab. 5 shows the results. We observed that the λ_{sps} has a direct impact on the percentage of patched layers and the patch footprint. In general, a large λ_{sps} value forces the footprint to be small, while a small λ_{sps} leads to a more accurate model. If we only use a small value (λ_{sps}=1.00E-05), the method still patches the

Table 5: **Impact of sparsity constraint**. We show the percentage of patched layers, relative footprint, and mAP regarding models trained using different λ_{sps}.

	Patched layers (%)			Footprint (%)			mAP (%)		
	1E-03	1E-04	1E-05	1E-03	1E-04	1E-05	1E-03	1E-04	1E-05
Bird	12.7	8.5	7.0	3.93	2.97	2.93	35.8	35.8	34.0
Car	12.7	25.4	29.6	3.39	3.89	3.92	89.3	89.8	89.7
COCO	19.7	42.3	53.5	4.12	4.62	4.76	35.5	36.6	36.8
Dog	22.5	19.7	87.3	4.14	3.54	5.09	64.9	63.3	66.0
Face	29.6	32.4	95.8	3.45	3.39	5.12	28.6	30.1	30.4
Kitti	31.0	69.0	81.7	4.08	5.01	4.77	58.8	64.0	63.3
Pet	22.5	53.5	52.1	4.24	4.97	4.32	90.9	90.3	90.5
RPC	42.3	53.5	67.6	4.67	4.37	4.68	98.3	98.9	98.9
VOC	0.0	0.0	2.8	2.08	2.08	2.54	70.5	70.6	69.9
	Avg Patched layers (%)			Sum Footprint (%)			Avg mAP (%)		
	21.4	33.8	53.1	34.1	34.8	38.1	63.6	64.4	64.4

majority of the model layers (53.1% in average), with a corresponding mAP of 64.4%. However, if we increase λ_{sps} to 1.00E-03, the proportion of patched layers is significantly reduced (21.4%) and mAP is only slightly reduced to 63.6%. We use λ_{sps}=1.00E-4 throughout the paper.

Tab. 5 also highlights how the patching difficulty on different tasks varies. For example, the VOC target task is the most similar to the OpenImages source task. Our method learned that updating the batch normalization layers is enough. It thus degraded to Bn Patch, as almost none of the layers were patched.

4.5 Visualization of model patches

Next, we shed light on the nature of the patches learned by SpotPatch, and the effect of the source/target task similarity. Fig. 6 shows the results for all the convolutional layers of the FPNLite model. For the Detection Decathlon problem, our approach patched fewer layers on the target tasks most similar to the source one, while modified more layers on the most dissimilar target tasks. Our model learned that it is okay to leave all of the convolutional layers unchanged, for the VOC dataset. In this case, it degraded to the Bn Patch approach, which tunes only the batch normalization layers. The reason, we argue, is that VOC labels are a subset of the OpenImages V4 labels. In contrast, our model patched 69.0% of the model layers for the Kitti task. Though the Kitti dataset features everyday objects such as vehicles and pedestrians, the appearance of these objects is significantly different from OpenImages V4 because they are captured by cameras mounted on cars for autonomous driving research.

Similar observations are made on the model updating. To patch the pre-trained models armed with 40% and 80% COCO information, our method patched 28.2% and 0.0% of the layers. However, for the imprecise and low-quality pre-trained models, for example, the models with 10% and 20% COCO information, our model patched 67.6% and 54.9% of the layers.

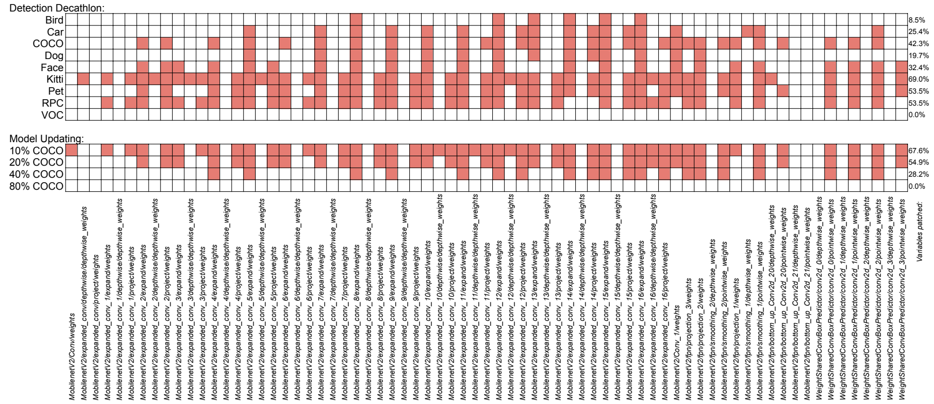


Fig. 6: **Patched layers for different tasks.** We emphasize our method learned different model patches based on the complexity of the tasks. In the figure, each row denotes a patching policy for a specific dataset, and each column represents the gating indicator of a particular layer. For each row, a red block means that the learned model patch includes the weights residuals of the specific kernel. White blocks indicate that the source kernel is reused as is, with no contribution to the footprint. We show the proportion of patched layers on the right.

The patched models also shared some common patterns. Our method did not patch the first few convolutional layers and the FPN upsampling layers for most tasks. We argue the reason is that these layers are responsible for recognizing fundamental visual features that can be shared across domains.

5 Conclusion

In this paper we drew the foundations for investigating parameter-efficient transfer learning in the context of mobile object detection. We introduced the Detection Decathlon problem, and provided the first systematic study of parameter-efficient transfer learning on this task. We proposed the SpotPatch approach, using task-specific weight transformations and dynamic routing to minimize the footprint of the learned patch. We also demonstrated how to use our technique for updating a pre-trained model. In all the considered benchmarks SpotPatch was shown to provide similar mAP as standard Weight-Transform, while being significantly more parameter-efficient. Additional potential gains were shown in the case of 8-bit quantization. We also noted how differently from classification benchmarks, none of the tested approaches was actually able to beat fine-tuning mAP, which calls for more work on the Detection Decathlon task.

References

1. Howard, A., Sandler, M., Chu, G., Chen, L.C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., Le, Q.V., Adam, H.: Searching for mobilenetv3. In: The IEEE International Conference on Computer Vision (ICCV). (2019)
2. Howard, A.G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., Adam, H.: Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861 (2017)
3. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: Mobilenetv2: Inverted residuals and linear bottlenecks. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
4. Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S., Fu, C.Y., Berg, A.C.: Ssd: Single shot multibox detector. In: European conference on computer vision, Springer (2016) 21–37
5. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition. (2016) 779–788
6. Rebuffi, S.A., Bilen, H., Vedaldi, A.: Learning multiple visual domains with residual adapters. In: Advances in Neural Information Processing Systems. (2017) 506–516
7. Cui, Y., Song, Y., Sun, C., Howard, A., Belongie, S.: Large scale fine-grained categorization and domain-specific transfer learning. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
8. Oquab, M., Bottou, L., Laptev, I., Sivic, J.: Learning and transferring mid-level image representations using convolutional neural networks. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2014)
9. Thomas, C., Kovashka, A.: Seeing behind the camera: Identifying the authorship of a photograph. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016)
10. Yosinski, J., Clune, J., Bengio, Y., Lipson, H.: How transferable are features in deep neural networks? In Ghahramani, Z., Welling, M., Cortes, C., Lawrence, N.D., Weinberger, K.Q., eds.: Advances in Neural Information Processing Systems 27. Curran Associates, Inc. (2014) 3320–3328
11. Mudrakarta, P.K., Sandler, M., Zhmoginov, A., Howard, A.: K for the price of 1. parameter efficient multi-task and transfer learning. In: International Conference on Learning Representations. (2019)
12. Guo, Y., Li, Y., Feris, R.S., Wang, L., Rosing, T.S.: Depthwise convolution is all you need for learning multiple visual domains. In: AAAI. (2019)
13. Tensorflow: (Post training quantization) `http://https://www.tensorflow.org/lite/performance/post_training_quantization`.
14. Baskin, C., Schwartz, E., Zheltonozhskii, E., Liss, N., Giryes, R., Bronstein, A.M., Mendelson, A.: Uniq: Uniform noise injection for non-uniform quantization of neural networks. arXiv preprint arXiv:1804.10969 (2018)
15. Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M., Howard, A., Adam, H., Kalenichenko, D.: Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
16. Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., Bengio, Y.: Quantized neural networks: Training neural networks with low precision weights and activations. The Journal of Machine Learning Research **18** (2017) 6869–6898

17. Tensorflow: (Quantization-aware training) https://github.com/tensorflow/tensorflow/tree/r1.15/tensorflow/contrib/quantize.
18. Mallya, A., Davis, D., Lazebnik, S.: Piggyback: Adapting a single network to multiple tasks by learning to mask weights. In: The European Conference on Computer Vision (ECCV). (2018)
19. Mancini, M., Ricci, E., Caputo, B., Rota Bulò, S.: Adding new tasks to a single network with weight transformations using binary masks. In: ECCV Workshop on Transferring and Adapting Source Knowledge in Computer Vision. (2018)
20. Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., Dean, J.: Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. arXiv preprint arXiv:1701.06538 (2017)
21. Liu, L., Deng, J.: Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In: Thirty-Second AAAI Conference on Artificial Intelligence. (2018)
22. Veit, A., Belongie, S.: Convolutional networks with adaptive inference graphs. In: The European Conference on Computer Vision (ECCV). (2018)
23. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016)
24. Guo, Y., Shi, H., Kumar, A., Grauman, K., Rosing, T., Feris, R.: Spottune: Transfer learning through adaptive fine-tuning. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2019)
25. Carlucci, F.M., D'Innocente, A., Bucci, S., Caputo, B., Tommasi, T.: Domain generalization by solving jigsaw puzzles. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2019)
26. Chen, Y., Li, W., Sakaridis, C., Dai, D., Van Gool, L.: Domain adaptive faster r-cnn for object detection in the wild. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
27. Ganin, Y., Ustinova, E., Ajakan, H., Germain, P., Larochelle, H., Laviolette, F., Marchand, M., Lempitsky, V.: Domain-adversarial training of neural networks. The Journal of Machine Learning Research **17** (2016) 2096–2030
28. Herath, S., Harandi, M., Fernando, B., Nock, R.: Min-max statistical alignment for transfer learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2019) 9288–9297
29. Long, M., Zhu, H., Wang, J., Jordan, M.I.: Unsupervised domain adaptation with residual transfer networks. In: Advances in Neural Information Processing Systems (NeurIPS). (2016) 136–144
30. Sun, Q., Liu, Y., Chua, T.S., Schiele, B.: Meta-transfer learning for few-shot learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2019) 403–412
31. Tzeng, E., Hoffman, J., Saenko, K., Darrell, T.: Adversarial discriminative domain adaptation. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2017)
32. Vu, T.H., Jain, H., Bucher, M., Cord, M., Perez, P.: Dada: Depth-aware domain adaptation in semantic segmentation. In: The IEEE International Conference on Computer Vision (ICCV). (2019)
33. Wang, T., Zhang, X., Yuan, L., Feng, J.: Few-shot adaptive faster r-cnn. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2019) 7173–7182

34. Yin, X., Yu, X., Sohn, K., Liu, X., Chandraker, M.: Feature transfer learning for face recognition with under-represented data. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. (2019) 5704–5713
35. Kovashka, A., Grauman, K.: Attribute adaptation for personalized image search. In: The IEEE International Conference on Computer Vision (ICCV). (2013)
36. Yang, J., Yan, R., Hauptmann, A.G.: Adapting svm classifiers to data with shifted distributions. In: Seventh IEEE International Conference on Data Mining Workshops (ICDMW 2007), IEEE (2007) 69–76
37. Uijlings, J., Popov, S., Ferrari, V.: Revisiting knowledge transfer for training object class detectors. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
38. Kovashka, A., Grauman, K.: Attribute adaptation for personalized image search. In: Proceedings of the IEEE International Conference on Computer Vision. (2013) 3432–3439
39. Li, Y., Gu, S., Gool, L.V., Timofte, R.: Learning filter basis for convolutional neural network compression. In: The IEEE International Conference on Computer Vision (ICCV). (2019)
40. Gordon, A., Eban, E., Nachum, O., Chen, B., Wu, H., Yang, T.J., Choi, E.: Morphnet: Fast & simple resource-constrained structure learning of deep networks. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2018)
41. Liu, Z., Li, J., Shen, Z., Huang, G., Yan, S., Zhang, C.: Learning efficient convolutional networks through network slimming. In: The IEEE International Conference on Computer Vision (ICCV). (2017)
42. Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., Murphy, K.: (Ssd-fpnlite architecture for object detection) https://github.com/tensorflow/models/blob/master/research/object_detection/samples/configs/ssd_mobilenet_v1_fpn_shared_box_predictor_640x640_coco14_sync.config.
43. Lin, T.Y., Goyal, P., Girshick, R., He, K., Dollar, P.: Focal loss for dense object detection. In: The IEEE International Conference on Computer Vision (ICCV). (2017)
44. Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., Fischer, I., Wojna, Z., Song, Y., Guadarrama, S., Murphy, K.: Speed/accuracy trade-offs for modern convolutional object detectors. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2017)
45. Kuznetsova, A., Rom, H., Alldrin, N., Uijlings, J., Krasin, I., Pont-Tuset, J., Kamali, S., Popov, S., Malloci, M., Duerig, T., et al.: The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. arXiv preprint arXiv:1811.00982 (2018)
46. Yang, S., Luo, P., Loy, C.C., Tang, X.: Wider face: A face detection benchmark. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2016)
47. Welinder, P., Branson, S., Mita, T., Wah, C., Schroff, F., Belongie, S., Perona, P.: Caltech-ucsd birds 200. (2010)
48. Geiger, A., Lenz, P., Urtasun, R.: Are we ready for autonomous driving? the kitti vision benchmark suite. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2012)
49. Krause, J., Stark, M., Deng, J., Fei-Fei, L.: 3d object representations for fine-grained categorization. In: Proceedings of the IEEE International Conference on Computer Vision Workshops. (2013)

50. Parkhi, O.M., Vedaldi, A., Zisserman, A., Jawahar, C.: Cats and dogs. In: The IEEE Conference on Computer Vision and Pattern Recognition (CVPR). (2012)
51. Lin, T.Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., Zitnick, C.L.: Microsoft coco: Common objects in context. In: European conference on computer vision, Springer (2014) 740–755
52. Wei, X.S., Cui, Q., Yang, L., Wang, P., Liu, L.: Rpc: A large-scale retail product checkout dataset. arXiv preprint arXiv:1901.07249 (2019)
53. Khosla, A., Jayadevaprakash, N., Yao, B., Fei-Fei, L.: Novel dataset for fine-grained image categorization. In: First Workshop on Fine-Grained Visual Categorization, IEEE Conference on Computer Vision and Pattern Recognition, Colorado Springs, CO (2011)
54. Everingham, M., Van Gool, L., Williams, C.K.I., Winn, J., Zisserman, A.: The pascal visual object classes (voc) challenge. International Journal of Computer Vision **88** (2010) 303–338
55. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE conference on computer vision and pattern recognition, Ieee (2009) 248–255