– Supplementary Material –
OBoW: Online Bag-of-Visual-Words Generation for Self-Supervised Learning

Spyros Gidaris[1], Andrei Bursuc[1], Gilles Puy[1], Nikos Komodakis[2], Matthieu Cord[1,3], Patrick Pérez[1]
[1]valeo.ai [2]University of Crete [3]Sorbonne Université

A. Comparing with MoCo for the same image augmentations

In the main paper we saw that the image augmentation techniques that we designed for our method have a strong positive impact on the quality of the learned representations. However, we stress that the performance improvement of our method over state-of-the-art instance-discrimination methods is not simply due a better mix of augmentations.

For example, in Table 1 we compare OBoW with MoCo v2, when the latter is implemented with the same image augmentations as those in the full solution of OBoW. We see that indeed, although the proposed augmentations also improve MoCo v2, our method is still significantly better, even in its vanilla version that employs simpler augmentations (i.e., only a single 160×160-sized crop). Therefore, the ability of OBoW to learn state-of-the-art representations is mainly due to its BoW-guided reconstruction formulation.

Method	EP	Few-shot $n=1$	Few-shot $n=5$	Linear
OBoW (vanilla)	80	42.11	62.44	45.86
OBoW (full)	80	**44.18**	**64.89**	**50.89**
MoCo v2	80	24.75	43.89	35.00
MoCo v2 (our aug.)	80	36.90	55.87	43.13

Table 1: Comparison with MoCo v2 for the same image augmentations. "EP": total number of epochs used for pre-training. The results are obtained by training ResNet18-based models on 20% of ImageNet, similar to Sections 4.1 and 4.2 of the main paper. "MoCo v2 (our aug.) " is a MoCo v2 model implemented with the same augmentations that we use in the full version of our work, i.e., two 160×160-sized crops plus five 96×96-sized patches.

B. Visualization of the vocabulary features

In Figures 1 and 2 we illustrate visual words from the `conv5` and `conv4` teacher feature maps of a ResNet50 OBoW model trained on ImageNet. Since we use a queue-based vocabulary that is constantly updated, for the visualizations we used the state of the vocabulary at the end of training. In order to visualize a visual word, we retrieve multiple image patches from images in the ImageNet training set and depict the 8 patches with the highest assignment score for that visual word. As it can be noticed, visual words encode high level visual concepts.

C. Implementation details

C.1. Image augmentation during pre-training

In Section 3.3 of the main paper, we described the two types of image crops that we extract from a training image in order to train the student network with them. In addition, beyond image cropping, similar to SimCLR [3], we also applied color jittering, color-to-grayscale conversion, Gaussian blurring and horizontal flipping as augmentation techniques.

All implementation details are provided in Section F in the form of PyTorch code.

C.2. Evaluation protocols in Section 4.1

To evaluate the quality of the learned representations, we use two protocols. **(1)** The first protocol consists in freezing the convnet and then training on its features 1000-way linear classifiers for the ImageNet classification task. The classifier is applied on top of the 512-dimensional feature vectors produced from the final global pooling layer of ResNet18. It is trained with SGD for 50 epochs using a learning rate of 10 that is divided by a factor of 10 every 15 epochs. The batch size is 256 and the weight decay $2\mathrm{e}{-6}$. For fast experimentation, we train the linear classifier with precached features extracted from the 224×224 central crop of the image and its horizontally flipped version. **(2)** For the second protocol, we use a few-shot episodic setting [14]. We choose 300 classes from ImageNet and run 200 episodes of 50-way few-shot classification tasks. Essentially, for each episode, we randomly select 50 classes from the 300 ones and, for each of these selected classes, n training examples and $m = 1$ test example (both randomly sampled from the validation images of ImageNet). For n, we use 1 and 5 examples corresponding to 1-shot and 5-shot classification settings, respectively. The m test examples per class are classified using a cosine-

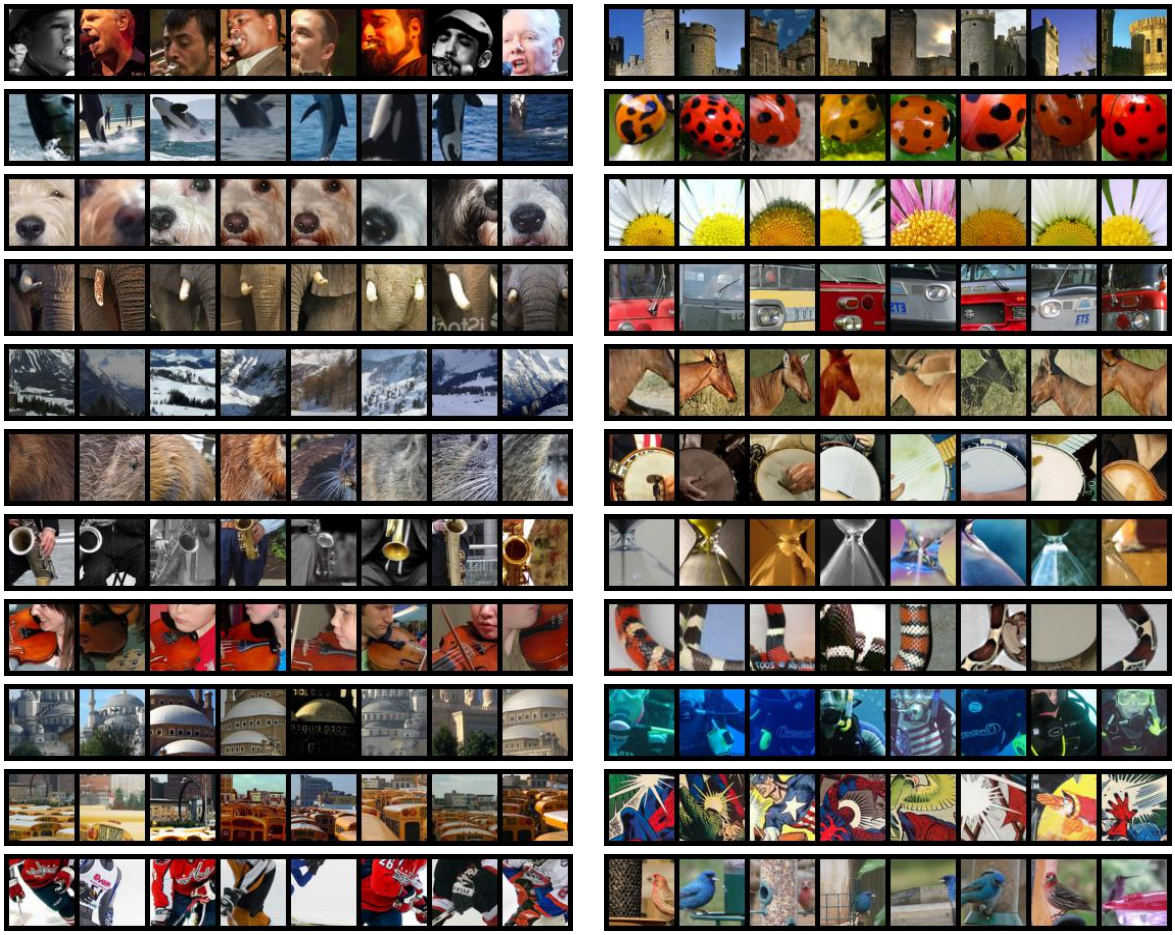

Figure 1: **Examples of visual-word members from the `conv5` layer of ResNet50.** The visualizations are created by using the state of the queue-based visual-words vocabulary at the end of training. For each visual word, we depict the 8 image patches retrieved from ImageNet with the highest assignment score for that word.

distance Prototypical-Networks [13] classifier applied on top of the frozen self-supervised representations. We report the mean accuracy over the 200 episodes. The purpose of this metric is to analyze the ability of the representations to be used for learning with few training examples. Furthermore, it has the advantage of not requiring tuning of any hyper-parameters, such as the learning rate of a linear classifier, the number of training steps, *etc*.

C.3. Self-supervised training on ImageNet

Here we provide implementation details for the pre-training of the ResNet50-based OBoW model that we use in Section 4.3 of the main paper. We present the full implementation of our method, which includes multi-scale BoWs from the `conv4` and `conv5` layers of ResNet50, and extraction of two crops of size 160×160 plus five patches of size 96×96 per training image. To extract BoW targets, we use $K = 8192$ as vocabulary size and we ignore the local feature vectors on the edge / border of the teacher's feature maps. The momentum coefficient α for the teacher updates is initialized at 0.99 and is annealed to 1.0 during training with a cosine schedule. Finally, the hyper-parameters κ and δ_{base} are set to 8 and $1/15$ respectively.

We train the model for 200 training epochs with SGD using $1\text{e}{-4}$ weight decay and 256-sized mini-batches. As a learning-rate schedule, we warm up the learning rate from 0 to 0.03 with linear annealing during the first 10 epochs and then, for the remaining 190 epochs, we decrease it from 0.03 to 0.00003 with cosine-based schedule. To train the model we use 4 Tesla V100 GPUs with data-distributed training (i.e., the mini-batch is divided across the 4 GPUs) while keeping the batch-norm statistics synchronized across all GPUs (i.e., use the `SyncBatchNorm` units of PyTorch).

C.4. Evaluation protocols in Section 4.3

Here we describe the evaluation protocols that we use in Section 4.3 of the main paper.

ImageNet linear classification. In this case, we evalu-

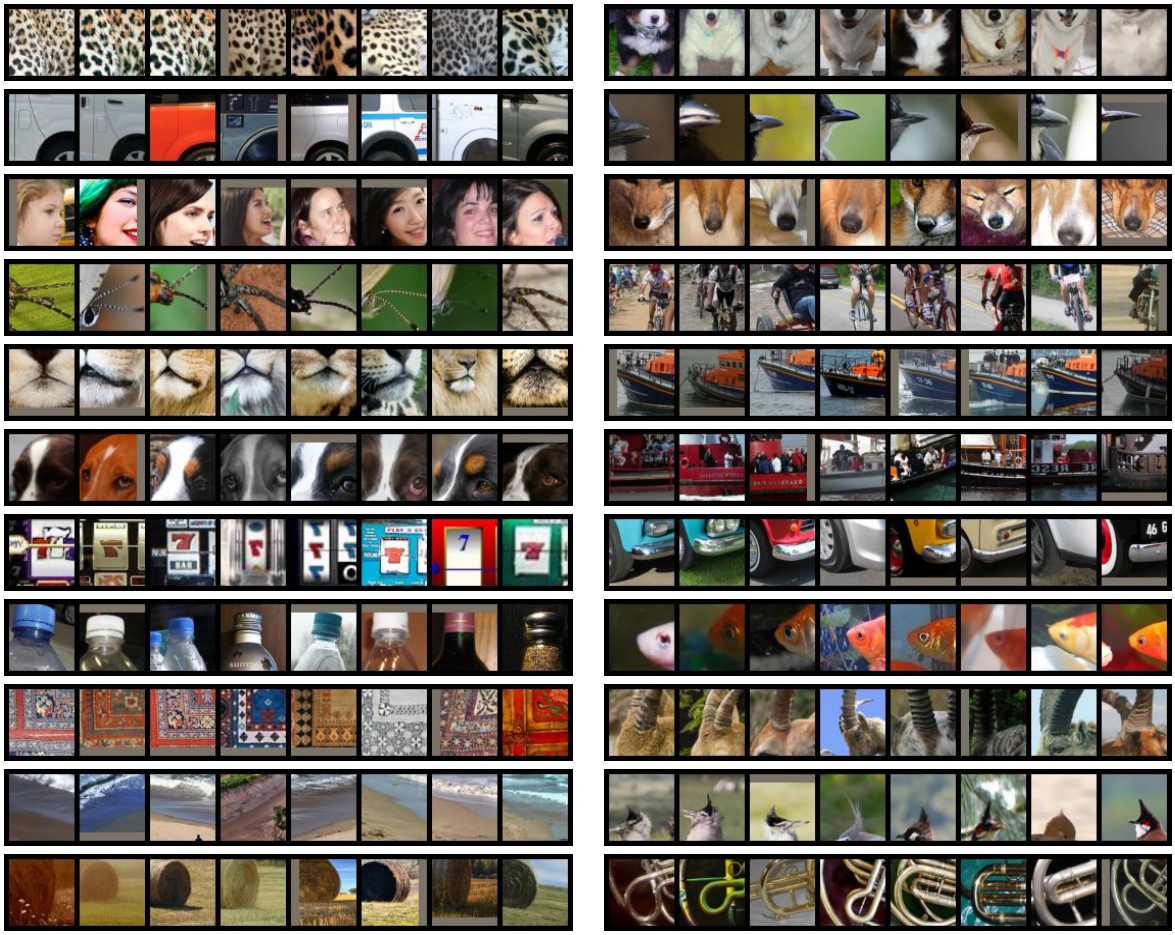

Figure 2: **Examples of visual-word members from the `conv4` layer of ResNet50.** The visualizations are created by using the state of the queue-based visual-words vocabulary at the end of training. For each visual word, we depict the 8 image patches retrieved from ImageNet with the highest assignment score for that word.

ate the performance on the 1000-way ImageNet classification task by applying a linear classifier on top of the 2048-dimensional frozen features of the `pool5` layer of ResNet50. We train the linear classifier using SGD for 100 training epochs with 0.9 momentum, 0 weight decay, 1024-sized mini-batches and cosine learning-rate schedule initialized at 10.0. We use the typical image augmentations used for the fully-supervised training of ResNet50 models on this dataset.

Places205 linear classification. For this protocol, we evaluate the performance on the 205-way Places classification task by applying a linear classifier on top of the 2048-dimensional frozen features of the `pool5` layer of ResNet50. We follow the guidelines of [6] and train the linear classifier using SGD for 14 training epochs and a learning rate of 0.01 that is multiplied by 0.1 after 5 and 10 epochs. The batch size is 256 and the weight decay is $1e{-}4$.

VOC07 linear classification with SVMs. Here we evaluate on the VOC07 classification task by training and testing linear SVMs on top of the 2048-dimensional frozen features of the `pool5` layer. To that end, we use the publicly available code for benchmarking self-supervised methods provided in [6] that trains the SVMs using the VOC07 train+val splits and tests them using the VOC07 test split.

Semi-supervised learning setting on ImageNet. For this semi-supervised setting, we fine-tune the self-supervised ResNet50 model (pre-trained on all ImageNet unlabelled images) on 1% or 10% of ImageNet labelled images. We use the same 1% and 10% splits as in SimCLR (i.e., we downloaded and use the split files of their official code release). We train using SGD with 256-sized mini-batches, 0 weight decay, and two distinct learning rates for the classification head and the feature extractor trunk network components respectively. Specifically, in the 1% setting, we use 40 epochs and the initial learning rates 0.5 and 0.0002 for the classification head and feature extractor trunk components, respectively, which are then multiplied by a factor of 0.2 after 24 and 32 epochs. In the 10% setting, we use 20

epochs and the initial learning rates 0.5 and 0.0002 for the classification head and feature extractor trunk components, respectively, which are multiplied by a factor of 0.2 after 12 and 16 epochs.

VOC object detection. Here we evaluate the utility of OBoW on a complex downstream task: object detection. We follow the setup considered in prior works [2, 5, 6, 7, 9]: we fine-tune the pre-trained OBoW with a Faster R-CNN [12] model using a ResNet50 backbone [8] (R50-C4 in Detectron2 [15]). We use the fine-tuning protocol and most hyper-parameters from He *et al*. [7]: fine-tune on `trainval07+12` and evaluate on `test07`. In detail, we train with mini-batches of size 16 across 4 GPUs for 24K steps, using `SyncBatchNorm` to fine-tune BatchNorm parameters, as well as inserting an additional BatchNorm layer for the RoI head after `conv5`, i.e., `Res5ROIHeadsExtraNorm` layer in Detectron2. The initial learning rate 0.01 is warmed-up with a slope of $1\mathrm{e}{-3}$ for 100 steps and then reduced by a factor of 10 after 18K and 22K steps. We report results for the final checkpoint averaged over 3 different runs.

D. On-line k-means vocabulary updates

As explained in the main paper, one of the explored choices for updating the vocabulary is to use online k-means after each training step. Specifically, as proposed in VQ-VAE [10, 11], we use exponential moving average for vocabulary updates. In this case, for each mini-batch, we compute the number n_k of feature vectors assigned to each cluster k and $\mathbf{m}_k$ the element-wise sum of all feature vectors assigned to cluster k and update

$$N_k \leftarrow \gamma N_k + (1-\gamma) n_k, \tag{1}$$

$$\mathbf{M}_k \leftarrow \gamma \mathbf{M}_k + (1-\gamma)\mathbf{m}_k, \tag{2}$$

with $\gamma = 0.99$. The k^{th} visual word of the vocabulary V satisfies $\mathbf{v}_k = \mathbf{M}_k / N_k$. A critical issue that arises in this case is that, as training progresses, the features distribution changes over time. The visual words computed by online k-means do not adapt to this distribution shift leading to extremely unbalanced cluster assignments and even to assignments that collapse to a single cluster. In order to counter this effect, we investigate two different strategies:

(a) Detection and replacement of rare visual words. In this case, for each visual word we keep track of the time of its most recent assignment as closest cluster centroid to a feature vector. If more than 1000 training steps have passed since then, then we replace it with a local feature vector randomly sampled with uniform distribution from the current mini-batch.

(b) Enforcing uniform assignments via Sinkhorn optimization. Let $\mathbf{x}_1, \ldots, \mathbf{x}_b$ be the b images of the current mini-batch and D be the $K \times B$ matrix ($B = h_\ell \times w_\ell \times b$) whose i^{th} row D_i contains the squared distances between all the local features in the mini-batch (across all images and spatial dimensions) and the i^{th} visual word: $D_i = \big[\|\mathrm{T}(\mathbf{x}_1)[1] - \mathbf{v}_i\|_2^2, \ldots, \|\mathrm{T}(\mathbf{x}_b)[h_\ell \times w_\ell] - \mathbf{v}_i\|_2^2,\big]$. Similarly to [1, 2], we compute the assignment codes by solving the regularised optimal transport problem $\min_{Q \in \mathcal{Q}} \sum_{i,j} Q_{i,j} D_{i,j} + \varepsilon\, Q_{i,j} \log Q_{i,j}$, where ε is a coefficient that controls the softness of the assignments. The set $\mathcal{Q}$ permits us to enforce uniform assignments among all the visual words and satisfies $\mathcal{Q} = \{Q \in \mathbb{R}_+^{K \times B} | Q\mathbf{1}_B = \frac{1}{K}\mathbf{1}_K, Q^\top \mathbf{1}_K = \frac{1}{B}\mathbf{1}_B\}$, where $\mathbf{1}_K$ and $\mathbf{1}_B$ are vectors of length K and B, respectively, with all entries equal to 1. We compute Q with the Sinkhorn algorithm [4] and use the resulting assignment codes for the on-line k-means updates and for the computation of the BoW targets.

E. Time and memory consumption

In Table 2, we provide the time and memory consumption of our method as well as of MoCo v2 and BYOL. We observe that OBoW achieves state-of-the-art results in less time ("Training time" row) than the competing methods. In terms of GPU memory consumption, with 256-sized mini-batches our method requires 15775Mb per GPU in a 4-GPU machine (or 8901Mb per GPU in a 8-GPU machine).

	Sup.	OBoW	MoCo v2	BYOL
Epochs	100	200	800	300
Measured with 256-sized mini-batches				
Time per epoch	1.00	3.91	1.58	3.47
Training time	1.00	7.82	12.64	10.41
Memory per GPU	1.00	2.00	1.13	1.72
ImageNet linear classification accuracy				
batch size = 256	76.5	73.8	71.1	-
batch size = 4096	-	-	-	72.5†

Table 2: Time and memory consumption relative to supervised training. "Sup." is the supervised ImageNet training. To measure the time and memory consumption, for all methods we used ResNet50-based implementations, 256-sized mini-batches and data-distributed training with 4 Tesla V100 GPUs. We measured the time consumption based on a single training epoch ("Time per epoch"). We also provide the projected time for the full training of a method ("Training time"), which is estimated based on the specified number of training epochs ("Epochs"). For OBoW, we used its full implementation. †: for BYOL we provide the time and memory consumption w.r.t. 256-sized mini-batches, but BYOL uses 4096-sized mini-batches to achieve the reported ImageNet classification accuracy. So, in reality BYOL has higher total GPU memory requirements.

F. PyTorch code of Image augmentations

Here we provide the PyTorch implementation of the image augmentations used in our work.

```python
import random
import torch
import torchvision.transforms as T
from PIL import ImageFilter

class CropImagePatches:
    """Crops from an image 3 x 3 overlapping patches."""
    def __init__(self, patch_size=96, patch_jitter=24, num_patches=5):
        self.patch_size = patch_size
        self.patch_jitter = patch_jitter
        self.num_patches = num_patches

    def __call__(self, img):
        _, height, width = img.size()
        split_per_side = 3
        offset_y = (height - self.patch_size - self.patch_jitter) // (split_per_side-1)
        offset_x = (width - self.patch_size - self.patch_jitter) // (split_per_side-1)

        patches = []
        for i in range(split_per_side):
            for j in range(split_per_side):
                y_top = i * offset_y + random.randint(0, self.patch_jitter)
                x_left = j * offset_x + random.randint(0, self.patch_jitter)
                y_bottom = y_top + self.patch_size
                x_right = x_left + self.patch_size
                patches.append(img[:, y_top:y_bottom, x_left:x_right])

        if self.num_patches < (split_per_side * split_per_side):
            indices = torch.randperm(len(patches))[:self.num_patches]
            patches = [patches[i] for i in indices.tolist()]

        return torch.stack(patches, dim=0)

class StackMultipleViews:
    def __init__(self, transform, num_views):
        self.transform = transform
        self.num_views = num_views

    def __call__(self, img):
        return torch.stack([self.transform(img) for _ in range(self.num_views)], dim=0)

class GaussianBlur:
    def __init__(self, sigma=[.1, 2.]):
        self.sigma = sigma

    def __call__(self, img):
        sigma = random.uniform(self.sigma[0], self.sigma[1])
        return img.filter(ImageFilter.GaussianBlur(radius=sigma))

normalize = T.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

# Define the transformations for extracting the central from of the original image.
transform_original_image = T.Compose([
    T.Resize(256),
    T.CenterCrop(224),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    normalize])

# Define the transformations for generating two 160x160-sized image crops.
transform_two_160x160_image_crops = StackMultipleViews(
    transform=T.Compose([
        T.RandomResizedCrop(160, scale=[0.08, 0.6]),
        T.RandomApply([T.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
        T.RandomGrayscale(p=0.2),
        T.RandomApply([GaussianBlur(sigma=[0.1, 2.0])], p=0.5),
        T.RandomHorizontalFlip(),
        T.ToTensor(),
        normalize]),
    num_views=2)

# Define the transformations for generating two 160x160-sized image crops.
transform_five_96x96_image_patches = T.Compose([
    T.RandomResizedCrop(256, scale=[0.6, 1.0]),
    T.RandomApply([T.ColorJitter(0.4, 0.4, 0.4, 0.1)], p=0.8),
    T.RandomGrayscale(p=0.2),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    normalize,
    CropImagePatches(patch_size=96, patch_jitter=24, num_patches=5)])
```

References

[1] Yuki Markus Asano, Christian Rupprecht, and Andrea Vedaldi. Self-labelling via simultaneous clustering and representation learning. In *ICLR*, 2020. 4

[2] Mathilde Caron, Ishan Misra, Julien Mairal, Priya Goyal, Piotr Bojanowski, and Armand Joulin. Unsupervised learning of visual features by contrasting cluster assignments. In *NeurIPS*, 2020. 4

[3] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. A simple framework for contrastive learning of visual representations. In *ICML*, 2020. 1

[4] Marco Cuturi. Sinkhorn distances: Lightspeed computation of optimal transport. In *NeurIPS*, 2013. 4

[5] Spyros Gidaris, Andrei Bursuc, Nikos Komodakis, Patrick Pérez, and Matthieu Cord. Learning representations by predicting bags of visual words. In *CVPR*, 2020. 4

[6] Priya Goyal, Dhruv Mahajan, Abhinav Gupta, and Ishan Misra. Scaling and benchmarking self-supervised visual representation learning. In *ICCV*, 2019. 3, 4

[7] Kaiming He, Haoqi Fan, Yuxin Wu, Saining Xie, and Ross Girshick. Momentum contrast for unsupervised visual representation learning. In *CVPR*, 2020. 4

[8] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. In *ICCV*, 2017. 4

[9] Ishan Misra and Laurens van der Maaten. Self-supervised learning of pretext-invariant representations. In *CVPR*, 2020. 4

[10] Aaron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. In *NeurIPS*, 2017. 4

[11] Ali Razavi, Aaron van den Oord, and Oriol Vinyals. Generating diverse high-fidelity images with VQ-VAE-2. In *NeurIPS*, 2019. 4

[12] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NeurIPS*, 2015. 4

[13] Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *NeurIPS*, 2017. 2

[14] Oriol Vinyals, Charles Blundell, Tim Lillicrap, and Daan Wierstra. Matching networks for one shot learning. In *NeurIPS*, 2016. 1

[15] Yuxin Wu, Alexander Kirillov, Francisco Massa, Wan-Yen Lo, and Ross Girshick. Detectron2, 2019. 4