torchdistill: A Modular, Configuration-Driven Framework for Knowledge Distillation

Yoshitomo Matsubara$^{(\boxtimes)}$ (iD)

University of California, Irvine, CA 92697, USA
`yoshitom@uci.edu`

Abstract. While knowledge distillation (transfer) has been attracting attentions from the research community, the recent development in the fields has heightened the need for reproducible studies and highly generalized frameworks to lower barriers to such high-quality, reproducible deep learning research. Several researchers voluntarily published frameworks used in their knowledge distillation studies to help other interested researchers reproduce their original work. Such frameworks, however, are usually neither well generalized nor maintained, thus researchers are still required to write a lot of code to refactor/build on the frameworks for introducing new methods, models, datasets and designing experiments. In this paper, we present our developed open-source framework built on PyTorch and dedicated for knowledge distillation studies. The framework is designed to enable users to design experiments by declarative PyYAML configuration files, and helps researchers complete the recently proposed ML Code Completeness Checklist. Using the developed framework, we demonstrate its various efficient training strategies, and implement a variety of knowledge distillation methods. We also reproduce some of their original experimental results on the ImageNet and COCO datasets presented at major machine learning conferences such as ICLR, NeurIPS, CVPR and ECCV, including recent state-of-the-art methods. All the source code, configurations, log files and trained model weights are publicly available at https://github.com/yoshitomo-matsubara/torchdistill.

Keywords: Knowledge distillation · Open source framework · Reproducibility

1 Introduction

Deep learning methods have been achieving state-of-the-art performances, contributing to the rapid development of applications for a variety of tasks such as image classification [11,23,41,43] and object detection [4,10,35]. One of the critical problems with such state-of-the-art models is their complexity, thus the complex models are difficult to be deployed for real-world applications. In general, there is a trade-off between model complexity and inference performance (*e.g.*,

Table 1. Knowledge distillation frameworks. **torchdistill** supports modules in PyTorch and torchvision such as loss, datasets and models. ImageNet: ILSVRC 2012 [37], YT Faces: YouTube Faces DB [47], MIT Scenes: Indoor Scenes dataset [32], CUB-2011: Caltech-UCSD Birds-200-2011 [45], Cars: Cars dataset [18], SOP: Stanford Online Products [27]. P: Pretrained models, M: Module abstraction, D: Distributed training.

Framework	Supported datasets	Models	P	M	D
Zagoruyko & Komodakis [52]	CIFAR-10, ImageNet	Hard-coded	✓		
Passalis & Tefas [29]	CIFAR-10, YT Faces	Hard-coded			
Heo *et al.* [12]	CIFAR-10, MIT scenes	Hard-coded	✓		
Park *et al.* [28]	Cars, CUB-2011, SOP	Hard-coded			
Tian *et al.* [42]	CIFAR-100	Hard-coded	✓		
Yuan *et al.* [51]	CIFAR-10, -100, Tiny ImageNet	Hard-coded	✓		
Xu *et al.* [49]	CIFAR-100	Hard-coded	✓		
torchdistill	torchvision*	torchvision*	✓	✓	✓

* **torchdistill** supports those implemented with PyTorch. In this paper, our focus is on torchvision.

measured as accuracy), and there are three different types of method to make models deployable: 1) designing lightweight models, 2) model compression/pruning, and 3) knowledge distillation. Lightweight models such as MobileNet [14,38], MnasNet [40] and YOLO series [33,34] often sacrifice inference performance to reduce inference time, compared to complex models *e.g.*, ResNet [11] and Mask R-CNN [10]. Model compression and pruning [9,21] techniques reduce model size by quantizing parameters and pruning redundant neurons, and such methods are covered by Distiller [54], an open-source library for model compression.

In this paper, our focus is on the last category, knowledge distillation, that trains a simpler (student) model to mimic the behavior of a powerful (teacher) model. Knowledge distillation [13] stems from the study by Buciluǎ *et al.* [3], that presents a method to compress large, complex ensembles into smaller models with small loss in inference performance. Interestingly, Ba and Caruana [2] report that student models trained to mimic the behavior of the teacher models (*soft-label*) significantly outperform those trained on the original (*hard-label*) dataset. Following these studies, knowledge distillation and transfer have been attracting attention from the research communities such as computer vision [36] and natural language processing [39].

As summarized in Table 1, some researchers voluntarily publish their knowledge distillation frameworks *e.g.*, [12,28,29,42,49,52] to help other researchers reproduce their original studies. However, such frameworks are usually not either well generalized or maintained to be built on. Besides, Distiller [54] supports only one method for knowledge distillation, and Catalyst [17] is a framework built on PyTorch with a focus on reproducibility of deep learning research. To support various deep learning methods, these frameworks are well generalized, yet require users to *hardcode* (reimplement) critical modules such as models and datasets,

even if the implementations are publicly available in popular libraries, to design complex knowledge distillation experiments. As pointed out by Gardner *et al.* [6], reference methods and models are often re-implemented from scratch, and this makes it difficult to reproduce the reported results. For further advancing the deep learning research, a new generalized framework is therefore needed, and the framework should be able to allow researchers to easily try different modules (*e.g.*, models, datasets, loss configurations), implement various approaches, and take care of reproducibility of their work.

The concept of our framework, **torchdistill**,[1] is highly inspired by AllenNLP [6], a platform built on PyTorch [30] for research on deep learning methods in natural language processing. Similar to AllenNLP, **torchdistill** supports the following features:

– module abstractions that enable researchers to write higher-level code for experiments *e.g.*, model, dataset, optimizer and loss;
– declarative PyYAML configuration files, which can be seen as high-level summaries of experiments (training and evaluation), enable to use anchors and aliases in the file to refer to the same object (*e.g.*, file paths) and simplify themselves, and make it easy to change the abstracted components and hyperparameters; and
– generalized reference code and configurations to apply knowledge distillation methods to PyTorch and torchvision models pretrained on well-known complex benchmark datasets: ImageNet (ILSVRC 2012) [37] and COCO 2017 [22].

Furthermore, **torchdistill** supports 1) seamless multi-stage training, 2) caching teacher's outputs, and 3) redesigning (pruning) teacher and student models without hard-coding (reimplementation). To the best of our knowledge, this is the first, highly generalized open-source framework that can support a variety of knowledge distillation methods, and lower barriers to high-quality, reproducible deep learning research [8]. Researchers can explore methods and shape new approaches, building on this generalized framework that makes it easy not only to customize existing methods and models, but also introduce completely new ones. Using some of our reimplemented methods, we also reproduce the experimental results on ILSVRC 2012 and COCO 2017 datasets reported in the original studies.

2   Framework Design

Our developed framework, **torchdistill**, is an open source framework dedicated for knowledge distillation studies, built on PyTorch [30]. For vision tasks such as image classification and object detection, the framework is designed to support torchvision, that offers a lot of options for datasets, model architectures and common image transformations. The collection of supported reference models and datasets in our framework are dependent on the version of user's installed

---

[1] https://github.com/yoshitomo-matsubara/torchdistill.

torchvision. For instance, when users find new models in the latest torchvision, they can shortly try the models simply by updating the torchvision and configuration files for their experiments with our framework.

2.1   Module Abstractions

An objective of module abstractions in our framework is to enable researchers to experiment with various modules by simply changing a PyYAML configuration file described in Sect. 2.3. We focus abstraction on critical modules to experiment, specifically model architectures, datasets, transforms, and losses to be minimized during training. These modules are often hard-coded (See Appendix A) in authors' published frameworks [12,28,29,42,49,52], and many of the hyperparameters are hard-coded as well.

Model Architectures: torchvision offers various model families for vision tasks from AlexNet [20] to R-CNNs [10,35], and many of them are pretrained on large benchmark datasets. Specifically, the latest release (v0.8.2) provides about 30 image classification models pretrained on ImageNet (ILSVRC 2012) [37] and 4 object detection models pretrained on COCO 2017 [22]. As our framework supports torchvision for vision tasks, researchers can use such pretrained models as teacher and/or baseline models (*e.g.*, student trained without teacher). In addition to the pretrained models available in torchvision, they can use their own pretrained model weights and any model architectures implemented with PyTorch. Moreover, **torchdistill** supports PyTorch Hub[2] and enable users to import modules via the hub by specifying repository names in a PyYAML configuration file.

Datasets: As described above, torchvision also supports a variety of datasets, and previous studies [1,12,16,24,28,29,31,36,42,44,46,50,52] use many of them to validate proposed distillation techniques such as ImageNet [37], COCO [22], CIFAR-10 and -100 [19], and Caltech101 [5]. Similar to model architectures, **torchdistill** supports such datasets and can collaborate with any datasets implemented with PyTorch.

Transforms: In vision tasks, there are de facto standard image transform techniques. Taking image classification on the ImageNet dataset as an example, a standard transform pipeline for training with torchvision[3] consists of 1) making a crop of random size of the original size and with a random aspect ratio of the original aspect ratio, 2) horizontal reflection with 50% chance for data augmentation to reduce a risk of overfitting [20], 3) PIL-to-Tensor conversion, and 4) channel-wise normalization using (0.485, 0.456, 0.406) and (0.229, 0.224, 0.225) as means and standard deviations, respectively. In **torchdistill**, users can define their own transform pipeline in a configuration file.

---

[2] https://pytorch.org/hub/.
[3] https://github.com/pytorch/vision/blob/master/references/classification/train.py.

Losses: In distillation process, student models are trained using outputs from teacher models, and the research community has been proposing a lot of unique losses with/without task-specific losses such as cross entropy loss for classification tasks. PyTorch [30] supports various loss classes/functions, and simple distillation losses can be defined in a configuration file by combining such supported losses using **torchdistill**'s customizable loss module (See Sect. 2.6).

2.2 Registry

The registry is an important component in **torchdistill** as abstracted modules are instantiated by mapping strings in the configuration file to the objects in code. Furthermore, it would make it easy for users to collaborate their implemented modules/functions with this framework. Similar to AllenNLP [6] and Catalyst [17], this can be done even outside the framework by using a Python decorator. The following example shows that a new model class, *MyModel*, is added to the framework by simply using *@register_model* (defined in the framework), and the new class can be instantiated by defining "MyModel" with required parameters at designated places in a configuration file.

```
@register_model
class MyModel(nn.Module):
    def __init__(self, *args, **kwargs):
        super().__init__()
        self.conv1 = nn.Conv2d(**kwargs['conv1_kwargs'])
        ...
```

2.3 Configurations

An experiment can be defined by a PyYAML configuration file (See Appendix B), that allows users to tune hyperparameters, and change methods/models without hard-coding. With PyYAML's features, configuration files allow users to leverage anchors and aliases, and these features would be helpful to simplify the configurations in cases that users would like to reuse parameters defined in the configuration file such as root directory path for datasets, parameters and model names as part of checkpoint file paths for better data management. In a configuration file, there are three main components to be defined: datasets, teacher and student models, and training. Each of the key components is defined by using abstracted and registered modules described in Sects. 2.1 and 2.2. A configuration file gives users a summary of the experiment, and shows all the parameters to reproduce the experimental results except implicit factors such as hardware specifications used for the experiment.

The following example illustrates how to define a global teacher model declared in a PyYAML configuration file. As described in the previous sections, various types of modules are abstracted in our framework, and such modules (classes and functions) in user's installed torchvision are registered. In this example, 'resnet34' function[4] is used to instantiate an object of type *ResNet* by using

---

[4] https://pytorch.org/docs/stable/torchvision/models.html#torchvision.models.resnet34.

a dictionary of keyword arguments (***params*). *i.e. num_classess* = 1000 and *pretrained* = True are given as arguments of 'resnet34' function. For image classification models implemented in torchvision or those users add to the registry in our framework, users can easily try different models by changing 'resnet34' *e.g.*, 'densenet201' [15], 'mnasnet1_0' [40]. Besides that, *ckpt* indicates the file path of checkpoint, that is './resnet34.pt' in the example defined by leveraging some of YAML features: anchors (&) and aliases (*). For teacher model, the checkpoint will be used to initialize the model with user's own model weights if the checkpoint file exists. Otherwise, 'resnet34' in this example will be initialized with torchvision's pretrained weights for ILSVRC 2012.

```
teacher_model:
    name: &teacher 'resnet34'
    params:
        num_classes: 1000
        pretrained: True
    ckpt: !join ['./', *teacher, '.pt']
```

Furthermore, **torchdistill** offers an option to generate log files that monitor the experiments. For instance, a log file presents what parameters were used, when executed, the trends of training behavior (*e.g.*, training loss, learning rate and validation accuracy) at a frequency set in the configuration file, and evaluation results.
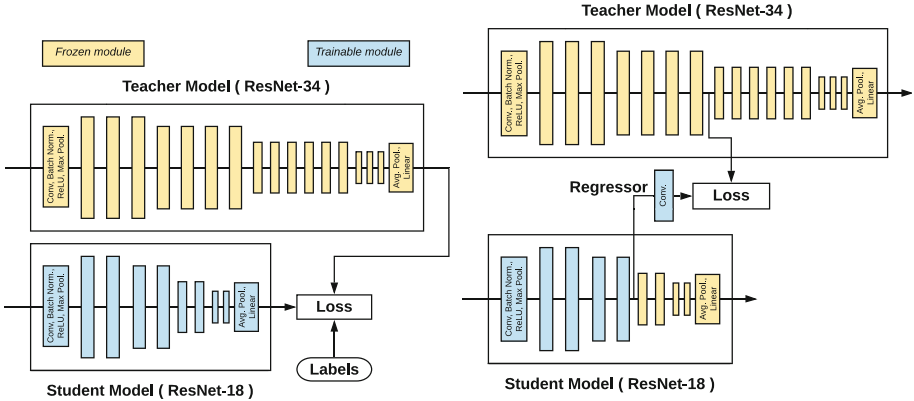
These configuration and log files[5] will also help the researchers complete *ML Code Completeness Checklist*,[6] that was recently proposed to facilitate reproducibility in the research community as part of the official code submission process at major machine learning conferences *e.g.*, NeurIPS, ICML and CVPR.

2.4   Dataset Wrappers

To support a wide variety of knowledge distillation methods, dataset is an important module to be generalized. Usually, the dataset module in PyTorch and torchvision returns a pair of input batch (*e.g.*, collated image tensors) and targets (ground-truth) at each iteration, but some of the existing knowledge distillation approaches require additional information for the batch. For instance, contrastive representation distillation (CRD) [42] requires an efficient strategy to retrieve a large number of negative samples in the training session, that requires the dataset module to return an additional object (*e.g.*, negative sample indices). To support such extensions, we design dataset wrappers to return input batch, targets, and a supplementary dictionary, that can be empty when not used. For the above case, the additional object can be stored in the supplementary dictionary, and used when computing the contrastive loss. This design also enables us to support caching teacher model's outputs against data indices in the original dataset so that teacher's inference can be skipped by caching (serializing) outputs of the teacher model given a data index at the first epoch, and reading and collating the cached outputs given batch of data indices at the following epochs.

---

[5] Available at https://github.com/yoshitomo-matsubara/torchdistill/tree/master/configs/.

[6] https://github.com/paperswithcode/releasing-research-code.

(a) Knowledge distillation [13] using ResNet-34 and ResNet-18 as teacher and student models, respectively.

(b) Hint-training with an auxiliary module (convolutional regressor) as stage 1 of FitNet method [36]. Its stage 2 is knowledge distillation as illustrated in Figure 1a.

Fig. 1. Knowledge distillation and FitNet methods. Yellow and blue modules indicate that their parameters are frozen and trainable, respectively (Color figure online).

Table 2. Epoch-level training speed improvement by **caching teacher's outputs** at the 1st epoch, using ResNet-18 as student model for knowledge distillation [13].

Teacher	ResNet-34	ResNet-50	ResNet-101	ResNet-152
No cache	801 sec	1,030 sec	1,348 sec	1,944 sec
Cache (1st)	859 sec	1,079 sec	1,402 sec	1,966 sec
Cache (2nd)	**651 sec**	**649 sec**	**656 sec**	**917 sec**

To demonstrate that caching improves training efficiency, we perform an experiment with knowledge distillation [13] illustrated in Fig. 1a that caches outputs of the teacher model at the first epoch for training ResNet-18 (student) on ILSVRC 2012 dataset, and skips the teacher model's inference by loading and feeding the outputs cached on disk to the loss module. Table 2 suggests that spending an extra one-minute at the 1st epoch to serialize teacher's outputs, the caching strategy makes the following training process (*i.e.* from the 2nd epoch) approximately 1.23 – 2.11 times faster at epoch-level when using 3 NVIDIA GeForce RTX 2080 Ti's with batch size of 256. Also, this improvement becomes more significant when using a larger teacher model such as ResNet-152 (approximately 2.11 times faster than training without cache). The ILSVRC 2012 training dataset consists of approximately 1.3 million images, and the cached files consumes only 10GB whereas the original training dataset uses about 140GB. Note that caching may not improve the training efficiency if teacher's outputs to be cached are much larger *e.g.*, hint-based training [36] requires intermediate
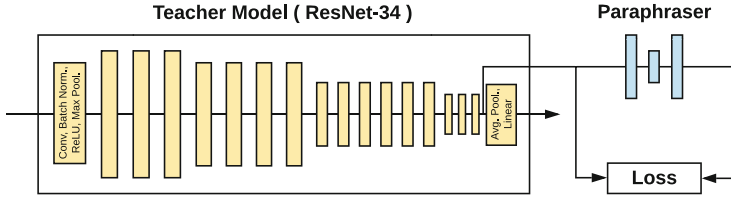
(a) 1st stage: training *paraphraser* for teacher model.
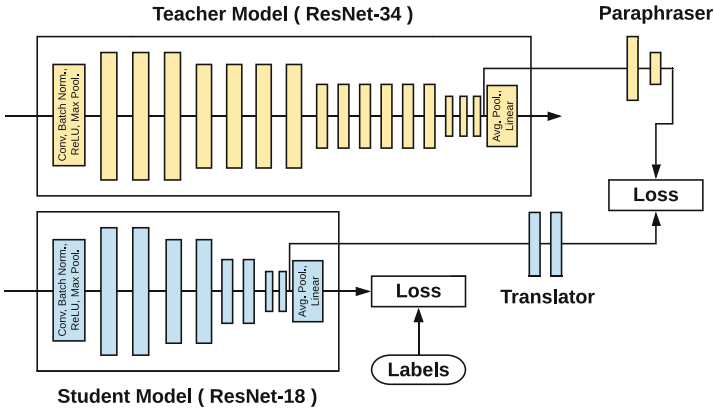
(b) 2nd stage: training student model and *translator*, using labels and outputs of *paraphraser*'s middle layer.

Fig. 2. Factor transfer with two auxiliary modules.

outputs from teacher and student models. Also, this mode should be turned off when applying data augmentation strategies.

2.5   Teacher and Student Models

Teacher-Student pairs are keys in knowledge distillation experiments, and recently proposed approaches [1,12,31,36,42,49,50,52,53] introduce auxiliary modules, which are used only in training session. Such auxiliary modules use tensors from intermediate layers in models, and introducing the modules to the models often results in branching their feedforward path as shown in Figs. 1 and 2. This paradigm, however, is also one of the backgrounds that researchers decide to hard-code the models (*e.g.*, modify the original implementations of models in torchvision every time they change the placement of auxiliary modules for preliminary experiments) to introduce such auxiliary modules used for their proposed methods, and make it difficult for other researchers to build on the published frameworks [12,28,29,42,49,52].

Taking an advantage of forward hook paradigm in PyTorch [30], **torchdistill** supports introducing such auxiliary modules without altering the original

implementations of the models. Specifically, users can register the framework's provided *forward hooks* to specific modules to store its input and/or output in a I/O dictionary by specifying the module paths (*e.g.*, "conv1" for a MyModel object in Sect. 2.2) in the configuration files. The I/O dictionaries for teacher and student models will be fed to a generalized, customizable loss module described in Sect. 2.6.

For methods that not only require to extract the intermediate outputs (See Fig. 1) but also feed the extracted outputs to trainable auxiliary modules in different branches to be processed (See Fig. 2b), we define a *special* module in the framework, that is designed to have a *post-forward* function. In Fig. 1, for instance, the framework first executes ResNet-18 and extracts intermediate output by a registered forward hook, and then the extracted output stored in the student's I/O dictionary will be fed to the regressor as part of the *post-forward* process. The concept of the special module gives users more flexibility in designing training methods while leaving the original implementations of models (ResNet-34 and ResNet-18 in Fig. 2) unaltered.

2.6   Customizable Loss Module

Leveraging the I/O dictionaries that contain input/output of specific modules with registered forward hooks, **torchdistill** provides a generalized customizable loss module that allows users to easily combine different loss modules with balancing factors by configuration files such as those in Fig. 2b. Given a pair of input x and ground-truth y, the I/O dictionaries consist of a set of keys J and the values z_j^{S} and z_j^{T} ($j \in J$) extracted from student and teacher models respectively. Using the I/O dictionaries and the ground-truth, the generalized loss is defined as

$$\mathcal{L} = \sum_{j \in J} \lambda_j \cdot \mathcal{L}_j(z_j^{\mathrm{S}}, z_j^{\mathrm{T}}, y), \tag{1}$$

where λ_j is a balancing weight (hyperparameter) for $\mathcal{L}_j$, which is either a loss module implemented in PyTorch [30] or user's defined loss module in registry.

For instance, the loss function to train student model on ILSVRC 2015 dataset [37] at the 2nd stage of factor transfer (Fig. 2b) can be defined as:

$$\mathcal{L} = \lambda_{\mathrm{cls}} \cdot \mathcal{L}_{\mathrm{cls}}(z_{\mathrm{cls}}^{\mathrm{S}}, z_{\mathrm{cls}}^{\mathrm{T}}, y) + \lambda_{\mathrm{FT}} \cdot \mathcal{L}_{\mathrm{FT}}(z_{\mathrm{FT}}^{\mathrm{S}}, z_{\mathrm{FT}}^{\mathrm{T}}, y) \tag{2}$$

$$\mathcal{L}_{\mathrm{cls}}(z_{\mathrm{cls}}^{\mathrm{S}}, z_{\mathrm{cls}}^{\mathrm{T}}, y) = \mathrm{CrossEntropyLoss}(z_{\mathrm{cls}}^{\mathrm{S}}, y)$$

$$\mathcal{L}_{\mathrm{FT}}(z_{\mathrm{FT}}^{\mathrm{S}}, z_{\mathrm{FT}}^{\mathrm{T}}, y) = \left\| \frac{z_{\mathrm{FT}}^{\mathrm{S}}}{\left\| z_{\mathrm{FT}}^{\mathrm{S}} \right\|_2} - \frac{z_{\mathrm{FT}}^{\mathrm{T}}}{\left\| z_{\mathrm{FT}}^{\mathrm{T}} \right\|_2} \right\|_p,$$

where $\lambda_{\mathrm{cls}} = 1$, $\lambda_{\mathrm{FT}} = 1,000$ and $p = 1$, following [16].

2.7   Stage-wise Training Configuration

In the previous sections, we describe the main features of **torchdistill**, and what modules are configurable in the framework. We emphasize that all the training configurations described above can be defined stage-wisely.

Seamless Multi-stage Training Configurations: Specifically, the framework is designed to enable users to configure critical components such as 1) number of epochs, 2) training and validation datasets, 3) teacher and student models, 4) modules (layers) to be trained/frozen, 5) optimizer, 6) learning rate scheduler, 7) loss module. These components can be re-defined at each of training stages, otherwise the framework reuses those from the previous stage. Notice that these training configurations can be declared in a configuration file, and this design enables to support not only two-stage training strategies [12,16,36,50], but also more complicated distillation methods such as teacher assistant knowledge distillation (TAKD) [26], that trains TAs to fill the gap between student and teacher models. Transfer learning also can be supported by changing models and datasets from stage to stage, and users would execute code with a configuration file only once. Therefore, they will not need to execute code multiple times to perform multi-stage training, including transfer learning.

Redesigning Models for Efficient Training: Furthermore, our framework gives users an option to redesign teacher and student models at each stage by specifying the required modules in a configuration file. Specifically, users are allowed to rebuild models by reusing modules in the models optionally with auxiliary modules. Figure 1 shows an example that modules after the 8th and the 5th blocks of the teacher and student models respectively can be pruned as the outputs of the modules are not used in the hint-training (1st stage), thus not required to be executed. In this specific case, the redesigned student model will consist of the trainable (blue) modules and a regressor (auxiliary module) as illustrated in Fig. 3, and the teacher and student architectures at the 2nd stage will be reverted to the original ones (Fig. 1a) with parameters learnt at the 1st stage. Also, the redesigned teacher/student model can be an empty module to save execution time. In Fig. 2a, for instance, there is no need to feed input batch to the student model (thus, can be empty) as at the 1st stage of factor transfer, only the teacher model is executed to train the paraphraser.

As introduced in Sect. 2.4, when the teacher's outputs are cacheable (*e.g.*, in terms of available disk space), teacher's inference can be skipped by loading the cache files produced at previous epoch. Redesigning models help users shorten training sessions even when teacher's outputs are not cacheable. Note that student model's outputs, however, cannot be cached as the model's parameters are updated every iteration. Table 3 suggests that redesigning models using only modules to be executed for training would be an effective approach to saving training time, and this improvement would be more critical for training models on large datasets and/or with a lot of epochs. We emphasize that users can redesign (minimize) the models by specifying the required modules in a configuration file rather than hardcode (reimplement) the pruned models.
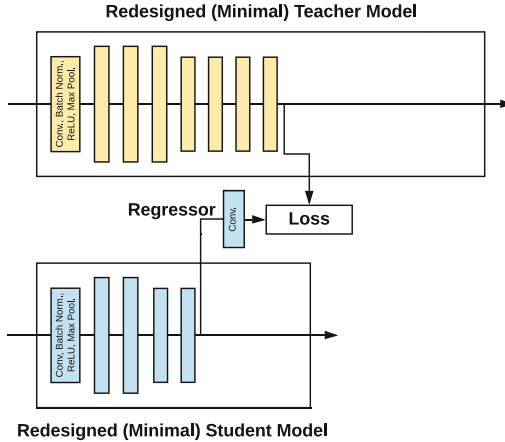
Fig. 3. Hint-training with teacher and student models pruned simply by specifying required modules in a configuration file for further efficient training, compared to a naive configuration in Fig. 1.

3   Reference Methods

Here, we describe the reimplementations of knowledge distillation methods and experiments to reproduce the reported results on ImageNet and COCO datasets.

Table 3. Epoch-level training speed improvement by **redesigning teacher and student (ResNet-18) models** with required modules only for hint-training shown in Fig. 3.

Teacher	ResNet-34	ResNet-50	ResNet-101	ResNet-152
Original	934 sec	1,175 sec	1,468 sec	1,779 sec
Minimal	**786 sec**	**929 sec**	**936 sec**	**1,022 sec**

3.1   Reimplementations

Given that the pretrained models in torchvision are trained on large benchmark datasets, ImageNet (ILSVRC 2012) [37], and COCO 2017 [22], we focus our implementations on these datasets as the pretrained models can be used as teacher models and/or baseline student models (naively trained on human-annotated datasets). Note that some of the methods are not validated on these datasets in their original work.

Table 4 shows a brief summary of reference distillation methods reimplemented with **torchdistill**, and indicates what additional modules were implemented and added to the registry for reimplementing the methods. We emphasize that methods without any check marks ($\checkmark$) in the *Required additional modules* columns such as KD, AT, PKT, RKD, HND, SPKD, Tf-KD, GHND and L_2 can be reimplemented simply by adding the new loss modules to the registry in the framework (Sect. 2.2).

Different from the existing frameworks [12, 28, 29, 42, 49, 52], all the methods in Table 4 are reimplemented independently from models in torchvision so that users can easily switch models by specifying a model name and its parameters in a configuration file. Taking image classification as an example, the shapes of inputs and (intermediate) outputs for the models are often fixed (*e.g.*, $3 \times 224 \times 224$ and 1,000 respectively, for models trained on ImageNet dataset), that makes it easy to match the shape of student's output with that of teacher when computing loss values to be minimized.

3.2 Reproducing ImageNet Experiments

In this section, we attempt to reproduce some experimental results with their proposed distillation methods. In particular, we choose the attention transfer (AT), factor transfer (FT) [16], contrastive representation distillation (CRD) [42], teacher-free knowledge distillation (Tf-KD) [51], self-supervised knowledge distillation (SSKD) [49], L_2 and prime-aware adaptive distillation (PAD-L_2) methods [53] for the following reasons:

- these methods are validated with the ImageNet datasets for ResNet-34 and ResNet-18 as teacher and student models in their original work;[7]
- the hyperparameters used in the ImageNet experiments are described in the original studies and/or their published source code; and
- we did not have time to tune hyperparameters for other methods that are not validated on the ImageNet dataset in their original papers.

In addition to the methods, we apply knowledge distillation (KD) [13] to the same teacher-student pair. Note that except KD[8], we reuse the hyperparameters (*e.g.*, number of epochs) for ImageNet given in their original work to reproduce their experimental results, and we provide the configuration and log files, and trained model weights (See footnote 5).

We also should note that Zagoruyko and Komodakis [52] propose attention transfer (AT), and define the following total loss function for their ImageNet experiment:

$$\mathcal{L}_{AT} = \mathcal{L}(\mathrm{W}_S, x) + \frac{\beta}{2} \sum_{j \in \mathcal{I}} \left\| \frac{Q_S^j}{\left\| Q_S^j \right\|_2} - \frac{Q_T^j}{\left\| Q_T^j \right\|_2} \right\|_p, \tag{3}$$

---

[7] The teacher model for Tf-KD is the pretrained ResNet-18 [51].

[8] For KD, we set hyperparameters as follows: temperature $T = 1$ and relative weight $\alpha = 0.5$.

Table 4. Reference knowledge distillation methods implemented in **torchdistill**.

Methods	Multi-stage training	Required additional modules		
		Auxiliary	Special	Custom dataset
KD [13]				
FitNet [36]	✓	✓		
FSP [50]	✓	✓		
AT [52]				
PKT [29]				
FT [16]	✓	✓	✓	
DAB [12]	✓	✓	✓	
RKD [28]				
VID [1]		✓	✓	
CCKD [31]		✓	✓	
HND [24]				
SPKD [44]				
CRD [42]		✓	✓	✓
Tf-KD [51]				
GHND [25]				
SSKD [49]	✓	✓	✓	✓
L_2 [53]				
PAD-L_2 [53]	✓	✓	✓	

where $\mathcal{L}(W_S, x)$ is a standard cross entropy loss, and Q_S^j and Q_T^j denote the vectorized forms of the j-th pair of student and teacher attention maps, respectively (Refer to their work [52] for more details). In their published framework[9], they set β and p to 1,000 and 2 respectively. However, we find a discrepancy between their defined loss function (Eq. (3)) and their implemented loss function (Eq. (4)), that computes mean squared error (MSE) between the teacher and student attention maps.

$$\mathcal{L}_{AT} = \mathcal{L}(W_S, x) + \frac{\beta}{2} \sum_{j \in \mathcal{I}} MSE\Big(\frac{Q_S^j}{\left\| Q_S^j \right\|_2} , \frac{Q_T^j}{\left\| Q_T^j \right\|_2} \Big) \tag{4}$$

In our preliminary experiment with hyperparameters the authors provide, the student model did not train well with the loss module based on Eq. (3). For this reason, we used Eq. (4) instead for AT in our experiments.

Table 5 summarizes the results of the experiments with the training configurations (*e.g.*, teacher-student pair, hyperparameters) described in each of the original studies and/or verified by the authors. In addition to experiments with a single GPU, we perform experiments with a distributed training strategy supported by PyTorch (reported with a dagger mark †) to demonstrate that our framework supports the strategy for saving training time. As for the L_2 and

---

[9] https://github.com/szagoruyko/attention-transfer.

Table 5. Validation accuracy of ResNet-18 (student) trained on ILSVRC 2012 dataset with ResNet-34 (teacher), using eight different distillation methods. With the hyperparameters (*e.g.*, # Epochs) either described in the original work or given by the authors, all the reimplemented methods outperform the student model trained without teacher.

	Accuracy[%]		# Epochs	Training time
	Top-1	Diff.		
Teacher: ResNet-34	73.31	+3.56	N/A	N/A
Student: ResNet-18	69.75	0.00	N/A	N/A
KD	71.23	+1.48	100	60 hr 04 min
KD †	71.37	+1.62	100	23 hr 07 min
AT	70.90	+1.15	100	59 hr 07 min
AT †	70.55	+0.80	100	23 hr 11 min
FT	71.56	+1.81	91	55 hr 06 min
FT †	71.13	+1.38	91	22 hr 15 min
CRD	70.81	+1.06	100	356 hr 31 min
CRD ‡	70.93	+1.18	100	179 hr 12 min
Tf-KD	70.52	+0.77	90	46 hr 34 min
Tf-KD †	70.21	+0.46	90	18 hr 50 min
SSKD ‡	70.09	+0.34	130	113 hr 12 min
L_2 ‡	71.08	+1.33	90	21 hr 25 min
PAD-L_2 ‡	71.71	+1.96	(90 +) 30	28 hr 34 min

† Distributed training on 3 GPUs with linear scaling rule [7]: Learning rates are modified according to the number of distributed training processes. (*i.e.* multiplied by the number of GPUs).
‡ Distributed training on 3 GPUs with total batch size used in original work.

PAD-L_2 methods, the original study [53] uses batch size of 512 for their ImageNet experiments, which did not fit in our single GPU. Thus, we split the batch size into 171 per GPU, and report only the results with the distributed training (marked with ‡). The same strategy is applied to SSKD (total batch size of 256 and 768 for normal and augmented samples, respectively [49]) as it takes at least 4 times as long at epoch-level to train a model, compared to the other methods due to their 4x augmented training data, and our batch size per GPU is 85 (for normal samples + 255 for augmented samples). Similarly, we apply the same strategy for CRD due to the limited time. We also note that Zhang *et al.* [53] applied their proposed PAD-L_2 to the student model trained with their proposed L_2 as a pretrained model, and train the student model with the PAD-L_2 method for 30 more epochs (*i.e.*, 120 epochs).[10]

Based on the methods we reimplemented with **torchdistill**, we successfully reproduce the results on the ILSVRC 2012 dataset for the teacher-student pair

---

[10] The configuration is not described in [53], but verified by the authors.

reported in the original papers of AT [52], Tf-KD [51], L_2 and PAD-L_2 [53] methods, and the result of PAD-L_2 was recently reported as the state-of-the-art performance for the teacher-student pair on the ILSVRC 2012 dataset [53]. All the results outperform the baseline performance (S: ResNet-18) which is trained with human-labels only, and the pretrained model is provided by torchvision. Note that FT was validated on ILSVRC 2015 dataset in their original work [16], and we confirm the FT's improvement over a baseline using ILSVRC 2012 dataset as the teacher model (ResNet-34) in torchvision is pretrained on the dataset. The result with the reimplemented CRD is almost comparable to the accuracy reported in the original study [42]. In CRD, both positive and negative samples are leveraged for learning representations, thus turns out to be the most-time consuming method in Table 5. The reimplemented SSKD outperforms the baseline model although the accuracy does not match the reported result [49]. A potential factor may be a different training configuration forced by our limited computing resource (*e.g.*, different batch size per GPU whereas 8 parallel GPUs were used in their work) since we simply refactored and made the authors' published code compatible with the ILSVRC 2012 dataset. As pointed out by Tian *et al.* [42], KD [13] is still a powerful method. Our reimplemented KD outperformed their proposed state-of-the-art method, CRD (71.17%), and achieved the comparable accuracy with their CRD+KD (71.38%) method.

3.3   Reproducing COCO Experiments

To demonstrate that our framework can 1) be applied to different tasks, and 2) collaborate with model architectures that are not implemented in torchvision, we apply the generalized head network distillation (GHND) to bottleneck-injected R-CNN object detectors for split computing [25], using COCO 2017 dataset. Their proposed bottleneck-injected Faster and Mask R-CNNs with ResNet-50 and FPN are designed to be partitioned into head and tail models which will be deployed on mobile device and edge server respectively, for reducing inference speed in resource-constrained edge computing systems. Following the original work on GHND, we apply the method to a pair of the original and bottleneck-injected Faster R-CNNs as teacher and student respectively, and conduct the same experiment for Mask R-CNN as well. As shown in Table 6, the reproduced mean average precision (mAP) match those reported in the original study [25].

Table 6. Validation mAP of bottleneck-injected R-CNN models for split computing (student) trained on COCO 2017 dataset by GHND with original Faster/Mask R-CNN models (teacher). Reproduced results match those reported in the original work [25].

Backbone: ResNet-50 and FPN	mAP		# Epochs	Training time
	BBox	Mask		
Faster R-CNN w/ Bottleneck	0.359	N/A	20	24 hr 13 min
Mask R-CNN w/ Bottleneck	0.369	0.336	20	24 hr 21 min

4   Conclusions

In this work, we presented **torchdistill**, an open-source framework dedicated for knowledge distillation studies, that supports efficient training and configurations systems designed to give users a summary of the experiments. Researchers can build on the framework (*e.g.*, by forking the repository) to conduct their knowledge distillation studies, and their studies can be integrated to the framework by sending a pull request. This will help the research community ensure the reproducibility of the work, and advance the deep learning research while supporting fair method comparison on benchmarks. Specifically, researchers can publish the log, configuration, and pretrained model weights for their champion performance, that will help them ensure the champion performance for specific datasets and teacher-student pairs.

Furthermore, the configuration files for and log files produced by **torchdistill** will help researchers complete the *ML Code Completeness Checklist* (See footnote 6), and we provide the full configurations (hyperparameters), log files and checkpoints including model weights for experimental results shown in Tables 5 and 6 in our code repository (See footnote 1). We provide reference code and configurations for image classification and object detection tasks, and plan to extend our framework for different tasks using popular packages *e.g.*, Transformers [48] for NLP tasks. Our framework will be maintained and updated along with the new releases of PyTorch and torchvision so that users can save time for coding and use it as a standard framework for reproducible knowledge distillation studies.

A   Hard-Coded Module and Forward Hook Configurations

For lowering barriers to high-quality knowledge distillation studies, it would be important to enable users to collaborate with models implemented in popular libraries such as torchvision. However, all the models in the existing frameworks described in this study are reimplemented to extract intermediate representations in addition to the models' final outputs. Figure 4 shows an example of original and hard-coded (reimplemented) forward functions in ResNet model for knowledge distillation experiments. As illustrated in the hard-coded example, the authors [42,49] unpacked an existing implementation of ResNet model and re-designed interfaces of some modules to extract additional representations (*i.e.*, "f0", "f1_pre", "f2", "f2_pre", "f3", "f3_pre", and "f4").

Furthermore, the modified interfaces also require those in the downstream processes to be modified accordingly, that will need extra coding cost. We emphasize that users are required to repeat this procedure every time they introduce

```
def _forward_impl(self, x):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)

    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)

    x = self.avgpool(x)
    x = torch.flatten(x, 1)
    x = self.fc(x)

    return x

def forward(self, x):
    return self._forward_impl(x)
```

```
def forward(self, x, is_feat=False, preact=False):
    x = self.conv1(x)
    x = self.bn1(x)
    x = self.relu(x)
    f0 = x

    x, f1_pre = self.layer1(x)
    f1 = x
    x, f2_pre = self.layer2(x)
    f2 = x
    x, f3_pre = self.layer3(x)
    f3 = x

    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    f4 = x
    x = self.fc(x)

    if is_feat:
        if preact:
            return [f0, f1_pre, f2_pre, f3_pre, f4], x
        else:
            return [f0, f1, f2, f3, f4], x
    else:
        return x
```

Fig. 4. Forward functions in **original** (left, torchvision-style) and **hard-coded** (right, [42,49]) implementations of ResNet. Only "x" from "self.fc" is used for vanilla training and prediction.

new models for experiments, and the same issues will be found when introducing new schemes implemented as other types of module (*e.g.*, dataset and sampler) required by specific methods such as CRD [42] and SSKD [49]. Using a forward hook manager in our framework, we can extract intermediate representations from the original models (*e.g.*, Fig. 4 (left)) without reimplementation like Fig. 4 (right), and help users introduce such schemes with wrappers of the module types so that they can apply the schemes simply by specifying in a configuration file used to design an experiment.

The following example illustrates how to specify the input to or output from modules we would like to extract from ResNet model whose forward function is shown in Fig. 4 (left). "f0", "f1_pre", "f2_pre", and "f3_pre" in Fig. 4 (right) correspond to the output from the first ReLU module "relu", and preactivation representations in "layer1", "layer2", and "layer3" modules, which are the inputs to their last ReLU modules (*i.e.*, "layer1.1.relu", "layer2.1.relu", and "layer3.1.relu"). "f4" is the flatten output from average pooling module "avgpool". Similarly, we can define a forward hook manager for teacher model, and reuse the module paths such as "layer1.1.relu" to define loss functions in the configuration file.

```
student:
  ...
  forward_hook:
    input: ['layer1.1.relu', 'layer2.1.relu', 'layer3.1.relu', 'fc']
    output: ['relu']
```

B   Example PyYAML Configuration

Figure 5 shows an example PyYAML configuration file (See footnote 5) to instantiate abstracted modules for an experiment with knowledge distillation by Hinton *et al.* [13].

```yaml
datasets:
  ilsvrc2012:
    name: &dataset_name 'ilsvrc2012'
    type: 'ImageFolder'
    root: &root_dir !join ['~/dataset/', *dataset_name]
    splits:
      train:
        dataset_id: &imagenet_train !join [*dataset_name, '/train']
        params:
          root: !join [*root_dir, '/train']
          transform_params:
            - type: 'RandomResizedCrop'
              params:
                size: &input_size [224, 224]
            - type: 'RandomHorizontalFlip'
              params:
                p: 0.5
            - &totensor
              type: 'ToTensor'
              params:
            - &normalize
              type: 'Normalize'
              params:
                mean: [0.485, 0.456, 0.406]
                std: [0.229, 0.224, 0.225]
      val:
        dataset_id: &imagenet_val !join [*dataset_name, '/val']
        params:
          root: !join [*root_dir, '/val']
          transform_params:
            - type: 'Resize'
              params:
                size: 256
            - type: 'CenterCrop'
              params:
                size: *input_size
            - *totensor
            - *normalize

models:
  teacher_model:
    name: 'resnet34'
    params:
      num_classes: 1000
      pretrained: True
    ckpt: '/path/to/your_own_checkpoint_if_you_have'
  student_model:
    name: 'resnet18'
    params:
      num_classes: 1000
      pretrained: False
    ckpt: './imagenet/kd/ilsvrc2012-resnet18_from_resnet34.pt'
```

```yaml
train:
  log_freq: 1000
  num_epochs: 100
  train_data_loader:
    dataset_id: *imagenet_train
    random_sample: True
    batch_size: 256
    num_workers: 16
    cache_output:
  val_data_loader:
    dataset_id: *imagenet_val
    random_sample: False
    batch_size: 128
    num_workers: 16
  teacher:
    sequential: []
    wrapper: 'DistributedDataParallel'
    requires_grad: False
  student:
    sequential: []
    wrapper: 'DistributedDataParallel'
    requires_grad: True
    frozen_modules: []
  apex:
    requires: False
    opt_level: '01'
  optimizer:
    type: 'SGD'
    params:
      lr: 0.1
      momentum: 0.9
      weight_decay: 0.0001
  scheduler:
    type: 'MultiStepLR'
    params:
      milestones: [30, 60, 90]
      gamma: 0.1
  criterion:
    type: 'GeneralizedCustomLoss'
    org_term:
      criterion:
        type: 'KDLoss'
        params:
          temperature: 1.0
          alpha: 0.5
          reduction: 'batchmean'
      factor: 1.0
    sub_terms:

test:
  test_data_loader:
    dataset_id: *imagenet_val
    random_sample: False
    batch_size: 1
    num_workers: 16
```

Fig. 5. First (left) and second (right) halves of an example PyYAML configuration to design a knowledge distillation experiment with hyperparameters using **torchdistill**.

References

1. Ahn, S., Hu, S.X., Damianou, A., Lawrence, N.D., Dai, Z.: Variational information distillation for knowledge transfer. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 9163–9171 (2019)

2. Ba, J., Caruana, R.: Do deep nets really need to be deep? In: Advances in Neural Information Processing Systems, pp. 2654–2662 (2014)

3. Buciluă, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 535–541 (2006)

4. Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., Zagoruyko, S.: End-to-end object detection with transformers. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J.M. (eds.) ECCV 2020, vol. 12346. LNCS. Springer, Heidelberg (2020). https://doi.org/10.1007/978-3-030-58452-8_13

5. Fei-Fei, L., Fergus, R., Perona, P.: One-shot learning of object categories. IEEE Trans. Pattern Anal. Mach. Intell. **28**(4), 594–611 (2006)

6. Gardner, M., et al.: AllenNLP: a deep semantic natural language processing platform. ACL **2018**, 1 (2018)

7. Goyal, P., et al.: Accurate, large minibatch SGD: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017)

8. Gundersen, O.E., Kjensmo, S.: State of the art: reproducibility in artificial intelligence. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 32 (2018)

9. Han, S., Mao, H., Dally, W.J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. In: Fourth International Conference on Learning Representations (2016)

10. He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask R-CNN. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 2961–2969 (2017)

11. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)

12. Heo, B., Lee, M., Yun, S., Choi, J.Y.: Knowledge transfer via distillation of activation boundaries formed by hidden neurons. In: Proceedings of the AAAI Conference on Artificial Intelligence, vol. 33, pp. 3779–3787 (2019)

13. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network. In: Deep Learning and Representation Learning Workshop: NIPS 2014 (2014)

14. Howard, A., et al.: Searching for MobileNetV3. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1314–1324 (2019)

15. Huang, G., Liu, Z., Van Der Maaten, L., Weinberger, K.Q.: Densely connected convolutional networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4700–4708 (2017)

16. Kim, J., Park, S., Kwak, N.: Paraphrasing complex network: network compression via factor transfer. In: Advances in Neural Information Processing Systems, pp. 2760–2769 (2018)

17. Kolesnikov, S.: Accelerated DL R&D (2018). https://github.com/catalyst-team/catalyst. Accessed 28 Sept 2020

18. 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, pp. 554–561 (2013)

19. Krizhevsky, A.: Learning multiple layers of features from tiny images (2009)

20. Krizhevsky, A., Sutskever, I., Hinton, G.E.: Imagenet classification with deep convolutional neural networks. In: Advances in Neural Information Processing Systems, pp. 1097–1105 (2012)
21. Li, H., Kadav, A., Durdanovic, I., Samet, H., Graf, H.P.: Pruning filters for efficient convnets. In: Fourth International Conference on Learning Representations (2016)
22. Lin, T.Y., et al.: Microsoft COCO: common objects in context. In: Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T. (eds.) ECCV 2014. LNCS, vol. 8693, pp. 740–755. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10602-1_48
23. Mahajan, D., et al.: Exploring the limits of weakly supervised pretraining. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) ECCV 2018. Exploring the limits of weakly supervised pretraining, vol. 11206, pp. 185–201. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01216-8_12
24. Matsubara, Y., Baidya, S., Callegaro, D., Levorato, M., Singh, S.: Distilled split deep neural networks for edge-assisted real-time systems. In: Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges, pp. 21–26 (2019)
25. Matsubara, Y., Levorato, M.: Neural Compression and Filtering for Edge-assisted Real-time Object Detection in Challenged Networks. arXiv preprint arXiv:2007.15818 (2020)
26. Mirzadeh, S.I., Farajtabar, M., Li, A., Ghasemzadeh, H.: Improved knowledge distillation via teacher assistant. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 5191–5198 (2020)
27. Oh Song, H., Xiang, Y., Jegelka, S., Savarese, S.: Deep metric learning via lifted structured feature embedding. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4004–4012 (2016)
28. Park, W., Kim, D., Lu, Y., Cho, M.: Relational knowledge distillation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3967–3976 (2019)
29. Passalis, N., Tefas, A.: Learning deep representations with probabilistic knowledge transfer. In: Ferrari, V., Hebert, M., Sminchisescu, C., Weiss, Y. (eds.) ECCV 2018. LNCS, vol. 11215, pp. 283–299. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-01252-6_17
30. Paszke, A., et al.: PyTorch: an imperative style, high-performance deep learning library. In: Advances in Neural Information Processing Systems, pp. 8024–8035 (2019)
31. Peng, B., et al.: Correlation congruence for knowledge distillation. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 5007–5016 (2019)
32. Quattoni, A., Torralba, A.: Recognizing indoor scenes. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 413–420. IEEE (2009)
33. Redmon, J., Farhadi, A.: YOLO9000: better, faster, stronger. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 7263–7271 (2017)
34. Redmon, J., Farhadi, A.: YOLOv3: An incremental improvement. arXiv preprint arXiv:1804.02767 (2018)
35. Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: towards real-time object detection with region proposal networks. In: Advances in Neural Information Processing Systems, pp. 91–99 (2015)
36. Romero, A., Ballas, N., Kahou, S.E., Chassang, A., Gatta, C., Bengio, Y.: FitNets: hints for thin deep nets. In: Third International Conference on Learning Representations (2015)

37. Russakovsky, O., et al.: ImageNet large scale visual recognition challenge. Int. J. Comput. Vis. **115**(3), 211–252 (2015)
38. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., Chen, L.C.: MobileNetV2: inverted residuals and linear bottlenecks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4510–4520 (2018)
39. Sanh, V., Debut, L., Chaumond, J., Wolf, T.: Distilbert, a distilled version of BERT: smaller, faster, cheaper and lighter. In: The 5th Workshop on Energy Efficient Machine Learning and Cognitive Computing (2019)
40. Tan, M., et al.: Mnasnet: platform-aware neural architecture search for mobile. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2820–2828 (2019)
41. Tan, M., Le, Q.: EfficientNet: rethinking model scaling for convolutional neural networks. In: International Conference on Machine Learning, pp. 6105–6114 (2019)
42. Tian, Y., Krishnan, D., Isola, P.: Contrastive representation distillation. In: Eighth International Conference on Learning Representations (2020)
43. Touvron, H., Vedaldi, A., Douze, M., Jégou, H.: Fixing the train-test resolution discrepancy. In: Advances in Neural Information Processing Systems, pp. 8250–8260 (2019)
44. Tung, F., Mori, G.: Similarity-preserving knowledge distillation. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1365–1374 (2019)
45. Wah, C., Branson, S., Welinder, P., Perona, P., Belongie, S.: The Caltech-UCSD Birds-200-2011 Dataset (2011)
46. Wang, T., Yuan, L., Zhang, X., Feng, J.: Distilling object detectors with fine-grained feature imitation. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4933–4942 (2019)
47. Wolf, L., Hassner, T., Maoz, I.: Face recognition in unconstrained videos with matched background similarity. In: CVPR 2011, pp. 529–534. IEEE (2011)
48. Wolf, T., et al.: Transformers: state-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45 (2020)
49. Xu, G., Liu, Z., Li, X., Loy, C.C.: Knowledge distillation meets self-supervision. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J.-M. (eds.) ECCV 2020. LNCS, vol. 12354, pp. 588–604. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58545-7_34
50. Yim, J., Joo, D., Bae, J., Kim, J.: A gift from knowledge distillation: fast optimization, network minimization and transfer learning. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 4133–4141 (2017)
51. Yuan, L., Tay, F.E., Li, G., Wang, T., Feng, J.: Revisiting knowledge distillation via label smoothing regularization. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 3903–3911 (2020)
52. Zagoruyko, S., Komodakis, N.: Paying more attention to attention: improving the performance of convolutional neural networks via attention transfer. In: Fifth International Conference on Learning Representations (2017)
53. Zhang, Y., Lan, Z., Dai, Y., Zeng, F., Bai, Y., Chang, J., Wei, Y.: Prime-aware adaptive distillation. In: Vedaldi, A., Bischof, H., Brox, T., Frahm, J.-M. (eds.) ECCV 2020. LNCS, vol. 12364, pp. 658–674. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58529-7_39
54. Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., Novik, G.: Neural Network Distiller: A Python Package for DNN Compression Research. arXiv preprint arXiv:1910.12232 (2019)