UNDER REVIEW

RESCIENCE C

Replication / Representation Learning

[Re] A Simple Framework for Contrastive Learning of Visual Representations

Alexandre Devillers[1] and Mathieu Lefort[1]
[1]Univ Lyon, UCBL, CNRS, INSA Lyon, LIRIS, UMR5205, F-69622 Villeurbanne, France,

Edited by
(Editor)

Reviewed by
(Reviewer 1)
(Reviewer 2)

Received
–

Published
–

DOI
–

Sections marked with an ‡ are identical or exhibit substantial similarity to corresponding sections in our companion reproduction paper [1] on "Bootstrap Your Own Latent: A new approach to self-supervised learning" [2]. This duplication is a consequence of the shared challenges and common methodologies inherent in the reproduction of the two closely related methods.

1 Reproducibility Summary

1.1 Scope of Reproducibility‡

In the course of our research, we undertook the task of reimplementing several methods of self-supervised learning of visual representations. This report details our efforts to reproduce parts of the article "A Simple Framework for Contrastive Learning of Visual Representations" [3], which introduced SimCLR in 2020. The core concept of these self-supervised approach is to learn meaningful representations by forcing similar embeddings for pairs of similar synthetic views of images. The ultimate goal of these representations is to transfer effectively to downstream tasks. Consequently, the evaluation of the representations involves linear evaluation, few-shot learning, and transfer learning. In this report, our focus is solely on reproducing the results related to the linear evaluation protocol on CIFAR10 and ImageNet. Additionally, we developed a code that is easily extensible, as well as adapted for high-performance computing.

1.2 Methodology‡

As for all the methods we reimplemented, we began by re-implementing SimCLR using the CIFAR10 dataset for preliminary feedback, as it is a small dataset, before transitioning to the ImageNet dataset.

Throughout the development process, we heavily relied on the information provided in the original paper, which explains the approach in detail, as well as various code bases, including the official one[1,2] written in TensorFlow. Additionally, we benefited from tutorials provided by the Jean Zay[3] cluster for the HPC (High-Performance Computing) aspects.

For the implementation, we utilized PyTorch as the machine learning framework, with support for multi-GPU and multi-node computation on the Slurm partition cluster of Jean Zay. We also ensured the code could run on single GPU and CPU setups. This proved useful during debugging processes on personal computers. Numerous experiments and runs were performed throughout the process, totaling around 30,000 GPU hours[4].

[1]https://github.com/google-research/simclr
[2]Note that, in the official repository, SimCLR is entangled with SimCLRv2.
[3]http://www.idris.fr/eng/jean-zay/jean-zay-presentation-eng.html
[4]In the following, *real hours* is used in opposition to *GPU hours* (which is the number of real hours times the number of GPU).

Correspondence should be addressed to Alexandre Devillers (alexandre.devillers@liris.cnrs.fr)
The authors have declared that no competing interests exists.

1.3 Results

We successfully reimplemented SimCLR with support for the Jean Zay supercomputer. Our results show that our reproduction even surpasses the original implementation in all our experiments.

1.4 What was easy‡

The original method was well-described in the article, making it easy to implement various aspects such as the architecture, hyperparameters, data augmentation policy, and the overall training procedure.

1.5 What was difficult‡

First, the optimizer used by these methods, including SimCLR, is not readily available in PyTorch, and existing repositories lack control over certain parameters, thus, it required its own reproduction.

Then, although the article provides significant precision and details, it still lacks some tips and tricks that were found either in the source code or in the implementations of other related methods in the literature. That was particularly a challenge as those implementation details belonged to unknown unknowns.

Finally, adapting the code for the Slurm (workload manager) infrastructure was challenging. Ensuring multi-node and multi-GPU support and respecting the HPC standards was far from straightforward.

1.6 Communication with original authors‡

We did not seek to contact the authors of the original article. As mentioned, the details provided in the article were sufficiently clear for us to re-implement the method when presented. Our most significant challenge was adapting the methods for high-performance computing, and we believed that seeking help from the authors would have not been particularly relevant for this specific aspect.

2 Introduction‡

The objective of self-supervised representation learning methods, such as SimCLR [3], is to pretrain a model without human supervision and enable it to learn meaningful features [3, 4, 5, 6]. These representations must fulfill two key requirements [7]: they should contain information relevant to downstream tasks (typically object recognition), which are unknown at training time, and they should possess a simple structure, to ensure that the representations can be easily utilized (either with a lightweight architecture model and/or with limited labeled data). Additionally, these representations should be applicable to out-of-domain datasets. As a result, the models should demonstrate good performance in linear evaluation, few-shot learning, and transfer to other datasets or tasks. For more comprehensive details on the current state-of-the-art methods and how the representations are evaluated, we recommend referring to [8].

3 Reproduced Method (SimCLR)

SimCLR, introduced in "A Simple Framework for Contrastive Learning of Visual Representations" [3], is a self-supervised representation learning method initially designed for the visual modality. It can be seen as the starting point of the recent interest in contrastive learning observed in the SSL (Self-Supervised Learning) field.

The fundamental idea behind SimCLR is to encourage high alignment in the representation space for pairs of images that share similar concepts. To achieve this without human supervision, positive pairs are generated through well chosen data augmentations. Additionally, to prevent the collapse of all data points to a single representation, embeddings of negative pairs are pushed away from each other. These negative pairs are all the other non-positive augmented images within the same batch.

Although the principles used in SimCLR were not entirely new, the novelty of the approach lay in combining and applying them through extensive experiments. Some of SimCLR's principles include using LARS (Layer-wise Adaptive Rate Scaling) optimizer, cosine learning rate scheduling, and a MLP (Multi-Layer Perceptron) projection head.

More formally, let t and t' signify two augmentations drawn from the distribution of augmentations $\mathcal{T}$. Two views are defined for the supplied input picture $\boldsymbol{x}$ as $\boldsymbol{v}_i := t(\boldsymbol{x})$ and $\boldsymbol{v}_j := t'(\boldsymbol{x})$. Thus, for N original images, this results in a batch of $2N$ views, with the first N items corresponding to a first view ($\boldsymbol{v}_i$), while the final N elements correspond to a second view ($\boldsymbol{v}_j$) for each image. Then, note f_θ, the encoder parameterized by θ, that produces representations from pictures, and g_ϕ, the projection head parameterized by ϕ, that projects representations in an embedding space. Thus, the representations are defined as $\boldsymbol{h}_i := f_\theta(\boldsymbol{v}_i)$ and $\boldsymbol{h}_j := f_\theta(\boldsymbol{v}_j)$, and the embeddings as $\boldsymbol{z}_i := g_\phi(\boldsymbol{h}_i)$ and $\boldsymbol{z}_j := g_\phi(\boldsymbol{h}_j)$. The model then learns to maximize the similarity between $\boldsymbol{z}_i$ and $\boldsymbol{z}_j$ while employing the negative-pairs trick to keep the embeddings' entropy high, preventing collapse to constant representations. This is done via the NT-Xent (Normalized Temperature-scaled cross entropy) loss, defining the loss for the positive pair (i, j) as:

$$\ell_{i,j} = -\log \frac{\exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_j)/\tau)}{\sum_{k=1}^{2N} \mathbb{1}_{[k \neq i]} \exp(\mathrm{sim}(\boldsymbol{z}_i, \boldsymbol{z}_k)/\tau)} \tag{1}$$

where τ is a temperature parameter, $\mathrm{sim}(\boldsymbol{a}, \boldsymbol{b}) = \boldsymbol{a}^\top \boldsymbol{b}/(\|\boldsymbol{a}\|\|\boldsymbol{b}\|)$ is the cosine similarity, and $\mathbb{1}_{[k \neq i]}$ is the indicator function evaluated to 1 (0 otherwise) when $k \neq i$. Finally, the total loss of the model is defined as the loss Eq. 1 applied to all pairs, both (i, j) and (j, i), of the batch.

To evaluate the learned representations, SimCLR performs the usual linear evaluation, where the representations are frozen, and a linear classifier is trained on top of them. Furthermore, they measure label efficiency through few-shot learning, where only a subset (e.g., 10% or 1%) of the labels is used. Finally, the transferability of the representations is studied by assessing how well they perform on datasets different from, or even outside the domain of, the training set. As a reminder, this report will only focus on the linear evaluation.

4 Reproducibility Objectives[‡]

The objective of this reproduction was to develop a flexible and modifiable version of SimCLR, among other SSL methods, for studying the incorporation of an equivariance module to enhance performance of recent SSL approaches [9]. Therefore, our primary focus was on replicating similar results for the linear evaluation on CIFAR10 and ImageNet, which is a widely used protocol in the literature for comparing existing methods. Additionally, since this project utilized the computational resources provided by the supercomputer Jean Zay, we designed our code to be scalable and specifically optimized for Jean Zay's HPC system.

5 Difficulties encountered

5.1 Difficulties related to recent SSL

LARS with parameter exclusion[‡] — The recent SSL methods rely on using large batch sizes, either to prevent collapse by requiring numerous negative pairs or to accelerate the training process [3, 4, 5, 6]. However, training with such large batch sizes using stochastic gradient descent can lead to instability. To mitigate this, it is common to utilize the LARS optimizer [10]. During the training process of these methods, certain parameters should be excluded from the LARS adaptation, as well as from weight decay. However, existing PyTorch implementations of LARS lack the flexibility for such control. One workaround is to employ SGD (Stochastic Gradient Descent) for the parameters that are excluded from LARS adaptation, but this requires managing two separate optimizers, leading to additional complications. To overcome these challenges, we've chosen to develop our custom LARS implementation. This version introduces a weight parameter to regulate the strength of the LARS adaptation applied (i.e., a value of 0.0 signifies the use of SGD, while 1.0 represents the original LARS). This parameter can be set for a group of parameters, enabling the straightforward exclusion of certain parameters from the LARS adaptation.

No bias before batch normalization[‡] — An often omitted detail, as in the original article, yet observed in many codebases (including the official one), is the absence of bias in layers followed by a BN (Batch Normalization) [11]. While this is already implemented in PyTorch's version of ResNets, it must also be applied to the projection head. The reasoning behind this is that the batch normalization layer will center back the data, canceling the bias [11]. Thus, it does not improve the computational capacity of the model, while it adds an extra parameter, which subsequently complicates the optimization process.

Interpolation mode of the resize augmentation[‡] — Another detail we uncovered by reviewing various codebases, together with the official one, is that the interpolation method used in the resize augmentation is not the default one. These codebases appeared to predominantly use bicubic interpolation, which yielded superior results in our experiments.

Zero-initializing the residual branches[‡] — Lastly, another unmentioned technique involves initializing to zero the last BN layer of the residual branches for the ResNet backbone. We discovered this strategy in the official codebase, as well as in some other implementations, and found that it enhances performance.

5.2 Difficulties related to HPC

PyTorch's slow cosine similarity[‡] — The native cosine operation provided by PyTorch is slow and used to be a significant bottleneck. This issue was addressed by executing the computation using PyTorch's primitives. This process involves the normalization of the embeddings matrix, followed by the matrix multiplication of this matrix with its transpose.

Loss simplification and repetitive tensor creation — The loss computations of SimCLR can be expressed as a cross-entropy of the normalized similarity matrix (i.e., divided by the temperature parameter τ). In this computation, the diagonal is masked (to follow the indicator function, see Eq. 1), and the target index of the cross-entropy is the index of the positive pair (i.e., j for $\ell_{i,j}$). This method substantially speeds up the loss computations, compared to more straightforward implementations, as it can be computed on the whole batch through mostly matrix operations.

Moreover, as the mask used in this computation, as well as the targets, are only dependent on the batch size, it can be cached. This strategy eliminates the need for recreating it during each loss computation, which is relatively slow on GPU, further enhancing the computational efficiency.

These optimizations significantly reduce the computational time required for loss computation compared to more naive implementations. After testing many different implementations of the NT-Xent loss, both ones we crafted and those from reviewed codebases, we found this approach to be the most efficient, probably because it does not require slow tensor operations such as creation, reshape, concatenation, or view.

Compute the loss on a sharded batch — SimCLR's loss requires negative pairs from the batch. However, in multi-GPU computations, the batch is distributed across multiple devices. Consequently, each device must perform a gather operation to collect all embeddings from all other devices and locally compute the loss. But this only backpropagates through local embeddings as the gather operation stops gradient for non-local tensors. As a result, the gradient magnitude is divided by the number of devices (world size). This issue is prevalent in most public codebases. Yet, a straightforward solution to counter this effect is to multiply the loss by the number of devices.

Batch norm multi-node synchronization‡ — Analogous to the previous section, the batch normalization computation requires the entire batch to be accurately performed. Thankfully, PyTorch offers a solution by providing a way to convert all existing batch normalization layers of a model to synchronized batch normalization, effectively solving this issue. This problem was highlighted in the original SimCLR paper, which also proposed other, more complex solutions.

Jobs too long and checkpoints‡ — A final, yet significant issue, relates to the quality-of-service offered by the Jean Zay supercomputer. Due to the large number of GPUs and nodes required for the experiments, the only suitable quality-of-service allows for jobs with a maximum duration of 20 real hours. However, typical ImageNet training takes approximately 170 real hours. Therefore, the jobs had to be split multiple times during a single training session. This necessitated the implementation of a checkpoint system to resume training after each job interruption. The system had to handle the saving and loading of the epoch, model, optimizer, and learning rate scheduler.

6 Experimental settings

6.1 Datasets‡

We began our experiments by working with the CIFAR10 dataset [12], which was provided via the torchvision package. We chose to start with this dataset because of its small size, which allowed for quick iterations during our reimplementation. CIFAR10 consists of 50,000 training images and 10,000 test images, all of which are 32×32 pixels in size. The images are categorized into 10 classes, representing various animals and vehicles.

Next, we conducted additional experiments using the ImageNet dataset [13], which was made available through Jean Zay. ImageNet is the standard benchmark for comparison in visual representation learning, and thus, is used in the original article. This dataset comprises 1,000 classes of natural images with varying resolutions. It consists of 1.28 million training images and 100,000 test images.

It is worth noting that throughout the process of learning representations, the labels of the data were not used, as the objective was to achieve self-supervised learning.

6.2 Architectures‡

The architectures vary depending on the dataset, but in all cases, utilize a ResNet [14] as the backbone. For CIFAR10, a ResNet18 architecture is used with the usual CIFAR10 modifications: the first convolutional layer has a 3×3 kernel with a stride of 1, and the max-pooling layer is removed. On the other hand, for the ImageNet dataset, a standard ResNet50 architecture is employed. Please note that in both instances, the final linear layer is removed and replaced by the projection head. This projection head is composed of 2 fully-connected layers, yet, we also used a three-layer architecture as described in [6] because it demonstrates increased robustness to other hyperparameters (both results are in Sec. 1.3). Each projection head's layer is followed by a BN and a ReLU activation function, except for the last layer, which does not have a ReLU. Hidden layers consist of 2048 neurons.
For the ResNet, we based our implementation on the torchvision library, which provides a convenient way to instantiate ResNet models. We then modified these models' architectures as previously described.

6.3 Hyperparameters‡

When available, we use the original implementation's hyperparameters, therefore, the models are trained for 800 epochs using a cosine decay learning rate schedule and 10 warm-up epochs. A batch size of 4096 is used for ImageNet and 512 for CIFAR10, with an initial learning rate of 4.8 for ImageNet and 4.0 for CIFAR10. The momentum is set to 0.9 and the weight decay to 10^{-6} for the optimizer. The dimensions of the embedding space have been set to 128. For the loss, τ is set to 0.2 on ImageNet and $\tau = 0.5$ with CIFAR10. Finally, the LARS [10] optimizer is used for all experiments, although biases and BN parameters were removed from both weight decay and LARS adaptation.

6.4 Augmentation policy‡

We reimplemented the distinct augmentation policies for the CIFAR10 and ImageNet datasets. Note that all these augmentations are provided by the torchvision library.

For the CIFAR10 dataset, a random resized crop is applied, which generates output images of 32×32 pixels. This transformation features a scale parameter ranging from 0.2 to 1.0 and an aspect ratio that varies from $3/4$ to $4/3$. As previously noted in the challenges section, bicubic interpolation is utilized for the resizing operation. Subsequently, the cropped image undergoes a potential horizontal flip, determined probabilistically with a 50% likelihood. This was followed by a color jitter transformation with 80% probability, altering the brightness, contrast, saturation, and hue parameters by up to 0.4, 0.4, 0.4, and 0.1 respectively. Images were also converted to grayscale with a 20% probability. The final step was the normalization of image tensors using mean $(0.4914, 0.4822, 0.4465)$ and standard deviation $(0.247, 0.243, 0.261)$.

For the ImageNet dataset, the training augmentations was similar to the one used for CIFAR10, but with a few key differences. The random resized crop generated images of size 224×224 pixels, with scale ranging from 0.08 to 1.0. The parameters for color jitter were adjusted to 0.8 for brightness, contrast, and saturation, while 0.2 for hue. Furthermore, a Gaussian blur transformation was applied with a 50% probability, using a kernel size that is 10% of the image's dimensions (i.e., 23×23), and sigma ranging from 0.1 to 2.0. The final normalization step used mean $(0.485, 0.456, 0.406)$ and standard deviation $(0.229, 0.224, 0.225)$.

6.5 Linear evaluation‡

The linear evaluation of the trained models is performed using SGD optimizer with Nesterov momentum and cross-entropy loss. The learning rate is initialized to 0.2 and follows a cosine scheduler to adjust it over the 90 epochs of learning. The batch size for this phase is 256. The momentum parameter is set to 0.9 and no weight decay is applied, reflecting a standard configuration in such scenarios.

For this evaluation phase, the augmentation policies are also modified for both CIFAR10 and ImageNet datasets. For CIFAR10 during the training phase, color and grayscale transformations are removed, and the lower bound of the scale parameter in the random resized crop operation is reduced to 0.08. For the testing phase, the random resized crop and random horizontal flip are replaced by a resize operation to 36 pixels, followed by a center crop to 32 pixels.

In the case of ImageNet, that training phase sees the removal of color, grayscale, and blur transformations. For the testing phase, similar to CIFAR10, transformations are replaced, this time by a resize operation to 256 pixels, and a subsequent center crop to 224 pixels.

6.6 Resources, setup, and code‡

Experiments were conducted using the Jean Zay supercomputer, which employs Slurm for cluster management. The code, written in Python, leverages PyTorch 1.10.0 and TorchVision 0.11.0, CIFAR10 experiments were executed on a single node equipped with 2 Tesla V100 32GB GPUs for approximately 4 real hours of training, while ImageNet experiments utilized 8 nodes, each with 4 Tesla V100 32GB GPUs during 170 real hours of training.

7 Detailed results

7.1 Results reproducing original paper

Tab. 1 summarizes the results obtained from different datasets on one run (due to computational constraints), alongside the original results. Note that for CIFAR10, the original paper only mentioned results using ResNet50, which deviates from the literature standard of using the adapted ResNet18 previously mentioned. However, the official GitHub implementation does mention that with this last ResNet18 architecture, they achieved "around 91% accuracy", which is the result we've reported in Tab. 1. Unfortunately, they did not mention the Top-5 accuracy for this setup.

Our primary focus during the development process was to replicate the results on CIFAR10 before attempting to scale the experiments to ImageNet. As Tab. 1 indicates, we were successful in replicating the results on CIFAR10, as our re-implementation seems to perform better than the official one. Upon conducting experiments on ImageNet, we achieved results that surpassed the anticipated ones, thus reproducing and improving upon the original results for both Top-1 and Top-5 accuracy. Additionally, experiments conducted with a three-layer (instead of two-layer) projection head improved performance in all our tested setups.

Altogether, these results affirm the reproducibility of SimCLR, and suggest that our reimplementation, tailored for the Jean Zay supercomputer, was successful.

Implementations	CIFAR10		ImageNet	
	Top-1	Top-5	Top-1	Top-5
Original	~ 91.0	-	69.3	89.0
Our	91.70	99.71	70.78	90.36
Our (3-layer projection head)	91.93	99.79	71.84	90.53

Table 1. **Linear Evaluation**; SimCLR's implementations top-1 and top-5 accuracies (in %) under linear evaluation on CIFAR10 and ImageNet.

8 Conclusion

In this report, we described our efforts to re-implement SimCLR, and to reproduce the linear evaluation results, while developing a code base that is adaptable and compatible with the high-performance computing resources provided by the Jean Zay cluster.

Our initial efforts were focused on the CIFAR10 dataset. These efforts were successful, resulting in the replication of the reported results. Then, we moved on to the larger and more complex ImageNet dataset. Here, we not only matched the original performance metrics but surpassed them, therefore effectively achieving our goal of SimCLR reproduction. The clarity of the original paper and the resources available in public repositories, both original and others, played a crucial role in enabling us to reproduce these results, despite the various challenges we encountered.

It is crucial to acknowledge that these methodologies are computationally demanding, requiring the simultaneous utilization of numerous GPUs for extensive periods. Without the support of Jean Zay, carrying out this work would have been unfeasible. In addition to our gratitude towards Jean Zay, we take pride in contributing to the scientific community by sharing the weights of our trained models (while the official ones of SimCLR are available on the official GitHub). This gesture is aimed at sparing others the significant costs, time, and energy that would otherwise be needed to replicate these computationally intense training processes.

9 Acknowledgements

This work was performed using HPC resources from GENCI-IDRIS (Grant 2021-AD011013160, 2022-A0131013831, and 2022-AD011013646) and GPUs donated by the NVIDIA Corporation. We gratefully acknowledge this support.

References

1. A. Devillers and M. Lefort. “[Re] Bootstrap Your Own Latent: A new approach to self-supervised learning.” In: **ReScience C, Under Review** (2023).
2. J.-B. Grill, F. Strub, F. Altché, C. Tallec, P. H. Richemond, E. Buchatskaya, C. Doersch, B. A. Pires, Z. D. Guo, and M. G. Azar. “Bootstrap your own latent: A new approach to self-supervised learning.” In: **arXiv preprint arXiv:2006.07733** (2020).
3. T. Chen, S. Kornblith, M. Norouzi, and G. Hinton. “A simple framework for contrastive learning of visual representations.” In: **International conference on machine learning**. PMLR, 2020, pp. 1597–1607.
4. J. Zbontar, L. Jing, I. Misra, Y. LeCun, and S. Deny. “Barlow twins: Self-supervised learning via redundancy reduction.” In: **arXiv preprint arXiv:2103.03230** (2021).
5. A. Bardes, J. Ponce, and Y. LeCun. “Vicreg: Variance-invariance-covariance regularization for self-supervised learning.” In: **arXiv preprint arXiv:2105.04906** (2021).
6. X. Chen and K. He. “Exploring simple siamese representation learning.” In: **Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition**. 2021, pp. 15750–15758.
7. T. Wang and P. Isola. “Understanding contrastive representation learning through alignment and uniformity on the hypersphere.” In: **International Conference on Machine Learning**. PMLR, 2020, pp. 9929–9939.
8. R. Liu. “Understand and Improve Contrastive Learning Methods for Visual Representation: A Review.” In: **arXiv preprint arXiv:2106.03259** (2021).
9. A. Devillers and M. Lefort. “Equimod: An equivariance module to improve self-supervised learning.” In: **arXiv preprint arXiv:2211.01244** (2022).
10. Y. You, I. Gitman, and B. Ginsburg. “Large batch training of convolutional networks.” In: **arXiv preprint arXiv: 1708.03888** (2017).
11. S. Ioffe and C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift.” In: **International conference on machine learning**. pmlr. 2015, pp. 448–456.
12. J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. “Imagenet: A large-scale hierarchical image database.” In: **2009 IEEE conference on computer vision and pattern recognition**. Ieee. 2009, pp. 248–255.
13. A. Krizhevsky, G. Hinton, et al. “Learning multiple layers of features from tiny images.” In: (2009).
14. K. He, X. Zhang, S. Ren, and J. Sun. “Deep residual learning for image recognition.” In: **Proceedings of the IEEE conference on computer vision and pattern recognition**. 2016, pp. 770–778.