SimSiam

SimSiam (Simple Siamese Representation Learning) [0] is a self-supervised learning framework for visual representation learning that eliminates the need for negative samples, large batches, or momentum encoders. Instead, SimSiam directly optimizes the similarity between two augmented views of an image. It employs a simple Siamese architecture where these augmented views are processed by a shared encoder, with a prediction MLP on one branch and a stop-gradient operation on the other. SimSiam challenges conventional beliefs regarding collapsing representations by demonstrating that only the stop-gradient mechanism is essential for preventing collapse, rather than relying on momentum encoders or architectural modifications. Experimental results highlight the crucial role of the predictor layer and the application of batch normalization in hidden layers for stable training and improved representation quality. Furthermore, unlike SimCLR [1] and SwAV [2], SimSiam performs robustly across a wide range of batch sizes.

Key Components

- **Data Augmentations**: SimSiam employs the same augmentations as SimCLR, including random resized cropping, horizontal flipping, color jittering, Gaussian blur, and solarization. These augmentations provide diverse views of an image for representation learning.
- **Backbone**: SimSiam utilizes ResNet-type architectures as the encoder network. The model does not employ a momentum encoder.
- **Projection & Prediction Head**: A projection MLP maps the encoder output to a lower-dimensional space, followed by a prediction MLP on one branch. The stop-gradient operation is applied to the second branch to prevent collapse.
- **Loss Function**: SimSiam minimizes the negative cosine similarity between the predicted representation of one view and the projected representation of the other, with a symmetrical loss formulation. It also works for a symmetrized cross-entropy loss.

Good to Know

- **Backbone Networks**: SimSiam is specifically optimized for convolutional neural networks, with a focus on ResNet architectures. We do not recommend using it with transformer-based models and instead suggest using DINO [3].
- **Relation to SimCLR**: SimSiam can be thought of as “SimCLR without negative pairs.”
- **Relation to SwAV**: SimSiam is conceptually analogous to “SwAV without online clustering.”

- **Relation to BYOL** [4]: SimSiam can be considered a variation of BYOL that removes the momentum encoder subject to many implementation differences.

Reference:

[0] Exploring Simple Siamese Representation Learning, 2020

[1] A Simple Framework for Contrastive Learning of Visual Representations, 2020

[2] Unsupervised Learning of Visual Features by Contrasting Cluster Assignments, 2020

[3] Emerging Properties in Self-Supervised Vision Transformers, 2021

[4] Bootstrap Your Own Latent: A New Approach to Self-Supervised Learning, 2020

Tutorials:

Tutorial 4: Train SimSiam on Satellite Images

PyTorch   Lightning   Lightning Distributed

Open in Colab

This example can be run from the command line with:

```
python lightly/examples/pytorch/simsiam.py
```

```python
# This example requires the following dependencies to be installed:
# pip install lightly

# Note: The model and training settings do not follow the reference settings
# from the paper. The settings are chosen such that the example can easily be
# run on a small dataset with a single GPU.

import torch
import torchvision
from torch import nn

from lightly.loss import NegativeCosineSimilarity
from lightly.models.modules import SimSiamPredictionHead, SimSiamProjectionHead
from lightly.transforms import SimSiamTransform


class SimSiam(nn.Module):
    def __init__(self, backbone):
        super().__init__()
        self.backbone = backbone
        self.projection_head = SimSiamProjectionHead(512, 512, 128)
        self.prediction_head = SimSiamPredictionHead(128, 64, 128)

    def forward(self, x):
        f = self.backbone(x).flatten(start_dim=1)
        z = self.projection_head(f)
        p = self.prediction_head(z)
        z = z.detach()
        return z, p


resnet = torchvision.models.resnet18()
backbone = nn.Sequential(*list(resnet.children())[:-1])
model = SimSiam(backbone)

device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)

transform = SimSiamTransform(input_size=32)
dataset = torchvision.datasets.CIFAR10(
    "datasets/cifar10", download=True, transform=transform
)
# or create a dataset from a folder containing images or videos:
# dataset = LightlyDataset("path/to/folder", transform=transform)

dataloader = torch.utils.data.DataLoader(
    dataset,
    batch_size=256,
    shuffle=True,
    drop_last=True,
    num_workers=8,
)

criterion = NegativeCosineSimilarity()
optimizer = torch.optim.SGD(model.parameters(), lr=0.06)

print("Starting Training")
for epoch in range(10):
    total_loss = 0
    for batch in dataloader:
        x0, x1 = batch[0]
        x0 = x0.to(device)
        x1 = x1.to(device)
        z0, p0 = model(x0)
        z1, p1 = model(x1)
        loss = 0.5 * (criterion(z0, p1) + criterion(z1, p0))
        total_loss += loss.detach()
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
    avg_loss = total_loss / len(dataloader)
    print(f"epoch: {epoch:>02}, loss: {avg_loss:.5f}")
```