lightly.loss

The lightly.loss package provides loss functions for self-supervised learning.

class **lightly.loss.barlow_twins_loss.BarlowTwinsLoss**(*lambda_param: float = 0.005, gather_distributed: bool = False*)

Implementation of the Barlow Twins Loss from Barlow Twins[0] paper.

This code specifically implements the Figure Algorithm 1 from [0]. [0] Zbontar,J. et.al, 2021, Barlow Twins... https://arxiv.org/abs/2103.03230

Examples

```
>>> # initialize loss function
>>> loss_fn = BarlowTwinsLoss()
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through SimSiam model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*z_a: Tensor*, *z_b: Tensor*) → Tensor

Computes the Barlow Twins loss for the given projections.

Parameters
- **z_a** – Output projections of the first set of transformed images.
- **z_b** – Output projections of the second set of transformed images.

Returns Computed Barlow Twins Loss.

class **lightly.loss.dcl_loss.DCLLoss**(*temperature: float = 0.1, weight_fn: Optional[Callable[[Tensor, Tensor], Tensor]] = None, gather_distributed: bool = False*)

Implementation of the Decoupled Contrastive Learning Loss from Decoupled Contrastive Learning [0].

This code implements Equation 6 in [0], including the sum over all images *i* and views *k*. The loss is reduced to a mean loss over the mini-batch. The implementation was inspired by [1].

- [0] Chun-Hsiao Y. et. al., 2021, Decoupled Contrastive Learning https://arxiv.org/abs/2110.06848
- [1] https://github.com/raminnakhli/Decoupled-Contrastive-Learning

temperature

Similarities are scaled by inverse temperature.

weight_fn

Weighting function *w* from the paper. Scales the loss between the positive views (views from the same image). No weighting is performed if weight_fn is None. The function must take the two input tensors passed to the forward call as input and return a weight tensor. The returned weight tensor must have the same length as the input tensors.

gather_distributed

If True, negatives from all GPUs are gathered before the loss calculation.

Examples

```
>>> loss_fn = DCLLoss(temperature=0.07)
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # embed images using some model, for example SimCLR
>>> out0 = model(t0)
>>> out1 = model(t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
>>>
>>> # you can also add a custom weighting function
>>> weight_fn = lambda out0, out1: torch.sum((out0 - out1) ** 2, dim=1)
>>> loss_fn = DCLLoss(weight_fn=weight_fn)
```

forward(*out0: Tensor*, *out1: Tensor*)→ Tensor

Forward pass of the DCL loss.

Parameters
- **out0** – Output projections of the first set of transformed images. Shape: (batch_size, embedding_size)
- **out1** – Output projections of the second set of transformed images. Shape: (batch_size, embedding_size)

Returns Mean loss over the mini-batch.

class **lightly.loss.dcl_loss.DCLWLoss**(*temperature: float = 0.1, sigma: float = 0.5, gather_distributed: bool = False*)

Implementation of the Weighted Decoupled Contrastive Learning Loss from Decoupled Contrastive Learning [0].

This code implements Equation 6 in [0] with a negative Mises-Fisher weighting function. The loss returns the mean over all images *i* and views *k* in the mini-batch. The implementation was inspired by [1].

- [0] Chun-Hsiao Y. et. al., 2021, Decoupled Contrastive Learning https://arxiv.org/abs/2110.06848
- [1] https://github.com/raminnakhli/Decoupled-Contrastive-Learning

temperature

Similarities are scaled by inverse temperature.

sigma

Similar to temperature but applies the inverse scaling in the weighting function.

gather_distributed

If True, negatives from all GPUs are gathered before the loss calculation.

Examples

```
>>> loss_fn = DCLWLoss(temperature=0.07)
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # embed images using some model, for example SimCLR
>>> out0 = model(t0)
>>> out1 = model(t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

class **lightly.loss.detcon_loss.DetConBLoss**(*temperature: float = 0.1, gather_distributed: bool = True*)

Implementation of the DetConB loss. [0]

The inputs are three-fold:

- Two latent representations of the same batch under different views, as generated by BYOL's [1] prediction branch and additional pooling over the regions of the segmentation.

- Two latent representations of the same batch under different views, as generated by BYOL's target branch and additional pooling over the regions of the segmentation.
- Two integer masks that indicate the regions of the segmentation that were used for pooling.

For calculating the contrastive loss, regions under the same mask in the same image (under a different view) are considered as positives and everything else as negatives. With v_m and $v'_{m'}$ being the pooled feature maps under mask m and m' respectively, and additionally scaled to a norm of $\frac{1}{\sqrt{\tau}}$, the formula for the contrastive loss is

$$\mathcal{L} = \sum_m \sum_{m'} \mathbb{1}_{m,m'} \left[-\log \frac{\exp(v_m \cdot v'_{m'})}{\exp(v_m \cdot v'_{m'}) + \sum_n \exp(v_m \cdot v'_{m'})} \right]$$

where $\mathbb{1}_{m,m'}$ is 1 if the masks are the same and 0 otherwise. Since v_m and $v'_{m'}$ stem from different branches, the loss is symmetrized by also calculating the loss with the roles of the views reversed. [1]

References

[0] DetCon https://arxiv.org/abs/2103.10957

[1] (1, 2) BYOL https://arxiv.org/abs/2006.07733

`temperature`

The temperature τ in the contrastive loss.

`gather_distributed`

If True, the similarity matrix is gathered across all GPUs before the loss is calculated. Else, the loss is calculated on each GPU separately.

`forward`(*pred_view0: Tensor, pred_view1: Tensor, target_view0: Tensor, target_view1: Tensor, mask_view0: Tensor, mask_view1: Tensor*)→ Tensor

Calculate the contrastive loss under the same mask in the same image.

The tensor shapes and value ranges are given by variables B, M, D, N, where B is the batch size, M is the sampled number of image masks / regions, D is the embedding size and N is the total number of masks.

Parameters

- **pred_view0** – Mask-pooled output of the prediction branch for the first view, a float tensor of shape (B, M, D).
- **pred_view1** – Mask-pooled output of the prediction branch for the second view, a float tensor of shape (B, M, D).

- **target_view0** – Mask-pooled output of the target branch for the first view, a float tensor of shape (B, M, D).
- **target_view1** – Mask-pooled output of the target branch for the second view, a float tensor of shape (B, M, D).
- **mask_view0** – Indices corresponding to the sampled masks for the first view, an integer tensor of shape (B, M) with (possibly repeated) indices in the range $[0, N)$.
- **mask_view1** – Indices corresponding to the sampled masks for the second view, an integer tensor of shape (B, M) with (possibly repeated) indices in the range $[0, N)$.

Returns A scalar float tensor containing the contrastive loss.

---

class **lightly.loss.detcon_loss.DetConSLoss**(*temperature: float = 0.1, gather_distributed: bool = True*)

Implementation of the DetConS loss. [2]

The inputs are two-fold:

- Two latent representations of the same batch under different views, as generated by SimCLR [3] and additional pooling over the regions of the segmentation.
- Two integer masks that indicate the regions of the segmentation that were used for pooling.

For calculating the contrastive loss, regions under the same mask in the same image (under a different view) are considered as positives and everything else as negatives. With v_m and $v'_{m'}$ being the pooled feature maps under mask m and m' respectively, and additionally scaled to a norm of $\frac{1}{\sqrt{\tau}}$, the formula for the contrastive loss is

$$\mathcal{L} = \sum_m \sum_{m'} \mathbb{1}_{m,m'} \left[-\log \frac{\exp(v_m \cdot v'_{m'})}{\exp(v_m \cdot v'_{m'}) + \sum_n \exp(v_m \cdot v'_{m'})} \right]$$

where $\mathbb{1}_{m,m'}$ is 1 if the masks are the same and 0 otherwise.

References

[2] DetCon https://arxiv.org/abs/2103.10957

[3] SimCLR https://arxiv.org/abs/2002.05709

temperature

The temperature τ in the contrastive loss.

gather_distributed

If True, the similarity matrix is gathered across all GPUs before the loss is calculated. Else, the loss is calculated on each GPU separately.

forward(*view0: Tensor, view1: Tensor, mask_view0: Tensor, mask_view1: Tensor*)→ Tensor

Calculate the contrastive loss under the same mask in the same image.

The tensor shapes and value ranges are given by variables B, M, D, N, where B is the batch size, M is the sampled number of image masks / regions, D is the embedding size and N is the total number of masks.

Parameters

- **view0** – Mask-pooled output for the first view, a float tensor of shape (B, M, D).
- **pred_view1** – Mask-pooled output for the second view, a float tensor of shape (B, M, D).
- **mask_view0** – Indices corresponding to the sampled masks for the first view, an integer tensor of shape (B, M) with (possibly repeated) indices in the range $[0, N)$.
- **mask_view1** – Indices corresponding to the sampled masks for the second view, an integer tensor of shape (B, M) with (possibly repeated) indices in the range $[0, N)$.

Returns A scalar float tensor containing the contrastive loss.

class **lightly.loss.dino_loss.DINOLoss**(*output_dim: int = 65536, warmup_teacher_temp: float = 0.04, teacher_temp: float = 0.04, warmup_teacher_temp_epochs: int = 30, student_temp: float = 0.1, center_momentum: float = 0.9, center_mode: str = 'mean'*)

Implementation of the loss described in 'Emerging Properties in Self-Supervised Vision Transformers'. [0]

This implementation follows the code published by the authors. [1] It supports global and local image crops. A linear warmup schedule for the teacher temperature is implemented to stabilize training at the beginning. Centering is applied to the teacher output to avoid model collapse.

- [0]: DINO, 2021, https://arxiv.org/abs/2104.14294
- [1]: https://github.com/facebookresearch/dino

output_dim

Dimension of the model output.

teacher_temp

Temperature parameter for the teacher network.

student_temp

Temperature parameter for the student network.

center

Center used for the teacher output. It is updated with a moving average during training.

center_momentum

Momentum term for the center calculation.

warmup_teacher_temp_epochs

Number of epochs for the warmup phase of the teacher temperature (for backward compatibility).

teacher_temp_schedule

A linear schedule for the teacher temperature during the warmup phase (for backward compatibility).

Examples

```
>>> # initialize loss function
>>> loss_fn = DINOLoss(128)
>>>
>>> # generate two views of the images with a random transform
>>> view0, view1 = transform(images), transform(images)
>>>
>>> # embed the views with a student and teacher model
>>> teacher_out = [teacher(view0), teacher(view1)]
>>> student_out = [student(view0), student(view1)]
>>>
>>> # calculate loss
>>> loss = loss_fn(teacher_out, student_out)
```

forward(*teacher_out: list[Tensor], student_out: list[Tensor], teacher_temp: float | None = None, epoch: int | None = None*)→ Tensor

Cross-entropy between softmax outputs of the teacher and student networks.

Parameters

- **teacher_out** – List of tensors with shape (batch_size, output_dim) containing features from the teacher model. Each tensor must represent one view of the batch.
- **student_out** – List of tensors with shape (batch_size, output_dim) containing features from the student model. Each tensor must represent one view of the batch.

- **teacher_temp** – The temperature used for the teacher output. If None, the default temperature defined in __init__ is used.
- **epoch** – The current epoch for backward compatibility.

Returns The average cross-entropy loss.

update_center(*teacher_out: Tensor*)→ None

Moving average update of the center used for the teacher output.

Parameters **teacher_out** – Tensor with shape (num_views, batch_size, output_dim) containing features from the teacher model.

class **lightly.loss.hypersphere_loss.HypersphereLoss**(*t: float = 1.0, lam: float = 1.0, alpha: float = 2.0*)

Implementation of the loss described in 'Understanding Contrastive Representation Learning through Alignment and Uniformity on the Hypersphere.' [0]

[0] Tongzhou Wang. et.al, 2020, ... https://arxiv.org/abs/2005.10242

Note

In order for this loss to function as advertised, an L1-normalization to the hypersphere is required. This loss function applies this L1-normalization internally in the loss layer. However, it is recommended that the same normalization is also applied in your architecture, considering that this L1-loss is also intended to be applied during inference. Perhaps there may be merit in leaving it out of the inference pathway, but this use has not been tested.

Moreover it is recommended that the layers preceding this loss function are either a linear layer without activation, a batch-normalization layer, or both. The directly upstream architecture can have a large influence on the ability of this loss to achieve its stated aim of promoting uniformity on the hypersphere; and if by contrast the last layer going into the embedding is a RELU or similar nonlinearity, we may see that we will never get very close to achieving the goal of uniformity on the hypersphere, but will confine ourselves to the subspace of positive activations. Similar architectural considerations are relevant to most contrastive loss functions, but we call it out here explicitly.

Examples

```
>>> # initialize loss function
>>> loss_fn = HypersphereLoss()
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through SimSiam model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*z_a: Tensor*, *z_b: Tensor*) → Tensor

Computes the Hypersphere loss, which combines alignment and uniformity loss terms.

Parameters
- **z_a** – Tensor of shape (batch_size, embedding_dim) for the first set of embeddings.
- **z_b** – Tensor of shape (batch_size, embedding_dim) for the second set of embeddings.

Returns The computed loss.

class **lightly.loss.ibot_loss.IBOTPatchLoss**(*output_dim: int = 65536*, *teacher_temp: float = 0.04*, *student_temp: float = 0.1*, *center_mode: str = 'mean'*, *center_momentum: float = 0.9*)

Implementation of the iBOT patch loss [0] as used in DINOv2 [1].

Implementation is based on [2].

- [0]: iBOT, 2021, https://arxiv.org/abs/2111.07832
- [1]: DINOv2, 2023, https://arxiv.org/abs/2304.07193
- [2]: https://github.com/facebookresearch/dinov2/blob/main/dinov2/loss/ibot_patch_loss.py

output_dim

Dimension of the model output.

teacher_temp

Temperature for the teacher output.

student_temp

Temperature for the student output.

center_mode

Mode for center calculation. Only ‘mean’ is supported.

center_momentum

Momentum term for the center update.

forward(*teacher_out: Tensor, student_out: Tensor, mask: Tensor, teacher_temp: float | None = None*)→ Tensor

Forward pass through the iBOT patch loss.

Parameters

- **teacher_out** – Tensor with shape (batch_size * sequence_length, embed_dim) containing the teacher output of the masked tokens.
- **student_out** – Tensor with shape (batch_size * sequence_length, embed_dim) containing the student output of the masked tokens.
- **mask** – Boolean tensor with shape (batch_size, height, width) containing the token mask. Exactly batch_size * sequence_length entries must be set to True in the mask.
- **teacher_temp** – The temperature used for the teacher output. If None, the default temperature defined in __init__ is used.

Returns The loss value.

class **lightly.loss.ibot_loss.IBOTPlusPlusPatchLoss**(*output_dim: int = 65536, teacher_temp: float = 0.04, student_temp: float = 0.1, center_mode: str = 'mean', center_momentum: float = 0.9*)

Implementation of the iBOT++ patch loss from TIPSv2.

iBOT++ extends the iBOT masked patch loss by applying patch-level self-distillation to all patch tokens, including visible tokens. This is useful for dense downstream tasks because all patch features are directly anchored to the teacher distribution.

The loss expects full teacher and student patch logits with shape `(B, N, K)` where `B` is the batch size, `N` is the number of patch tokens, and `K` is the number of prototypes. Inputs with shape `(B * N, K)` are also supported when `mask` is provided to infer the batch size.

Examples

```
>>> criterion = IBOTPlusPlusPatchLoss(output_dim=8192)
>>> teacher_out = torch.randn(8, 196, 8192)
>>> student_out = torch.randn(8, 196, 8192)
>>> loss = criterion(teacher_out=teacher_out, student_out=student_out)
>>>
>>> ssl_loss = torch.tensor(1.0)
>>> loss = ssl_loss + 2.0 * criterion(teacher_out, student_out)
```

References

TIPSv2, 2026, https://arxiv.org/abs/2604.12012 iBOT, 2021, https://arxiv.org/abs/2111.07832

forward(*teacher_out: Tensor, student_out: Tensor, mask: Tensor | None = None, teacher_temp: float | None = None, visible_loss_weight: float = 1.0*) → Tensor

Forward pass through the iBOT++ patch loss.

When `mask` is provided, the per-token cross-entropy is split into a masked term and a visible term. The masked term is normalized by the number of masked tokens per image (matching the original iBOT masked image modeling signal), and the visible term is normalized by the number of visible tokens per image and scaled by `visible_loss_weight`:

```
loss = mean_b(masked_b + visible_loss_weight * visible_b)
```

Setting `visible_loss_weight=0` recovers the exact iBOT behavior, while `visible_loss_weight>0` adds the iBOT++ supervision on visible tokens without diluting the masked-token signal. When `mask` is `None` the loss falls back to a plain mean over all patch tokens.

Parameters

- **teacher_out** – Tensor with shape `(B, N, K)` or `(B * N, K)` containing full patch logits from the teacher model.
- **student_out** – Tensor with the same shape as `teacher_out` containing full patch logits from the student model.
- **mask** – Optional boolean tensor with shape `(B, H, W)` or `(B, N)` where `True` marks masked tokens. Used to weight masked vs. visible tokens, and required when `teacher_out` has rank 2 so that the batch size `B` can be inferred.
- **teacher_temp** – The temperature used for the teacher output. If None, the default temperature defined in `__init__` is used.
- **visible_loss_weight** – Weight applied to the visible-token (unmasked) loss term. Only used when `mask` is provided. Defaults to `1.0`. Use `0.0` to recover the original iBOT masked-only behavior.

Returns The loss value as a scalar tensor.

Raises **ValueError** – If `teacher_out` and `student_out` shapes differ, if the input rank is not 2 or 3, if rank-2 inputs are given without a mask, or if the mask shape is incompatible with the input.

class **lightly.loss.lejepa_loss.SIGReg**(**, knots: int = 17, t_max: float = 3.0, num_vectors: int = 1024, gather_distributed: bool = False*)

Sketched Isotropic Gaussian Regularization for projected embeddings.

SIGReg is the LeJEPA regularizer [0] that drives the empirical distribution of the projected embeddings toward an isotropic Gaussian. It draws random unit projection vectors (slices) and compares the empirical characteristic function of the sliced projections against the ideal Gaussian characteristic function via an Epps-Pulley integral over a frequency grid.

- [0]: LeJEPA, 2025, https://arxiv.org/abs/2511.08544
- [1]: https://github.com/galilai-group/lejepa

`num_vectors`

Number of random unit projection vectors (slices) drawn per forward pass.

`gather_distributed`

If True, statistics are aggregated across distributed ranks so the regularization uses the global batch.

`t`

Buffer of shape `(knots,)` containing the frequency grid (`linspace(0, t_max, knots)`).

`phi`

Buffer of shape `(knots,)` containing the target Gaussian characteristic function evaluated on the frequency grid (`exp(-t**2 / 2)`).

`weights`

Buffer of shape `(knots,)` containing the trapezoidal integration weights pre-multiplied by `phi` .

Examples

```
>>> # initialize the regularizer
>>> sigreg = SIGReg()
>>>
>>> # apply a transform and feed through model
>>> view = transform(images)
>>> proj = model(view)  # shape (N, D)
>>>
>>> # compute the regularization loss
>>> loss = sigreg(proj)
```

`forward`(*proj: Tensor*)→ Tensor

Compute the SIGReg loss for a batch of projections.

Parameters **proj** – Projected embeddings of shape (..., N, C).

class **lightly.loss.lejepa_loss.LeJEPALoss**(*lambda_param: float = 0.05, gather_distributed: bool = False, sigreg_knots: int = 17, sigreg_t_max: float = 3.0, sigreg_num_vectors: int = 1024*)

LeJEPA loss combining SIGReg regularization with view invariance.

The loss is a convex combination of two terms:

- `SIGReg(local_proj)` regularizes local projections toward an isotropic Gaussian distribution.
- `lejepa_invariance_loss(local_proj=local_proj, global_proj=global_proj)` pulls each local view toward the mean of the global views.

The total loss is `lambda_param * SIGReg(local_proj) + (1 - lambda_param) * invariance(local_proj, global_proj)`.

The default `lambda_param=0.05` matches the reference implementation [1]. The paper [0] explores values between 0.01 and 0.1.

- [0]: LeJEPA, 2025, https://arxiv.org/abs/2511.08544
- [1]: https://github.com/galilai-group/lejepa

lambda_param

Weight for the SIGReg term in [0, 1]. The invariance term is weighted by (1 - lambda_param).

sigreg

The SIGReg regularization module instantiated with the supplied sigreg_* parameters.

Examples

```
>>> # initialize loss function
>>> loss_fn = LeJEPALoss()
>>>
>>> # generate local and global views
>>> local_views = [transform(images) for _ in range(n_local)]
>>> global_views = [transform(images) for _ in range(n_global)]
>>>
>>> # project each view and stack to shape (V, N, D)
>>> local_proj = torch.stack([model(v) for v in local_views])
>>> global_proj = torch.stack([model(v) for v in global_views])
>>>
>>> # calculate loss
>>> loss = loss_fn(local_proj=local_proj, global_proj=global_proj)
```

forward(*, *local_proj: Tensor*, *global_proj: Tensor*) → Tensor

Compute the LeJEPA loss for a batch of multi-view projections.

Parameters
- **local_proj** – Local-view projected embeddings of shape `(Vl, N, D)`.
- **global_proj** – Global-view projected embeddings of shape `(Vg, N, D)`.

class **lightly.loss.koleo_loss.KoLeoLoss**(*p: float = 2*, *eps: float = 1e-08*)

KoLeo loss based on [0].

KoLeo loss is a regularizer that encourages a uniform span of the features in a batch by penalizing the distance between the features and their nearest neighbors.

Implementation is based on [1].

- [0]: Spreading vectors for similarity search, 2019, https://arxiv.org/abs/1806.03198
- [1]: https://github.com/facebookresearch/dinov2/blob/main/dinov2/loss/koleo_loss.py

p

The norm degree for pairwise distance calculation.

eps

Small value to avoid division by zero.

forward(*x: Tensor*) → Tensor

Forward pass through KoLeo Loss.

Parameters **x** – Tensor with shape (batch_size, embedding_size).
Returns Loss value.

class **lightly.loss.memory_bank.MemoryBankModule**(*size: Union[int, Sequence[int]] = 65536*, *gather_distributed: bool = False*, *feature_dim_first: bool = True*)

Memory bank implementation

This is a parent class to all loss functions implemented by the lightly Python package. This way, any loss can be used with a memory bank if desired.

size

Size of the memory bank as (num_features, dim) tuple. If num_features is 0 then the memory bank is disabled. Deprecated: If only a single integer is passed, it is interpreted as the number of features and the feature dimension is inferred from the first batch stored

in the memory bank. Leaving out the feature dimension might lead to errors in distributed training.

gather_distributed

If True then negatives from all gpus are gathered before the memory bank is updated. This results in more frequent updates of the memory bank and keeps the memory bank contents independent of the number of gpus. But it has the drawback that synchronization between processes is required and diversity of the memory bank content is reduced.

feature_dim_first

If True, the memory bank returns features with shape (dim, num_features). If False, the memory bank returns features with shape (num_features, dim).

Examples

```
>>> class MyLossFunction(MemoryBankModule):
>>>
>>>     def __init__(self, memory_bank_size: Tuple[int, int] = (2 ** 16, 128)):
>>>         super().__init__(memory_bank_size)
>>>
>>>     def forward(self, output: Tensor, labels: Optional[Tensor] = None):
>>>         output, negatives = super().forward(output)
>>>
>>>         if negatives is not None:
>>>             pass  # evaluate loss with negative samples
>>>         else:
>>>             pass  # evaluate loss without negative samples
```

forward(*output: Tensor*, *labels: Optional[Tensor] = None*, *update: bool = False*)→ Tuple[Tensor, Optional[Tensor]]

Query memory bank for additional negative samples

Parameters
- **output** – The output of the model.
- **labels** – Should always be None, will be ignored.
- **update** – If True, the memory bank will be updated with the current output.

Returns The output if the memory bank is of size 0, otherwise the output and the entries from the memory bank. Entries from the memory bank have shape (dim, num_features) if feature_dim_first is True and (num_features, dim) otherwise.

class **lightly.loss.mmcr_loss.MMCRLoss**(*lmda: float = 0.005*)

Implementation of the loss function from MMCR [0] using Manifold Capacity. All hyperparameters are set to the default values from the paper for ImageNet.

- **[0]: Efficient Coding of Natural Images using Maximum Manifold Capacity**

  Representations, 2023, https://arxiv.org/pdf/2303.03307.pdf

Examples

```
>>> # initialize loss function
>>> loss_fn = MMCRLoss()
>>> transform = MMCRTransform(k=2)
>>>
>>> # transform images, then feed through encoder and projector
>>> x = transform(x)
>>> online = online_network(x)
>>> momentum = momentum_network(x)
>>>
>>> # calculate loss
>>> loss = loss_fn(online, momentum)
```

forward(*online: Tensor*, *momentum: Tensor*)→ Tensor

Computes the MMCR loss for the online and momentum network outputs.

Parameters

- **online** – Output of the online network for the current batch. Expected to be of shape (batch_size, k, embedding_size), where k represents the number of randomly augmented views for each sample.
- **momentum** – Output of the momentum network for the current batch. Expected to be of shape (batch_size, k, embedding_size), where k represents the number of randomly augmented views for each sample.

Returns The computed loss value.

class **lightly.loss.msn_loss.MSNLoss**(*temperature: float = 0.1*, *sinkhorn_iterations: int = 3*, *regularization_weight: float = 1.0*, *me_max_weight: Optional[float] = None*, *gather_distributed: bool = False*)

Implementation of the loss function from MSN [0].

Code inspired by [1].

- [0]: Masked Siamese Networks, 2022, https://arxiv.org/abs/2204.07141
- [1]: https://github.com/facebookresearch/msn

temperature

Similarities between anchors and targets are scaled by the inverse of the temperature. Must be in (0, inf).

sinkhorn_iterations

Number of sinkhorn normalization iterations on the targets.

regularization_weight

Weight factor lambda by which the regularization loss is scaled. Set to 0 to disable regularization.

me_max_weight

Deprecated, use *regularization_weight* instead. Takes precedence over *regularization_weight* if not None. Weight factor lambda by which the mean entropy maximization regularization loss is scaled. Set to 0 to disable mean entropy maximization reguliarization.

gather_distributed

If True, then target probabilities are gathered from all GPUs.

Examples

```
>>> # initialize loss function
>>> loss_fn = MSNLoss()
>>>
>>> # generate anchors and targets of images
>>> anchors = transforms(images)
>>> targets = transforms(images)
>>>
>>> # feed through MSN model
>>> anchors_out = model(anchors)
>>> targets_out = model.target(targets)
>>>
>>> # calculate loss
>>> loss = loss_fn(anchors_out, targets_out, prototypes=model.prototypes)
```

forward(*anchors: Tensor*, *targets: Tensor*, *prototypes: Tensor*, *target_sharpen_temperature: float = 0.25*) → Tensor

Computes the MSN loss for a set of anchors, targets, and prototypes.

Parameters

- **anchors** – Tensor with shape (batch_size * anchor_views, dim).
- **targets** – Tensor with shape (batch_size, dim).
- **prototypes** – Tensor with shape (num_prototypes, dim).
- **target_sharpen_temperature** – Temperature used to sharpen the target probabilities.

Returns Mean loss over all anchors.

regularization_loss(*mean_anchor_probs: Tensor*)→ Tensor

Calculates mean entropy regularization loss.

Parameters **mean_anchor_probs** – The mean anchor probabilities.
Returns The calculated regularization loss.

class **lightly.loss.negative_cosine_similarity.NegativeCosineSimilarity**(*dim: int = 1, eps: float = 1e-08*)

Implementation of the Negative Cosine Simililarity used in the SimSiam[0] paper.

- [0] SimSiam, 2020, https://arxiv.org/abs/2011.10566

Examples

```
>>> # initialize loss function
>>> loss_fn = NegativeCosineSimilarity()
>>>
>>> # generate two representation tensors
>>> # with batch size 10 and dimension 128
>>> x0 = torch.randn(10, 128)
>>> x1 = torch.randn(10, 128)
>>>
>>> # calculate loss
>>> loss = loss_fn(x0, x1)
```

forward(*x0: Tensor*, *x1: Tensor*)→ Tensor

Computes the negative cosine similarity between two tensors.

Parameters
- **x0** – First input tensor.
- **x1** – Second input tensor.

Returns The mean negative cosine similarity.

class **lightly.loss.ntx_ent_loss.NTXentLoss**(*temperature: float = 0.5, memory_bank_size: Union[int, Sequence[int]] = 0, gather_distributed: bool = False*)

Implementation of the Contrastive Cross Entropy Loss.

This implementation follows the SimCLR[0] paper. If you enable the memory bank by setting the *memory_bank_size* value > 0 the loss behaves like the one described in the MoCo[1] paper.

- [0] SimCLR, 2020, https://arxiv.org/abs/2002.05709
- [1] MoCo, 2020, https://arxiv.org/abs/1911.05722

temperature

Scale logits by the inverse of the temperature.

memory_bank_size

Size of the memory bank as (num_features, dim) tuple. num_features are the number of negative samples stored in the memory bank. If num_features is 0, the memory bank is disabled. Use 0 for SimCLR. For MoCo we typically use numbers like 4096 or 65536. Deprecated: If only a single integer is passed, it is interpreted as the number of features and the feature dimension is inferred from the first batch stored in the memory bank. Leaving out the feature dimension might lead to errors in distributed training.

gather_distributed

If True then negatives from all GPUs are gathered before the loss calculation. If a memory bank is used and gather_distributed is True, then tensors from all gpus are gathered before the memory bank is updated.

Raises **ValueError** – If abs(temperature) < 1e-8 to prevent divide by zero.

Examples

```
>>> # initialize loss function without memory bank
>>> loss_fn = NTXentLoss(memory_bank_size=0)
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through SimCLR or MoCo model
>>> out0, out1 = model(t0), model(t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*out0: Tensor*, *out1: Tensor*)→ Tensor

Forward pass through Contrastive Cross-Entropy Loss.

If used with a memory bank, the samples from the memory bank are used as negative examples. Otherwise, within-batch samples are used as negative samples.

Parameters

- **out0** – Output projections of the first set of transformed images. Shape: (batch_size, embedding_size)
- **out1** – Output projections of the second set of transformed images. Shape: (batch_size, embedding_size)

Returns Contrastive Cross Entropy Loss value.

class
lightly.loss.patch_kernel_alignment_loss.PatchKernelAlignmentLoss(*max_tokens: int | None = None*)

Patch kernel alignment loss for dense self-supervised learning.

Aligns the *relational* structure of student and teacher dense features rather than asking for exact patch correspondence. For each image it builds the pairwise linear kernels of the student and teacher tokens and maximizes their Centered Kernel Alignment (CKA):

K_s = Z_s Z_s^T, K_t = Z_t Z_t^T CKA = <H K_s H, H K_t H>_F / (||H K_s H||_F * ||H K_t H||_F) L = 1 - CKA

where `H = I - (1/N) 11^T` is the centering matrix. CKA is invariant to isotropic scaling and orthogonal transforms of the features, so the loss captures region grouping and object-level organization without requiring nearest-neighbour sorting, clustering, prototypes, or geometric patch identity. This makes it well suited to detection and segmentation.

The loss is model-agnostic: it operates on dense features `(B, N, D)` where `N` is the number of ViT patch tokens (ConvNet features must be flattened to `(B, N, D)` by the caller). Teacher features are detached internally. Geometry/ROI alignment for a cross-view variant is the caller's responsibility; the loss receives already aligned dense features.

Reference:

- [0]: Patch-Level Kernel Alignment for Dense Self-Supervised Learning (PaKa), 2025, https://arxiv.org/abs/2509.05606

Parameters **max_tokens** – If not None and `N > max_tokens`, randomly subsample `max_tokens` tokens per image (preferring valid ones) before forming the kernel, to bound the `O(B N^2 D)` cost.

Example (same view, e.g. iBOT-style masking):

```
>>> criterion = PatchKernelAlignmentLoss(max_tokens=512)
>>> loss = criterion(
...     student_features=student_features,  # (B, N, D)
...     teacher_features=teacher_features,  # (B, N, D)
... )
```

Example (cross view, student and teacher see different crops):

```
>>> from lightly.loss import roi_resample_to_grid
>>> # Reshape patch tokens back to the (B, C, H, W) patch grid per view.
>>> student_map = student_tokens.transpose(1, 2).reshape(B, C, H, W)
>>> teacher_map = teacher_tokens.transpose(1, 2).reshape(B, C, H, W)
>>> # Resample the shared region of both crops onto a common 7x7 grid so
>>> # token i refers to the same location in both views.
>>> student = roi_resample_to_grid(student_map, student_boxes, 7, 7)
>>> teacher = roi_resample_to_grid(teacher_map, teacher_boxes, 7, 7)
>>> loss = PatchKernelAlignmentLoss()(student, teacher)  # (B, 49, C)
```

forward(*student_features: Tensor, teacher_features: Tensor, mask: Tensor | None = None*) → Tensor

Computes the patch kernel alignment loss.

Parameters

- **student_features** – Student dense features with shape `(B, N, D)`.
- **teacher_features** – Teacher dense features with shape `(B, N, D)`. Detached internally.
- **mask** – Optional boolean tensor `(B, N)` where `True` marks tokens to ignore. Images with fewer than two valid tokens do not contribute to the loss.

Returns The mean loss over images with enough valid tokens, as a scalar tensor. A differentiable zero is returned if no image qualifies.

Raises **ValueError** – If any input has an invalid shape or dtype.

class **lightly.loss.pmsn_loss.PMSNLoss**(*temperature: float = 0.1, sinkhorn_iterations: int = 3, regularization_weight: float = 1, power_law_exponent: float = 0.25, gather_distributed: bool = False*)

Implementation of the loss function from PMSN [0] using a power law target distribution.

- [0]: Prior Matching for Siamese Networks, 2022, https://arxiv.org/abs/2210.07277

temperature

Similarities between anchors and targets are scaled by the inverse of the temperature. Must be in (0, inf).

sinkhorn_iterations

Number of sinkhorn normalization iterations on the targets.

regularization_weight

Weight factor lambda by which the regularization loss is scaled. Set to 0 to disable regularization.

power_law_exponent

Exponent for power law distribution. Entry k of the distribution is proportional to (1 / k) ^ power_law_exponent, with k ranging from 1 to dim + 1.

gather_distributed

If True, then target probabilities are gathered from all GPUs.

Examples

```
>>> # initialize loss function
>>> loss_fn = PMSNLoss()
>>>
>>> # generate anchors and targets of images
>>> anchors = transforms(images)
>>> targets = transforms(images)
>>>
>>> # feed through PMSN model
>>> anchors_out = model(anchors)
>>> targets_out = model.target(targets)
>>>
>>> # calculate loss
>>> loss = loss_fn(anchors_out, targets_out, prototypes=model.prototypes)
```

regularization_loss(*mean_anchor_probs: Tensor*) → Tensor

Calculates the regularization loss with a power law target distribution.

Parameters	**mean_anchor_probs** – The mean anchor probabilities.
Returns	The calculated regularization loss.

class **lightly.loss.pmsn_loss.PMSNCustomLoss**(*target_distribution: Callable[[Tensor], Tensor], temperature: float = 0.1, sinkhorn_iterations: int = 3, regularization_weight: float = 1, gather_distributed: bool = False*)

Implementation of the loss function from PMSN [0] with a custom target distribution.

- [0]: Prior Matching for Siamese Networks, 2022, https://arxiv.org/abs/2210.07277

target_distribution

A function that takes the mean anchor probabilities tensor with shape (dim,) as input and returns a target probability distribution tensor with the same shape. The returned distribution should sum up to one. The final regularization loss is calculated as KL(mean_anchor_probs, target_dist) where KL is the Kullback-Leibler divergence.

temperature

Similarities between anchors and targets are scaled by the inverse of the temperature. Must be in (0, inf).

sinkhorn_iterations

Number of sinkhorn normalization iterations on the targets.

regularization_weight

Weight factor lambda by which the regularization loss is scaled. Set to 0 to disable regularization.

gather_distributed

If True, then target probabilities are gathered from all GPUs.

Examples

```
>>> # define custom target distribution
>>> def my_uniform_distribution(mean_anchor_probabilities: Tensor) -> Tensor:
>>>     dim = mean_anchor_probabilities.shape[0]
>>>     return mean_anchor_probabilities.new_ones(dim) / dim
>>>
>>> # initialize loss function
>>> loss_fn = PMSNCustomLoss(target_distribution=my_uniform_distribution)
>>>
>>> # generate anchors and targets of images
>>> anchors = transforms(images)
>>> targets = transforms(images)
>>>
>>> # feed through PMSN model
>>> anchors_out = model(anchors)
>>> targets_out = model.target(targets)
>>>
>>> # calculate loss
>>> loss = loss_fn(anchors_out, targets_out, prototypes=model.prototypes)
```

regularization_loss(*mean_anchor_probs: Tensor*) → Tensor

Calculates regularization loss with a custom target distribution.

Parameters	**mean_anchor_probs** – The mean anchor probabilities.
Returns	The calculated regularization loss.

class lightly.loss.regularizer.co2.CO2Regularizer(*alpha: float = 1, t_consistency: float = 0.05, memory_bank_size: Union[int, Sequence[int]] = 0*)

Implementation of the CO2 regularizer [0] for self-supervised learning.

- [0] CO2, 2021, https://arxiv.org/abs/2010.02217

alpha

Weight of the regularization term.

t_consistency

Temperature used during softmax calculations.

memory_bank_size

Size of the memory bank as (num_features, dim) tuple. num_features is the number of negatives stored in the bank. If set to 0, the memory bank is disabled. Deprecated: If only a single integer is passed, it is interpreted as the number of features and the feature dimension is inferred from the first batch stored in the memory bank. Leaving out the feature dimension might lead to errors in distributed training.

Examples

```
>>> # initialize loss function for MoCo
>>> loss_fn = NTXentLoss(memory_bank_size=(4096, 128))
>>>
>>> # initialize CO2 regularizer
>>> co2 = CO2Regularizer(alpha=1.0, memory_bank_size=(4096, 128))
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through the MoCo model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss and apply regularizer
>>> loss = loss_fn(out0, out1) + co2(out0, out1)
```

forward(*out0: Tensor*, *out1: Tensor*) → Tensor

Computes the CO2 regularization term for two model outputs.

Parameters
- **out0** – Output projections of the first set of transformed images.
- **out1** – Output projections of the second set of transformed images.

Returns The regularization term multiplied by the weight factor alpha.

class **lightly.loss.regularizer.dse.DSERegularizer**(*weight: float = 1.0*, *normalize: bool = True*, *local_clusters: int = 3*, *global_clusters: int = 24*, *max_kmeans_iters: int = 20*, *tol: float = 0.0001*)

Dense Structure Estimator regularizer for dense representations.

This experimental regularizer is based on the training regularizer variant from “Exploring Structural Degradation in Dense Representations for Self-supervised Learning”, 2025, https://arxiv.org/abs/2510.17299.

The regularizer expects dense representations of shape `(B, N, D)`, where `B` is the batch size, `N` is the number of dense locations such as patches or spatial positions, and `D` is the feature dimension. Higher DSE values indicate better dense structure, so this module returns

`-weight * dse` and can be added to an existing SSL loss.

Examples

```
>>> regularizer = DSERegularizer(weight=0.1)
>>> ssl_loss = torch.tensor(1.0)
>>>
>>> # Transformer output: patch_tokens has shape (B, N, D).
>>> patch_tokens = torch.randn(8, 196, 384)
>>> loss = ssl_loss + regularizer(patch_tokens)
>>>
>>> # Convolutional output: features has shape (B, C, H, W).
>>> features = torch.randn(8, 384, 14, 14)
>>> dense_features = features.flatten(2).transpose(1, 2)
>>> loss = ssl_loss + regularizer(dense_features)
```

weight

Weight of the regularization term.

normalize

If True, L2-normalize representations along the feature dimension.

local_clusters

Number of clusters for patches within each image.

global_clusters

Number of clusters for patches across image groups.

max_kmeans_iters

Maximum number of k-means iterations.

tol

Early stopping threshold for centroid shift in k-means.

forward(*representations: Tensor*) → Tensor

Computes the DSE regularization loss.

Parameters	**representations** – Dense representations of shape `(B, N, D)`.
Returns	Scalar DSE regularization loss.

class **lightly.loss.swav_loss.SwaVLoss**(*temperature: float = 0.1, sinkhorn_iterations: int = 3, sinkhorn_epsilon: float = 0.05, sinkhorn_gather_distributed: bool = False*)

Implementation of the SwaV loss.

temperature

Temperature parameter used for cross entropy calculations.

sinkhorn_iterations

Number of iterations of the sinkhorn algorithm.

sinkhorn_epsilon

Temperature parameter used in the sinkhorn algorithm.

sinkhorn_gather_distributed

If True, features from all GPUs are gathered to calculate the soft codes in the sinkhorn algorithm.

forward(*high_resolution_outputs: List[Tensor], low_resolution_outputs: List[Tensor], queue_outputs: Optional[List[Tensor]] = None*)→ Tensor

Computes the SwaV loss for a set of high and low resolution outputs.

- [0]: SwaV, 2020, https://arxiv.org/abs/2006.09882

Parameters
- **high_resolution_outputs** – List of similarities of features and SwaV prototypes for the high resolution crops.
- **low_resolution_outputs** – List of similarities of features and SwaV prototypes for the low resolution crops.
- **queue_outputs** – List of similarities of features and SwaV prototypes for the queue of high resolution crops from previous batches.

Returns Swapping assignments between views loss (SwaV) as described in [0].

subloss(*z: Tensor, q: Tensor*)→ Tensor

Calculates the cross entropy for the SwaV prediction problem.

Parameters
- **z** – Similarity of the features and the SwaV prototypes.
- **q** – Codes obtained from Sinkhorn iterations.

Returns Cross entropy between predictions z and codes q.

class **lightly.loss.sym_neg_cos_sim_loss.SymNegCosineSimilarityLoss**

Implementation of the Symmetrized Loss used in the SimSiam[0] paper.

- [0] SimSiam, 2020, https://arxiv.org/abs/2011.10566

Examples

```
>>> # initialize loss function
>>> loss_fn = SymNegCosineSimilarityLoss()
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through SimSiam model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*out0: Tensor, out1: Tensor*)→ Tensor

Forward pass through Symmetric Loss.

Parameters

- **out0** – Output projections of the first set of transformed images. Expects the tuple to be of the form (z0, p0), where z0 is the output of the backbone and projection MLP, and p0 is the output of the prediction head.
- **out1** – Output projections of the second set of transformed images. Expects the tuple to be of the form (z1, p1), where z1 is the output of the backbone and projection MLP, and p1 is the output of the prediction head.

Returns Negative Cosine Similarity loss value.

class **lightly.loss.tico_loss.TiCoLoss**(*beta: float = 0.9, rho: float = 8.0, gather_distributed: bool = False*)

Implementation of the Tico Loss from Tico[0] paper.

This implementation takes inspiration from the code published by sayannag using Lightly. [1]

- [0] Jiachen Zhu et. al, 2022, Tico... https://arxiv.org/abs/2206.10698
- [1] https://github.com/sayannag/TiCo-pytorch

Args

beta:

Coefficient for the EMA update of the covariance Defaults to 0.9 [0].

rho:

Weight for the covariance term of the loss Defaults to 8.0 [0].

gather_distributed:

If True, the cross-correlation matrices from all GPUs are gathered and summed before the loss calculation.

Examples

```
>>> # initialize loss function
>>> loss_fn = TiCoLoss()
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*z_a: Tensor*, *z_b: Tensor*, *update_covariance_matrix: bool = True*) → Tensor

Computes the TiCo loss.

It maximizes the agreement among embeddings of different distorted versions of the same image while avoiding collapse using Covariance matrix.

Parameters
- **z_a** – Tensor of shape [batch_size, num_features=256]. Output of the learned backbone.
- **z_b** – Tensor of shape [batch_size, num_features=256]. Output of the momentum updated backbone.
- **update_covariance_matrix** – Parameter to update the covariance matrix at each iteration.

Returns The computed loss.

Raises
- **AssertionError** – If z_a or z_b have a batch size <= 1.
- **AssertionError** – If z_a and z_b do not have the same shape.

class **lightly.loss.vicreg_loss.VICRegLoss**(*lambda_param: float = 25.0*, *mu_param: float = 25.0*, *nu_param: float = 1.0*, *gather_distributed: bool = False*, *eps: float = 0.0001*)

Implementation of the VICReg loss [0].

This implementation is based on the code published by the authors [1].

- [0] VICReg, 2022, https://arxiv.org/abs/2105.04906
- [1] https://github.com/facebookresearch/vicreg/

lambda_param

Scaling coefficient for the invariance term of the loss.

mu_param

Scaling coefficient for the variance term of the loss.

nu_param

Scaling coefficient for the covariance term of the loss.

gather_distributed

If True, the cross-correlation matrices from all GPUs are gathered and summed before the loss calculation.

eps

Epsilon for numerical stability.

Examples

```
>>> # initialize loss function
>>> loss_fn = VICRegLoss()
>>>
>>> # generate two random transforms of images
>>> t0 = transforms(images)
>>> t1 = transforms(images)
>>>
>>> # feed through model
>>> out0, out1 = model(t0, t1)
>>>
>>> # calculate loss
>>> loss = loss_fn(out0, out1)
```

forward(*z_a: Tensor*, *z_b: Tensor*) → Tensor

Returns VICReg loss.

Parameters	• **z_a** – Tensor with shape (batch_size, ..., dim). • **z_b** – Tensor with shape (batch_size, ..., dim).
Returns	The computed VICReg loss.
Raises	• **AssertionError** – If z_a or z_b have a batch size <= 1. • **AssertionError** – If z_a and z_b do not have the same shape.

class **lightly.loss.vicregl_loss.VICRegLLoss**(*lambda_param: float = 25.0*, *mu_param: float = 25.0*, *nu_param: float = 1.0*, *alpha: float = 0.75*, *gather_distributed: bool = False*, *eps: float = 0.0001*, *num_matches: Tuple[int, int] = (20, 4)*)

Implementation of the VICRegL loss from VICRegL paper [0].

This implementation follows the code published by the authors [1].

- [0]: VICRegL, 2022, https://arxiv.org/abs/2210.01571
- [1]: https://github.com/facebookresearch/VICRegL

`lambda_param`

Coefficient for the invariance term of the loss.

`mu_param`

Coefficient for the variance term of the loss.

`nu_param`

Coefficient for the covariance term of the loss.

`alpha`

Coefficient to weight global with local loss. The final loss is computed as (self.alpha * global_loss + (1-self.alpha) * local_loss).

`gather_distributed`

If True, the cross-correlation matrices from all gpus are gathered and summed before the loss calculation.

`eps`

Epsilon for numerical stability.

`num_matches`

Number of local features to match using nearest neighbors.

Examples

```
>>> # initialize loss function
>>> criterion = VICRegLLoss()
>>> transform = VICRegLTransform(n_global_views=2, n_local_views=4)
>>>
>>> # generate two random transforms of images
>>> views_and_grids = transform(images)
>>> views = views_and_grids[:6]  # 2 global views + 4 local views
>>> grids = views_and_grids[6:]
>>>
>>> # feed through model images
>>> features = [model(view) for view in views]
>>>
>>> # calculate loss
>>> loss = criterion(
...     global_view_features=features[:2],
...     global_view_grids=grids[:2],
...     local_view_features=features[2:],
...     local_view_grids=grids[2:],
... )
```

forward(*global_view_features: Sequence[Tuple[Tensor, Tensor]], global_view_grids: Sequence[Tensor], local_view_features: Optional[Sequence[Tuple[Tensor, Tensor]]] = None, local_view_grids: Optional[Sequence[Tensor]] = None*)→ Tensor

Computes the global and local VICRegL loss from the input features.

Parameters

- **global_view_features** – Sequence of (global_features, local_features) tuples from the global crop views. global_features must have size (batch_size, global_feature_dim) and local_features must have size (batch_size, grid_height, grid_width, local_feature_dim).
- **global_view_grids** – Sequence of grid tensors from the global crop views. Every tensor must have shape (batch_size, grid_height, grid_width, 2).
- **local_view_features** – Sequence of (global_features, local_features) tuples from the local crop views. global_features must have size (batch_size, global_feature_dim) and local_features must have size (batch_size, grid_height, grid_width, local_feature_dim). Note that grid_height and grid_width can differ between global_view_features and local_view_features.
- **local_view_grids** – Sequence of grid tensors from the local crop views. Every tensor must have shape (batch_size, grid_height, grid_width, 2). Note that grid_height and grid_width can differ between global_view_features and local_view_features.

Returns *Weighted sum of the global and local loss, calculated as – self.alpha * global_loss + (1-self.alpha) * local_loss.*

Raises

- **ValueError** – If the lengths of global_view_features and global_view_grids are not the same.

- **ValueError** – If the lengths of local_view_features and local_view_grids are not the same.
- **ValueError** – If only one of local_view_features or local_view_grids is set.

class **lightly.loss.visreg_loss.VISRegLoss**(*lambda_param: float = 0.9, num_slices: int = 4096, lambda_scale: float = 1.0, lambda_shape: float = 1.0, lambda_center: float = 1.0, gather_distributed: bool = False, eps: float = 0.0001*)

Implementation of the VISReg loss [0].

VISReg regularizes the projected embeddings with three decoupled terms: a center loss on the batch mean, a scale loss on the per-dimension standard deviation, and a sliced-Wasserstein shape loss that sketches the embedding distribution toward an isotropic Gaussian. The regularization is combined with the LeJEPA multi-view invariance loss:

`total = (1 - lambda_param) * pred + lambda_param * reg` where `reg = lambda_scale * scale + lambda_shape * shape + lambda_center * center`.

The implementation follows Algorithm 1 of the paper. The default `lambda_param=0.9` and equal component weights are the best settings reported for large datasets such as ImageNet-1K (Tables 3 and 12 in [0]); the paper recommends `lambda_param=0.6` for small datasets and increasing `lambda_shape` for long-tailed or low-rank data.

In distributed training every device draws its own random slices and computes the loss on its local batch, so no communication is required and the effective number of slices grows with the world size (Section 3.2 in [0]). This is the paper's setting and the default. Note that this requires the random number generators of the devices to be seeded differently; identically seeded devices draw identical slices. Set `gather_distributed=True` to additionally gather the embeddings from all devices before computing the regularization statistics, which can help when the per-device batch is very small.

- [0]: VISReg, Wu et al., 2026, https://arxiv.org/abs/2606.02572
- [1]: https://haiyuwu.github.io/visreg

lambda_param

Weight for the regularization term in [0, 1]. The invariance term is weighted by (1 - lambda_param).

num_slices

Number of random 1D projection directions (slices) drawn per forward pass on each device.

lambda_scale

Weight for the scale loss inside the regularization term.

lambda_shape

Weight for the shape loss inside the regularization term.

lambda_center

Weight for the center loss inside the regularization term.

gather_distributed

If True, the embeddings from all devices are gathered before computing the regularization statistics.

eps

Numerical stability term for the shape loss normalization.

Examples

```
>>> # initialize loss function
>>> loss_fn = VISRegLoss()
>>>
>>> # generate local and global views
>>> local_views = [transform(images) for _ in range(n_local)]
>>> global_views = [transform(images) for _ in range(n_global)]
>>>
>>> # project each view and stack to shape (V, N, D)
>>> local_proj = torch.stack([model(v) for v in local_views])
>>> global_proj = torch.stack([model(v) for v in global_views])
>>>
>>> # calculate loss
>>> loss = loss_fn(local_proj=local_proj, global_proj=global_proj)
```

__init__(*lambda_param: float = 0.9, num_slices: int = 4096, lambda_scale: float = 1.0, lambda_shape: float = 1.0, lambda_center: float = 1.0, gather_distributed: bool = False, eps: float = 0.0001*)

Initializes the VISRegLoss module with the specified parameters.

Parameters

- **lambda_param** – Weight for the regularization term in `[0, 1]`. The invariance term is weighted by `1 - lambda_param`.
- **num_slices** – Number of random 1D projection directions (slices) drawn per forward pass on each device.
- **lambda_scale** – Weight for the scale loss inside the regularization term.
- **lambda_shape** – Weight for the shape loss inside the regularization term.

- **lambda_center** – Weight for the center loss inside the regularization term.
- **gather_distributed** – If True, the embeddings from all devices are gathered before computing the regularization statistics.
- **eps** – Numerical stability term for the shape loss normalization.

Raises **ValueError** – If any parameter is outside its valid range or if gather_distributed is True but torch.distributed is not available.

`forward`(*, *local_proj: Tensor*, *global_proj: Tensor*) → Tensor

Computes the VISReg loss for a batch of multi-view projections.

Parameters

- **local_proj** – Local-view projected embeddings of shape `(Vl, N, D)` where `Vl` is the number of local views, `N` is the batch size, and `D` is the projection dimensionality.
- **global_proj** – Global-view projected embeddings of shape `(Vg, N, D)` where `Vg` is the number of global views.

Returns The combined VISReg loss.

`forward_components`(*, *local_proj: Tensor*, *global_proj: Tensor*) → VISRegLossComponents

Computes the VISReg loss and returns all components separately.

The prediction term pulls every view, global and local, toward the centroid of the global views (Eq. 8 in the paper). The regularization terms are computed over all views with each view's batch treated as an independent sample set.

Parameters

- **local_proj** – Local-view projected embeddings of shape `(Vl, N, D)` where `Vl` is the number of local views, `N` is the batch size, and `D` is the projection dimensionality.
- **global_proj** – Global-view projected embeddings of shape `(Vg, N, D)` where `Vg` is the number of global views.

Returns A VISRegLossComponents tuple with the total loss and the individual pred, scale, shape, and center losses.

Raises **ValueError** – If the projections have invalid shapes or a batch size smaller than two.