Robustness_Code_W&B_Sweep

October 5, 2021

[0]:
```
%%capture
!pip install wandb pytorch-lightning wandb
```

[0]:
```
import wandb
wandb.login()
```

[0]:
```
<IPython.core.display.Javascript object>

wandb: Appending key for api.wandb.ai to your netrc file:
/root/.netrc
```

[0]:
```
True
```

[0]:
```
%%writefile train.py
import sys

import wandb
import torch
import torchvision

import numpy as np
import torch.nn.functional as F

from math import log
from torch import Tensor
from typing import Optional, List

from torch import nn
from torch.utils.data.dataloader import DataLoader
from torch.utils.data import random_split

from torchvision import transforms
from torchvision.datasets import CIFAR10, CIFAR100, MNIST, FashionMNIST
from torchvision.models import resnet34

from numpy.testing import assert_array_almost_equal

import pytorch_lightning as pl
```

```
from pytorch_lightning.loggers import CSVLogger, WandbLogger

from torchmetrics import Accuracy

csv_logger = CSVLogger("drive/MyDrive/moments-penalization/logs",␣
↪name="cifar100")


#### DATASET WITH NOISE ####

class MNISTNOISY(FashionMNIST):
  def __init__(self, root, train=True,  transform=None,      ␣
↪target_transform=None,             download=False,   nosiy_rate=0.0,␣
↪asym=False):
    super().__init__(root, train=train, transform=transform,␣
↪target_transform=target_transform, download=download,)

    # Save targets before we shuffle
    self.original_targets = self.targets.detach().clone()

    if asym:
      # automobile < - truck, bird -> airplane, cat <-> dog, deer -> horse
      source_class = [9, 2, 3, 5, 4]
      target_class = [1, 0, 5, 3, 7]
      for s, t in zip(source_class, target_class):
        cls_idx = np.where(np.array(self.targets) == s)[0]
        n_noisy = int(nosiy_rate * cls_idx.shape[0])
        noisy_sample_index = np.random.choice(cls_idx, n_noisy, replace=False)
        for idx in noisy_sample_index:
            self.targets[idx] = t


    elif nosiy_rate > 0:

      n_samples = len(self.targets)
      n_noisy = int(nosiy_rate * n_samples)
      print("%d Noisy samples" % (n_noisy))
      class_index = [np.where(np.array(self.targets) == i)[0] for i in␣
↪range(10)]
      class_noisy = int(n_noisy / 10)
      noisy_idx = []
      for d in range(10):
        noisy_class_index = np.random.choice(class_index[d], class_noisy,␣
↪replace=False)
        noisy_idx.extend(noisy_class_index)
        print("Class %d, number of noisy % d" % (d, len(noisy_class_index)))
      for i in noisy_idx:
```

```
        self.targets[i] = self._other_class(n_classes=10, current_class=self.
↪targets[i])
      print(len(noisy_idx))
      print("Print noisy label generation statistics:")
      for i in range(10):
        n_noisy = np.sum(np.array(self.targets) == i)
        print("Noisy class %s, has %s samples." % (i, n_noisy))


  def __getitem__(self, index: int):
    item = super().__getitem__(index)

    original_target = self.original_targets[index]

    return item[0], item[1], original_target, item[1] != original_target


  def other_class(self, n_classes, current_class):
    """
    Returns a list of class indices excluding the class indexed by class_ind
    :param nb_classes: number of classes in the task
    :param class_ind: the class index to be omitted
    :return: one random class that != class_ind
    """
    if current_class < 0 or current_class >= n_classes:
      error_str = "class_ind must be within the range (0, nb_classes - 1)"
      raise ValueError(error_str)

    other_class_list = list(range(n_classes))
    other_class_list.remove(current_class)
    other_class = np.random.choice(other_class_list)

    return other_class



class CIFAR10NOISY(CIFAR10):
  def __init__(self, root, train=True,  transform=None,     ␣
↪target_transform=None,            download=False,   nosiy_rate=0.0,␣
↪asym=False):
    super().__init__(root, train=train, transform=transform,␣
↪target_transform=target_transform, download=download,)

    # Save targets before we shuffle
    self.original_targets = self.targets.copy()

    if asym:
```

```
      # automobile <- truck, bird -> airplane, cat <-> dog, deer -> horse
      source_class = [9, 2, 3, 5, 4]
      target_class = [1, 0, 5, 3, 7]
      for s, t in zip(source_class, target_class):
        cls_idx = np.where(np.array(self.targets) == s)[0]
        n_noisy = int(nosiy_rate * cls_idx.shape[0])
        noisy_sample_index = np.random.choice(cls_idx, n_noisy, replace=False)
        for idx in noisy_sample_index:
            self.targets[idx] = t


    elif nosiy_rate > 0:

      n_samples = len(self.targets)
      n_noisy = int(nosiy_rate * n_samples)
      print("%d Noisy samples" % (n_noisy))
      class_index = [np.where(np.array(self.targets) == i)[0] for i in␣
↪range(10)]
      class_noisy = int(n_noisy / 10)
      noisy_idx = []
      for d in range(10):
        noisy_class_index = np.random.choice(class_index[d], class_noisy,␣
↪replace=False)
        noisy_idx.extend(noisy_class_index)
        print("Class %d, number of noisy % d" % (d, len(noisy_class_index)))
      for i in noisy_idx:
        self.targets[i] = self. other_class(n_classes=10, current_class=self.
↪targets[i])
      print(len(noisy_idx))
      print("Print noisy label generation statistics:")
      for i in range(10):
        n_noisy = np.sum(np.array(self.targets) == i)
        print("Noisy class %s, has %s samples." % (i, n_noisy))


  def __getitem__(self, index: int):
    item = super().__getitem__(index)

    original_target = self.original_targets[index]

    return item[0], item[1], original_target, item[1] != original_target


  def other_class(self, n_classes, current_class):
    """
    Returns a list of class indices excluding the class indexed by class_ind
    :param nb_classes: number of classes in the task
```

```
    :param class_ind: the class index to be omitted
    :return: one random class that != class_ind
    """
    if current_class < 0 or current_class >= n_classes:
      error_str = "class_ind must be within the range (0, nb_classes - 1)"
      raise ValueError(error_str)

    other_class_list = list(range(n_classes))
    other_class_list.remove(current_class)
    other_class = np.random.choice(other_class_list)

    return other_class


class CIFAR100NOISY(CIFAR100):
    def __init__(self, root, train=True, transform=None, target_transform=None,␣
↪download=False, nosiy_rate=0.0, asym=False, seed=0):

      super().__init__(root, train=train, download=download,␣
↪transform=transform, target_transform=target_transform)

      # Save targets before we shuffle
      self.original_targets = self.targets.copy()

      if asym:
          """mistakes are inside the same superclass of 10 classes, e.g. 'fish'
          """
          nb_classes = 100
          P = np.eye(nb_classes)
          n = nosiy_rate
          nb_superclasses = 20
          nb_subclasses = 5

          if n > 0.0:
              for i in np.arange(nb_superclasses):
                  init, end = i * nb_subclasses, (i+1) * nb_subclasses
                  P[init:end, init:end] = self.
↪build_for_cifar100(nb_subclasses, n)

                  y_train_noisy = self.multiclass_noisify(np.array(self.
↪targets), P=P, random_state=seed)
                  actual_noise = (y_train_noisy != np.array(self.targets)).
↪mean()
              assert actual_noise > 0.0
              print('Actual noise %.2f' % actual_noise)
              self.targets = y_train_noisy.tolist()
          return
```

```
    elif nosiy_rate > 0:
        n_samples = len(self.targets)
        n_noisy = int(nosiy_rate * n_samples)
        print("%d Noisy samples" % (n_noisy))
        class_index = [np.where(np.array(self.targets) == i)[0] for i in␣
↪range(100)]
        class_noisy = int(n_noisy / 100)
        noisy_idx = []
        for d in range(100):
            noisy_class_index = np.random.choice(class_index[d], class_noisy,␣
↪replace=False)
            noisy_idx.extend(noisy_class_index)
            print("Class %d, number of noisy % d" % (d,␣
↪len(noisy_class_index)))
        for i in noisy_idx:
            self.targets[i] = self.other_class(n_classes=100,␣
↪current_class=self.targets[i])
        print(len(noisy_idx))
        print("Print noisy label generation statistics:")
        for i in range(100):
            n_noisy = np.sum(np.array(self.targets) == i)
            print("Noisy class %s, has %s samples." % (i, n_noisy))
        return


  def __getitem__(self, index: int):
    item = super().__getitem__(index)

    original_target = self.original_targets[index]

    return item[0], item[1], original_target, item[1] != original_target


  def build_for_cifar100(self, size, noise):
      """ random flip between two random classes.
      """
      assert(noise >= 0.) and (noise <= 1.)

      P = (1. - noise) * np.eye(size)
      for i in np.arange(size - 1):
          P[i, i+1] = noise

      # adjust last row
      P[size-1, 0] = noise

      assert_array_almost_equal(P.sum(axis=1), 1, 1)
      return P
```

```
    def multiclass_noisify(self, y, P, random_state=0):
        """ Flip classes according to transition probability matrix T.
        It expects a number between 0 and the number of classes - 1.
        """

        assert P.shape[0] == P.shape[1]
        assert np.max(y) < P.shape[0]

        # row stochastic matrix
        assert_array_almost_equal(P.sum(axis=1), np.ones(P.shape[1]))
        assert (P >= 0.0).all()

        m = y.shape[0]
        new_y = y.copy()
        flipper = np.random.RandomState(random_state)

        for idx in np.arange(m):
            i = y[idx]
            # draw a vector with only an 1
            flipped = flipper.multinomial(1, P[i, :], 1)[0]
            new_y[idx] = np.where(flipped == 1)[0]

        return new_y


    def other_class(self, n_classes, current_class):
        """
        Returns a list of class indices excluding the class indexed by class_ind
        :param nb_classes: number of classes in the task
        :param class_ind: the class index to be omitted
        :return: one random class that != class_ind
        """
        if current_class < 0 or current_class >= n_classes:
            error_str = "class_ind must be within the range (0, nb_classes - 1)"
            raise ValueError(error_str)

        other_class_list = list(range(n_classes))
        other_class_list.remove(current_class)
        other_class = np.random.choice(other_class_list)
        return other_class


#### LOSSES ####
```

```
def generalized_cross_entropy(input: Tensor, target: Tensor, q:␣
↪Optional[float]=0.7,
                              k: Optional[float]=None, weights:␣
↪Optional[Tensor]=None,
                              version:Optional[int]=1, num_classes:
↪Optional[int]=10):
  """
  Interpolates between a linear function (q = 1) and log function (q = 0).

  Loss function introduced in "Generalized Cross Entropy Loss for Training Deep
  Neural Networks with Noisy Labels" by Zhilu Zhang and Mert R. Sabuncu

  Args:
    input: tensor of size (N, C) where N is number of samples and C is the
        number of classes. The values of the matrix should not be normalized to
        represent probabilities, this will be done by the function.
    target: tensor of size (N) where each value is between 0 and C-1.
    q: Interpolation parameter that selects the shape of the function. Values
        close to 1 increase robustness but decrease convergence and a value
        close to 0 produces a different behavior. Default: 0.7 as in the paper.
    k: Truncation treshold, larger values lead to tighter bounds and hence more
        noise-robustness, too large of a threshold would precipitate too many
        discarded samples for training. Optimal value depends on the noise level
        in the labels. The value used in the paper is 0.5.
    weights (Tensor, optional): Selects which samples to use for training. If k
        is not none but no weights are given then they will be computed by
        thresholding the probabilities of input with k.
    version: Selects the implementation version of the method. Version 1 uses
        gather approach, version 2 uses 1 hot encoding approach. Default: 1
    num_classes: Number of classes in the target vector. Required for version 2.

  Example::

    >>> # input is of size N x C = 3 x 5
    >>> input = torch.randn(3, 5, requires_grad=True)
    >>> # each element in target has to have 0 <= value < C
    >>> target = torch.tensor([1, 0, 4])
    >>> output = generalized_cross_entropy(input, target)
    >>> output.backward()

  """

  prob   = input.softmax(dim=1)

  if version == 1:
    f_j  = prob.gather(dim=1, index=target.unsqueeze(dim=1)).flatten()
  elif version == 2:
```

```
    f_j  = (prob * torch.nn.functional.one_hot(target, num_classes)).sum(dim=1)

  losses = (1 - f_j ** q) / q

  # Should we truncate?
  if k is not None:

    # Compute weights
    if weights is None:
      weights = 1.0 * (f_j > k)

    # Use eq (12)
    losses = weights * losses + (1 - weights) * (1 - k ** q) / q


  return losses.mean()


def term_transformation(losses: Tensor, t: Optional[int]=-2, version:␣
↪Optional[int]=1):
  """
  Computes the tilted loss with the tilt value of t.

  Tilted minimization problem introduced in "Tilted Empirical Risk Minimization"
  by Tian Li, Ahmad Beirami, Maziar Sanjabi, Virginia Smith

  Args:
    losses: tensor of size (N) where N is number of samples and the entries are
        value of the loss function for each sample.
    t: Interpolation parameter that selects what objective to minimize. Positive
        values tilt towards max loss recoverd when t approaches +inf, negative
        values tilt towards min loss recoverd when t approaches -inf, and the
        standard mean value optimization for t close to 0. Default: -2 as in the
        paper.
    version: Selects the implementation version of the method. Version 1 uses
        logsumexp method to preserve precision, version 2 is the naive
        implementation. Default: 1

  Example::

    >>> # input is of size N x C = 3 x 5
    >>> input = torch.randn(3, 5, requires_grad=True)
    >>> # each element in target has to have 0 <= value < C
    >>> target = torch.tensor([1, 0, 4])
    >>> losses = torch.nn.functional.cross_entropy(input, target,␣
↪reduction="none")
    >>> output = term_transformation(losses, t=-2)
```

```
    >>> output.backward()

    """

    if version == 1:
        loss = (t * losses - log(losses.shape[0])).logsumexp(dim=0) / t
    elif version == 2:
        loss = (t * losses).exp().mean().log() / t

    return loss


def taylor_cross_entropy(input: Tensor, target: Tensor, t: Optional[int]=2,
                         version:Optional[int]=1, num_classes:Optional[int]=10):
    """
    Approximates the cross entropy using the Taylor expansion up to a term t.

    Method introduced in "Can Cross Entropy Loss Be Robust to Label Noise?" by
    Lei Feng, Senlin Shu, Zhuoyi Lin, Fengmao Lv, Li Li, Bo An

    Args:
        input: tensor of size (N, C) where N is number of samples and C is the
            number of classes. The values of the matrix should not be normalized to
            represent probabilities, this will be done by the function.
        target: tensor of size (N) where each value is between 0 and C-1.
        t: Taylor series order to be used for the approximation. Default: 2 as in␣
↪the
            paper.
        version: Selects the implementation version of the method. Version 1 uses
            gather approach, version 2 uses 1 hot encoding approach. Default: 1
        num_classes: Number of classes in the target vector. Required for version 2.

    Example::

        >>> # input is of size N x C = 3 x 5
        >>> input = torch.randn(3, 5, requires_grad=True)
        >>> # each element in target has to have 0 <= value < C
        >>> target = torch.tensor([1, 0, 4])
        >>> output = taylor_cross_entropy(input, target)
        >>> output.backward()

    """

    prob   = input.softmax(dim=1)
    if version == 1:
        f_y  = prob.gather(dim=1, index=target.unsqueeze(dim=1)).flatten()
    elif version == 2:
```

```
    f_y  = (prob * torch.nn.functional.one_hot(target, num_classes)).sum(dim=1)

  losses = 0

  for k in range(1, t+1):
    losses += (1 - f_y)**t / t

  return losses.mean()


def normalized_cross_entropy(input: Tensor, target: Tensor, version:
 ↪Optional[int]=1,
                             num_classes:Optional[int]=10):
  """
  Normalized crosss entropy, symmetric loss.

  Method introduced in "Normalized Loss Functions for Deep Learning with Noisy
  Labels" by Xingjun Ma, Hanxun Huang, Yisen Wang, Simone Romano, Sarah Erfani,
  James Bailey

  Args:
    input: tensor of size (N, C) where N is number of samples and C is the
        number of classes. The values of the matrix should not be normalized to
        represent probabilities, this will be done by the function.
    target: tensor of size (N) where each value is between 0 and C-1.
    version: Selects the implementation version of the method. Version 1 uses
        gather approach, version 2 uses 1 hot encoding approach. Default: 1
    num_classes: Number of classes in the target vector. Required for version 2.

  Example::

    >>> # input is of size N x C = 3 x 5
    >>> input = torch.randn(3, 5, requires_grad=True)
    >>> # each element in target has to have 0 <= value < C
    >>> target = torch.tensor([1, 0, 4])
    >>> output = normalized_cross_entropy(input, target)
    >>> output.backward()

  """
  log_prob = torch.nn.functional.log_softmax(input, dim=1)
  if version == 1:
    log_fy = log_prob.gather(dim=1, index=target.unsqueeze(dim=1)).flatten()
  elif version == 2:
    log_fy = (log_prob * torch.nn.functional.one_hot(target, num_classes)).
 ↪sum(dim=1)

  return (log_fy / log_prob.sum(dim=1)).mean()
```

```
def reverse_cross_entropy(input: Tensor, target: Tensor, num_classes:
↪Optional[int]=10):
  """
  Reverse crosss entropy, used to make cross entropy a symmetric loss.

  Method introduced in "Symmetric Cross Entropy for Robust Learning with Noisy
  Labels" by Yisen Wang, Xingjun Ma, Zaiyi Chen, Yuan Luo, Jinfeng Yi, James␣
↪Bailey

  Args:
    input: tensor of size (N, C) where N is number of samples and C is the
        number of classes. The values of the matrix should not be normalized to
        represent probabilities, this will be done by the function.
    target: tensor of size (N) where each value is between 0 and C-1.
    num_classes: Number of classes in the target vector.

  Example::

    >>> # input is of size N x C = 3 x 5
    >>> input = torch.randn(3, 5, requires_grad=True)
    >>> # each element in target has to have 0 <= value < C
    >>> target = torch.tensor([1, 0, 4])
    >>> output = reverse_cross_entropy(input, target)
    >>> output.backward()

  """
  prob     = input.softmax(dim=1).clamp(min=1e-7)
  log_1hot = torch.nn.functional.one_hot(target, num_classes).float().
↪clamp(min=1e-4).log()

  return -(prob * log_1hot).sum(dim=1).mean()


def moments_penalization(losses: Tensor, lambdas:List[float]=[1.0,],
                         convex: Optional[bool]=True, z_mean:␣
↪Optional[Tensor]=None,
                         normalize: Optional[bool]=True):
  """
  Reverse crosss entropy, used to make cross entropy a symmetric loss.

  Introduced in current paper.

  Args:
    losses: tensor of size (N) where N is number of samples and the entries are
        value of the loss function for each sample.
```

```
    lambdas: list of floating numper representing the penalization factor for
        the moments of the loss. Examples, lamdas = [1.0, -0.5] is equivalent to
        1.0*mean(losses) -0.5*var(losses). Default: [1.0,] equivalent to
        optimizing the mean.
    convex: make negative weights be 0 to preserve the convexity of the loss
        function at the cost of possible reduced penalization impact. Default:␣
↪True
    z_mean: mean vector to use to calculate weights.

  Example::

    >>> # input is of size N x C = 3 x 5
    >>> input = torch.randn(3, 5, requires_grad=True)
    >>> # each element in target has to have 0 <= value < C
    >>> target = torch.tensor([1, 0, 4])
    >>> losses = torch.nn.functional.cross_entropy(input, target,␣
↪reduction="none")
    >>> output = moments_penalization(losses, [1.0, -0.5])
    >>> output.backward()

  """
  z      = losses.detach()

  if z_mean is None:
    z_mean = z.mean()

  weights = torch.zeros_like(z)
  for k, lmbda in enumerate(lambdas):
    if lmbda:
      if k == 0:
        weights += lmbda
      else:
        weights += lmbda * (z - z_mean) ** k

  if convex:
    weights = weights.clip(min=0.0)

  if normalize:
    weights = weights / weights.mean()

  return (weights * losses).mean()


def moments_penalization_cross_entropy(losses: Tensor, lambda_1=1.0, lambda_2=0.
↪0,
                                       convex: Optional[bool]=True):
  """
```

```
 Reverse crosss entropy, used to make cross entropy a symmetric loss.

 Introduced in current paper.

 Args:
   losses: tensor of size (N) where N is number of samples and the entries are
       value of the loss function for each sample.
   lambdas: list of floating numper representing the penalization factor for
       the moments of the loss. Examples, lamdas = [1.0, -0.5] is equivalent to
       1.0*mean(losses) -0.5*var(losses). Default: [1.0,] equivalent to
       optimizing the mean.
   convex: make negative weights be 0 to preserve the convexity of the loss
       function at the cost of possible reduced penalization impact. Default:␣
↪True
   z_mean: mean vector to use to calculate weights.

 Example::

   >>> # input is of size N x C = 3 x 5
   >>> input = torch.randn(3, 5, requires_grad=True)
   >>> # each element in target has to have 0 <= value < C
   >>> target = torch.tensor([1, 0, 4])
   >>> losses = torch.nn.functional.cross_entropy(input, target,␣
↪reduction="none")
   >>> output = moments_penalization(losses, [1.0, -0.5])
   >>> output.backward()

 """
 z        = losses.detach()
 z_mean   = z.mean()
 x_values = (-z).exp()
 weights  = lambda_1 + lambda_2 * (z - z_mean)

 # Critical point (CP) where convexity breaks, second derivative < 0
 x_c      = (lambda_1 / (2 * lambda_2) - z_mean / 2 + 1).exp()

 if convex and (x_values < x_c).sum():
   l_c = -x_c.log()                                # loss value at CP
   w_c = lambda_1 + lambda_2 * (l_c - z_mean)    # weight value at CP

   # derivative of the weights*loss at CP
   d_c = -1/x_c * (l_c * lambda_2 + lambda_1 + lambda_2 * (l_c - z_mean))

   # interpolate
   idx          = x_values < x_c
   weights[idx] = (w_c * l_c + d_c * (x_values[idx] - x_c)) / z[idx]
```

```
  return (weights * losses).mean()


#### MODEL ####

class Model(pl.LightningModule):

  """
  Model used in ICCV2019 paper "Symmetric Cross Entropy for Robust Learning␣
 ↪with Noisy Labels"
  Reproduced from https://github.com/HanxunH/SCELoss-Reproduce
  """

  def __init__(self, params):
    super().__init__()

    self.save_hyperparameters(params)

    if self.hparams["dataset"] == "CIFAR-10":

      self.conv_layers = nn.Sequential(
        self.conv_block(3, 64, 3),
        self.conv_block(64, 64, 3),
        nn.MaxPool2d(kernel_size=2, stride=2),
        self.conv_block(64, 128, 3),
        self.conv_block(128, 128, 3),
        nn.MaxPool2d(kernel_size=2, stride=2),
        self.conv_block(128, 196, 3),
        self.conv_block(196, 196, 3),
        nn.MaxPool2d(kernel_size=2, stride=2)
      )

      self.fc_layers = nn.Sequential(
        nn.Linear(3136, 256),
        nn.BatchNorm1d(256),
        nn.ReLU(),
        nn.Linear(256, 10)
      )
      self.acc_metric = Accuracy(average='none', num_classes=10)

      self.acc_clean_metric = Accuracy()
      self.acc_noisy_metric = Accuracy()

    elif self.hparams["dataset"] == "CIFAR-100":
      self.resnet = resnet34(num_classes=100)
      self.acc_metric = Accuracy(average='none', num_classes=100)
```

```
    elif self.hparams["dataset"] == "MNIST":
      self.conv_layers = nn.Sequential(
        self.conv_block(1, 32, 3),
        self.conv_block(32, 32, 3),
        nn.MaxPool2d(kernel_size=3, stride=2),
        self.conv_block(32, 64, 3),
        self.conv_block(64, 64, 3),
        nn.MaxPool2d(kernel_size=3, stride=2),
      )

      self.fc_layers = nn.Sequential(
        nn.Linear(2304, 1152),
        nn.ReLU(inplace=True),
        nn.Linear(1152, 576),
        nn.ReLU(inplace=True),
        nn.Linear(576, 10)
      )

      self.acc_metric = Accuracy(average='none', num_classes=10)

      self.acc_clean_metric = Accuracy()
      self.acc_noisy_metric = Accuracy()


    # self.sceloss = SCELoss(self.hparams["sce_alpha"], self.
↪hparams["sce_beta"])

    self.Z_bar = None

    self.max_acc = 0




  def conv_block(self, in_planes, out_planes, kernel_size=3):
    """Convolutional Block"""

    return nn.Sequential(
      nn.Conv2d(
        in_planes,
        out_planes,
        kernel_size=kernel_size,
        padding=(kernel_size - 1) // 2
      ),
      nn.BatchNorm2d(out_planes),
      nn.ReLU(inplace=True)
    )
```

```
def forward(self, x):
  if self.hparams["dataset"] == "CIFAR-10":
    return self.fc_layers(self.conv_layers(x).view(-1, 3136))

  elif self.hparams["dataset"] == "CIFAR-100":
    return self.resnet(x)

  elif self.hparams["dataset"] == "MNIST":
    return self.fc_layers(self.conv_layers(x).view(-1, 2304))


def calculate_weights(self, losses, log=True):

  Z = losses.detach()

  z_mean = Z.mean()

  l1 = self.hparams["lambda_1"]
  l2 = self.hparams["lambda_2"]
  l3 = self.hparams["lambda_3"]

  weights = (l1 + (Z - z_mean) * l2 + (Z - z_mean)**2 * l3).clip(min=0.0)

  if log:
    self.log_dict({
      "z-mean":     z_mean,
      "z-var":      Z.var(),
      "z-min":      Z.min(),
      "z-max":      Z.max(),
      "w-max":      weights.max(),
      "w-min":      weights.min(),
      "w-mean":     weights.mean(),
    })

  return weights


def training_step(self, batch, batch_idx):
  x, y_target, y_target_original, is_noisy_y = batch
  y_hat       = self(x)


  if self.hparams["loss_function"] == "moments":

    l1, l2  = [1.0, -0.5]
```

```
      self.hparams["lambda_1"] = l1
      self.hparams["lambda_2"] = l2
      self.hparams["lambda_3"] = 0.0

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      loss    = moments_penalization(losses, [l1, l2], normalize=self.
↪hparams["normalize"])

    elif self.hparams["loss_function"] == "moments-var":

      l1, l2  = self.hparams["lambda_1"], self.hparams["lambda_2"]

      # self.hparams["lambda_1"] = l1
      # self.hparams["lambda_2"] = l2
      # self.hparams["lambda_3"] = 0.0

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      loss    = moments_penalization(losses, [l1, l2], normalize=self.
↪hparams["normalize"])

    elif self.hparams["loss_function"] == "moments-convex":

      l1, l2  = self.hparams["lambda_1"], self.hparams["lambda_2"]

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      loss    = moments_penalization_cross_entropy(losses, l1, l2, convex=True)

    elif self.hparams["loss_function"] == "moments-skew":

      l1, l2, l3  = [1.0, -0.5, self.hparams["lambda_3"]]

      self.hparams["lambda_1"] = l1
      self.hparams["lambda_2"] = l2

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      loss    = moments_penalization(losses, [l1, l2, l3])

    elif self.hparams["loss_function"] == "moments-mean":

      l1, l2  = self.hparams["lambda_1"], self.hparams["lambda_2"]

      self.hparams["lambda_1"] = l1
      self.hparams["lambda_2"] = l2
      self.hparams["lambda_3"] = 0.0

      alpha       = self.hparams["moments_alpha"]
```

```
      num_classes = 100 if self.hparams["dataset"] == "CIFAR-100" else 10

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      z       = losses.detach()

      # Compute mean loss for each class
      if not hasattr(self, 'class_mean'):
        self.class_mean = torch.zeros(num_classes, device=z.device)

      z_mean = torch.zeros_like(z)
      for c in range(num_classes):
        mask               = (y_target == c)
        if mask.any():
          self.class_mean[c] = alpha * z[mask].mean() + (1.0 - alpha) * self.
↪class_mean[c]
          z_mean[mask]       = self.class_mean[c]

      if self.hparams["wandb_log_hist"]:
        wandb.log({"class-mean": wandb.Histogram(self.class_mean.cpu().
↪numpy())})

      loss    = moments_penalization(losses, [l1, l2], z_mean=z_mean)

    elif self.hparams["loss_function"] == "normalized-symmetric":

      loss1   = normalized_cross_entropy(y_hat, y_target, num_classes=100 if␣
↪self.hparams["dataset"] == "CIFAR-100" else 10)
      loss2   = reverse_cross_entropy(y_hat, y_target, num_classes=100 if self.
↪hparams["dataset"] == "CIFAR-100" else 10)

      loss    = 10 * loss1 + 1 * loss2

    elif self.hparams["loss_function"] == "term":

      losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
      loss    = term_transformation(losses, t=-0.5)

    elif self.hparams["loss_function"] == "generalized":

      loss    = generalized_cross_entropy(y_hat, y_target, q=0.7,␣
↪num_classes=100 if self.hparams["dataset"] == "CIFAR-100" else 10)

    elif self.hparams["loss_function"] == "taylor":

      loss    = taylor_cross_entropy(y_hat, y_target, t=2, num_classes=100 if␣
↪self.hparams["dataset"] == "CIFAR-100" else 10)
```

```
    elif self.hparams["loss_function"] == "symmetric":

      loss1   = nn.functional.cross_entropy(y_hat, y_target)
      loss2   = reverse_cross_entropy(y_hat, y_target, num_classes=100 if self.
↪hparams["dataset"] == "CIFAR-100" else 10)

      loss    = 0.1 * loss1 + 1.0 * loss2

    elif self.hparams["loss_function"] == "classical":

      loss    = nn.functional.cross_entropy(y_hat, y_target)


    self.log("train_loss", loss)

    return loss


  def validation_step(self, batch, batch_idx, dataloader_idx):
    x, y_target, y_target_original, is_noisy_y = batch
    y_hat       = self(x)
    prob        = torch.softmax(y_hat, dim=1)

    losses  = nn.functional.cross_entropy(y_hat, y_target, reduction="none")
    _, idx  = prob.topk(5, dim=1, sorted=True)
    y_pred  = prob.argmax(dim=1)

    # Compute only for clean data
    if dataloader_idx == 1:
      self.acc_metric(y_pred, y_target)

    # if dataloader_idx == 0:
    #   if (is_noisy_y == False).sum() > 0:
    #     self.acc_clean_metric(y_pred[is_noisy_y == False],␣
↪y_target[is_noisy_y == False])

    #   if (is_noisy_y == True).sum() > 0:
    #     self.acc_noisy_metric(y_pred[is_noisy_y == True], ␣
↪y_target[is_noisy_y == True])

    # losses and weights are returned as array so we can append by summing
    return {
      "num_samples":  len(y_target),
      "topk":         (idx == y_target[:, None]).sum(dim=0).cumsum(0),
      "losses":       [losses,],
      "weights":      [self.calculate_weights(losses, log=False),],
```

```
    }


  def validation_epoch_end(self, validation_outputs):

    class_acc = self.acc_metric.compute()

    self.log_dict({ "class-acc-{}".format(c): a for c, a in␣
↪enumerate(class_acc)})

    # self.log_dict({
    #   "acc-clean": self.acc_clean_metric.compute(),
    #   "acc-noisy": self.acc_noisy_metric.compute(),
    # })

    for ds, outputs in zip(["noisy", "clean"], validation_outputs):
      num_samples = 0
      topk        = 0
      losses      = []
      weights     = []

      for valid_return in outputs:
        num_samples += valid_return["num_samples"]
        topk        += valid_return["topk"]
        losses      += valid_return["losses"]      # Append to array
        weights     += valid_return["weights"]     # Append to array

      topk = topk / num_samples

      self.log_dict({
        ds+"-top-1-acc": topk[0],
        ds+"-top-2-acc": topk[1],
        ds+"-top-5-acc": topk[4],
      })

      if ds == "clean":
        self.max_acc = max(self.max_acc, topk[0].cpu().item())
        self.log("max_acc", self.max_acc)
        print("Epoch:", self.current_epoch, ", Max accuracy:", self.max_acc,␣
↪file=sys.stderr)

      if self.hparams["wandb_log_hist"]:
        wandb.log({ds+"-loss-hist": wandb.Histogram(torch.cat(losses).cpu().
↪numpy())})
        wandb.log({ds+"-weights-hist": wandb.Histogram(torch.cat(weights).cpu().
↪numpy())})
```

```
 def configure_optimizers(self):
   if self.hparams["dataset"] in ["CIFAR-10", "MNIST"]:

       main_optim = torch.optim.SGD(
         [
           {
             "params": self.conv_layers.parameters(),
             "weight_decay": self.hparams["conv_l2_reg"],
           },
           {
             "params": self.fc_layers.parameters(),
             "weight_decay": self.hparams["fc_l2_reg"],
           },
         ],
         lr = self.hparams["learning_rate"],
         momentum = 0.9,
         nesterov = True,
       )

   elif self.hparams["dataset"] == "CIFAR-100":

       main_optim = torch.optim.SGD(
         params = self.resnet.parameters(),
         weight_decay = self.hparams["conv_l2_reg"],
         lr = self.hparams["learning_rate"],
         momentum = 0.9,
         nesterov = True,
       )

   if self.hparams["lr_scheduler"] == "stepped":
     lr_scheduler = torch.optim.lr_scheduler.MultiStepLR(
       main_optim,
       milestones=[self.hparams["lr_milestone"], 2*self.
↪hparams["lr_milestone"]],
       gamma=0.1
     )
   elif self.hparams["lr_scheduler"] == "continuous":
     lr_scheduler = torch.optim.lr_scheduler.LambdaLR(
       main_optim,
       lr_lambda = lambda epoch: 1.0 / 10 ** (epoch / self.
↪hparams["lr_milestone"]),
     )

   return [main_optim], [lr_scheduler]
```

```
 def prepare_data(self):
   """Downloads and splits CIFAR dataset into training, validation, and test"""

   CIFAR_MEAN = [0.49139968, 0.48215827, 0.44653124]
   CIFAR_STD  = [0.24703233, 0.24348505, 0.26158768]

   train_transform = transforms.Compose([
     transforms.RandomCrop(32, padding=4),
     transforms.RandomHorizontalFlip(),
     transforms.ToTensor(),
     transforms.Normalize(CIFAR_MEAN, CIFAR_STD),
   ])

   test_transform = transforms.Compose([
     transforms.ToTensor(),
     transforms.Normalize(CIFAR_MEAN, CIFAR_STD),
   ])

   if self.hparams["dataset"] == "CIFAR-10":
     train_ds = CIFAR10NOISY('data/', True, train_transform, download=True,
                                 asym = self.hparams["asym_noise"],
                                 nosiy_rate = self.hparams["noise_rate"])
     valid_ds_clean = CIFAR10NOISY('data/', False, test_transform,␣
↪download=True,
                                 asym = False, nosiy_rate = 0)

   elif self.hparams["dataset"] == "CIFAR-100":
     train_ds = CIFAR100NOISY('data/', True, train_transform, download=True,
                                 asym = self.hparams["asym_noise"],
                                 nosiy_rate = self.hparams["noise_rate"])
     valid_ds_clean = CIFAR100NOISY('data/', False, test_transform,␣
↪download=True,
                                    asym = False, nosiy_rate = 0)

   elif self.hparams["dataset"] == "MNIST":
     transform = transforms.Compose([
       transforms.ToTensor(),
       transforms.Normalize((0.5,), (0.5,)),
     ])

     train_ds = MNISTNOISY('data/', True, transform, download=True,
                                 asym = self.hparams["asym_noise"],
                                 nosiy_rate = self.hparams["noise_rate"])
     valid_ds_clean = MNISTNOISY('data/', False, transform, download=True,
                                 asym = False, nosiy_rate = 0)
```

```
    # Compute 2 validation datasets, one with clean and one with noisy data
    train_size  = int(len(train_ds) * self.hparams["train_ratio"])
    valid_size  = len(valid_ds_clean)
    unused_size = len(train_ds) - train_size - len(valid_ds_clean)

    train_ds_noisy, valid_ds_noisy, _ = random_split(train_ds, [train_size,␣
↪valid_size, unused_size])

    print(len(train_ds_noisy), len(valid_ds_noisy), len(valid_ds_clean))

    self.train_ds =  train_ds_noisy
    self.valid_ds = (valid_ds_noisy, valid_ds_clean)


  def train_dataloader(self):
    return torch.utils.data.DataLoader(
      dataset = self.train_ds,
      batch_size = self.hparams["batch_size"],
      drop_last = True,
      shuffle = True
    )


  def val_dataloader(self):
    return torch.utils.data.DataLoader(
      dataset = self.valid_ds[0],
      batch_size = self.hparams["batch_size"],
      shuffle = False,
    ), torch.utils.data.DataLoader(
      dataset = self.valid_ds[1],
      batch_size = self.hparams["batch_size"],
      shuffle = False,
    )

if __name__ == "__main__":

  params = {
    "random_seed":          123,
    "dataset":              "CIFAR-100",

    "batch_size":           128,
    "train_ratio":          0.9,

    "max_epochs":           60,
```

```
  "learning_rate":        0.01,
  "lr_scheduler":         "stepped",
  "lr_milestone":         20,
  "conv_l2_reg":          1e-4,
  "fc_l2_reg":            0.01,

  "asym_noise":           False,
  "noise_rate":           0.2,

  "loss_function":        "classical",

  "lambda_1":             1.0,
  "lambda_2":             0.0,
  "lambda_3":             0.0,
  "moments_alpha":        0.01,
  "normalize":            True,

  "wandb_log_hist":       True,
}

# Pass your defaults to wandb.init
wandb.init(config=params)
config = dict(wandb.config)

# Adjust noise magnitude
if config["asym_noise"]:
  config["noise_rate"] /= 2.0

pl.seed_everything(config["random_seed"])
np.random.seed(config["random_seed"])

model   = Model(config)

trainer = pl.Trainer(
  gpus=1,
  max_epochs=config["max_epochs"],
  prepare_data_per_node=True,
  progress_bar_refresh_rate=20,
  logger=[csv_logger, WandbLogger()]
)

trainer.fit(model)

wandb.finish()
```

```
Writing train.py
```

[0]:
```
# Create a sweep

sweep_config = {
  "name": "moments-convex, classical - redo cifar100",
  "program": "train.py",
  "method": "grid",
  "parameters": {
      "loss_function":    { "values": ["moments-convex", "classical"]},
      "noise_rate":       { "values": [0.2, 0.4, 0.6, 0.8]},
      "random_seed":      { "values": [1, 2, 3, 4, 5]},
      "asym_noise":       { "values": [0, 1]},
      "lambda_2":         { "values": [-0.5]},
      "dataset":          { "values": ["CIFAR-100",]},
  }
}

wandb.sweep(sweep_config, project="cifar10-noise")
```

[0]:
```
import multiprocessing


multiprocessing.cpu_count()
```

[0]: 4

[0]:
```
!wandb agent username/cifar10-noise/2ovul8cc >> agent1.log 2>&1 &
```

[0]:
```
!wandb agent username/cifar10-noise/2ovul8cc >> agent2.log 2>&1 &
```

[0]:
```
!wandb agent username/cifar10-noise/2ovul8cc >> agent3.log 2>&1 &
```

[0]:
```
!wandb agent username/cifar10-noise/2ovul8cc >> agent4.log &
```