First of all, data (images) must be filtered out by the BRISQE.

BRISQUE codes with a sample are here:

# coding=utf-8

import os

from ctypes import c\_double

import cv2

import numpy as np

from scipy.special import gamma

import svmutil

from svmutil import gen\_svm\_nodearray

from brisque.utilities import root\_path

class BRISQUE(object):

def \_\_init\_\_(self):

self.\_model = svmutil.svm\_load\_model(root\_path('brisque', 'allmodel'))

self.\_scaler = np.array([

[-1, 1], [0.338, 10], [0.017204, 0.806612], [0.236, 1.642],

[-0.123884, 0.20293],[0.000155, 0.712298], [0.001122, 0.470257],

[0.244, 1.641], [-0.123586, 0.179083], [0.000152, 0.710456],

[0.000975, 0.470984], [0.249, 1.555], [-0.135687, 0.100858],

[0.000174, 0.684173], [0.000913, 0.534174], [0.258, 1.561],

[-0.143408, 0.100486], [0.000179, 0.685696], [0.000888, 0.536508],

[0.471, 3.264], [0.012809, 0.703171], [0.218, 1.046],

[-0.094876, 0.187459], [1.5e-005, 0.442057], [0.001272, 0.40803],

[0.222, 1.042], [-0.115772, 0.162604], [1.6e-005, 0.444362],

[0.001374, 0.40243], [0.227, 0.996],

[-0.117188, 0.09832299999999999], [3e-005, 0.531903],

[0.001122, 0.369589], [0.228, 0.99], [-0.12243, 0.098658],

[2.8e-005, 0.530092], [0.001118, 0.370399]])

@staticmethod

def preprocess\_image(img):

"""Handle any kind of input for our convenience.

:param img: The image path or array.

:type img: str, np.ndarray

"""

if isinstance(img, str):

if os.path.exists(img):

return cv2.imread(img, 0).astype(np.float64)

else:

raise FileNotFoundError('The image is not found on your '

'system.')

elif isinstance(img, np.ndarray):

if len(img.shape) == 2:

image = img

elif len(img.shape) == 3:

image = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

else:

raise ValueError('The image shape is not correct.')

return image.astype(np.float64)

else:

raise ValueError('You can only pass image to the constructor.')

@staticmethod

def \_estimate\_ggd\_param(vec):

"""Estimate GGD parameter.

:param vec: The vector that we want to approximate its parameter.

:type vec: np.ndarray

"""

gam = np.arange(0.2, 10 + 0.001, 0.001)

r\_gam = (gamma(1.0 / gam) \* gamma(3.0 / gam) / (gamma(2.0 / gam) \*\* 2))

sigma\_sq = np.mean(vec \*\* 2)

sigma = np.sqrt(sigma\_sq)

E = np.mean(np.abs(vec))

rho = sigma\_sq / E \*\* 2

differences = abs(rho - r\_gam)

array\_position = np.argmin(differences)

gamparam = gam[array\_position]

return gamparam, sigma

@staticmethod

def \_estimate\_aggd\_param(vec):

"""Estimate AGGD parameter.

:param vec: The vector that we want to approximate its parameter.

:type vec: np.ndarray

"""

gam = np.arange(0.2, 10 + 0.001, 0.001)

r\_gam = ((gamma(2.0 / gam)) \*\* 2) / (

gamma(1.0 / gam) \* gamma(3.0 / gam))

left\_std = np.sqrt(np.mean((vec[vec < 0]) \*\* 2))

right\_std = np.sqrt(np.mean((vec[vec > 0]) \*\* 2))

gamma\_hat = left\_std / right\_std

rhat = (np.mean(np.abs(vec))) \*\* 2 / np.mean((vec) \*\* 2)

rhat\_norm = (rhat \* (gamma\_hat \*\* 3 + 1) \* (gamma\_hat + 1)) / (

(gamma\_hat \*\* 2 + 1) \*\* 2)

differences = (r\_gam - rhat\_norm) \*\* 2

array\_position = np.argmin(differences)

alpha = gam[array\_position]

return alpha, left\_std, right\_std

def get\_feature(self, img):

"""Get brisque feature given an image.

:param img: The path or array of the image.

:type img: str, np.ndarray

"""

imdist = self.preprocess\_image(img)

scale\_num = 2

feat = np.array([])

for itr\_scale in range(scale\_num):

mu = cv2.GaussianBlur(

imdist, (7, 7), 7 / 6, borderType=cv2.BORDER\_CONSTANT)

mu\_sq = mu \* mu

sigma = cv2.GaussianBlur(

imdist \* imdist, (7, 7), 7 / 6, borderType=cv2.BORDER\_CONSTANT)

sigma = np.sqrt(abs((sigma - mu\_sq)))

structdis = (imdist - mu) / (sigma + 1)

alpha, overallstd = self.\_estimate\_ggd\_param(structdis)

feat = np.append(feat, [alpha, overallstd \*\* 2])

shifts = [[0, 1], [1, 0], [1, 1], [-1, 1]]

for shift in shifts:

shifted\_structdis = np.roll(

np.roll(structdis, shift[0], axis=0), shift[1], axis=1)

pair = np.ravel(structdis, order='F') \* \

np.ravel(shifted\_structdis, order='F')

alpha, left\_std, right\_std = self.\_estimate\_aggd\_param(pair)

const = np.sqrt(gamma(1 / alpha)) / np.sqrt(gamma(3 / alpha))

mean\_param = (right\_std - left\_std) \* (

gamma(2 / alpha) / gamma(1 / alpha)) \* const

feat = np.append(

feat, [alpha, mean\_param, left\_std \*\* 2, right\_std \*\* 2])

imdist = cv2.resize(

imdist,

(0, 0),

fx=0.5,

fy=0.5,

interpolation=cv2.INTER\_NEAREST

)

return feat

def get\_score(self, img):

"""Get brisque score given an image.

:param img: The path or array of the image.

:type img: str, np.ndarray

"""

feature = self.get\_feature(img)

scaled\_feature = self.\_scale\_feature(feature)

return self.\_calculate\_score(scaled\_feature)

def \_scale\_feature(self, feature):

"""Scale feature with svm scaler.

:param feature: Brisque unscaled feature.

:type feature: np.ndarray

"""

y\_lower = self.\_scaler[0][0]

y\_upper = self.\_scaler[0][1]

y\_min = self.\_scaler[1:, 0]

y\_max = self.\_scaler[1:, 1]

scaled\_feat = y\_lower + (y\_upper - y\_lower) \* ((feature - y\_min) / (

y\_max - y\_min))

return scaled\_feat

def \_calculate\_score(self, scaled\_feature):

"""Calculate score from scaled brisque feature.

:param scaled\_feature: Scaled brisque feature.

:type scaled\_feature: np.ndarray

"""

x, idx = gen\_svm\_nodearray(

scaled\_feature.tolist(),

isKernel=(self.\_model.param.kernel\_type == 'PRECOMPUTED')

)

nr\_classifier = 1

prob\_estimates = (c\_double \* nr\_classifier)()

return svmutil.libsvm.svm\_predict\_probability(

self.\_model, x, prob\_estimates)

……………………..

……………………..

In Fuzzification stage, for extracting patches we can use “sklearn” or “skimage”

For example:

>>> from sklearn.feature\_extraction.image import extract\_patches\_2d

>>> import numpy as np

>>> A = np.arange(4\*4).reshape(4,4)

>>> window\_shape = (2, 2)

>>> B = extract\_patches\_2d(A, window\_shape)

>>> B[0]

array([[0, 1],

[4, 5]])

>>> B

array([[[ 0, 1],

[ 4, 5]],

[[ 1, 2],

[ 5, 6]],

[[ 2, 3],

[ 6, 7]],

[[ 4, 5],

[ 8, 9]],

[[ 5, 6],

[ 9, 10]],

[[ 6, 7],

[10, 11]],

[[ 8, 9],

[12, 13]],

[[ 9, 10],

[13, 14]],

[[10, 11],

[14, 15]]])

………………………………….

Or:

>>> from skimage.util import view\_as\_windows

>>> import numpy as np

>>> A = np.arange(4\*4).reshape(4,4)

>>> A

array([[ 0, 1, 2, 3],

[ 4, 5, 6, 7],

[ 8, 9, 10, 11],

[12, 13, 14, 15]])

>>> window\_shape = (2, 2)

>>> B = view\_as\_windows(A, window\_shape)

>>> B[0]

array([[[0, 1],

[4, 5]],

[[1, 2],

[5, 6]],

[[2, 3],

[6, 7]]])

>>> B

array([[[[ 0, 1],

[ 4, 5]],

[[ 1, 2],

[ 5, 6]],

[[ 2, 3],

[ 6, 7]]],

[[[ 4, 5],

[ 8, 9]],

[[ 5, 6],

[ 9, 10]],

[[ 6, 7],

[10, 11]]],

[[[ 8, 9],

[12, 13]],

[[ 9, 10],

[13, 14]],

[[10, 11],

[14, 15]]]])

……………………………………

Membership functions μ1, μ2, and μ3 are used to represent small, medium, and large values.

In our project, we only used three membership functions (Triangular and Trapezoids).

Simple, we can use “skfuzzy.membership”

Or write from scratch:

generatemf.py: Library of standard fuzzy membership function generators.

"""

import numpy as np

def \_nearest(x, y0):

"""

Finds the index of the sequence elemnt value x0 in `x` that is closest

to the provided value, `y0`.

Parameters

----------

x : 1d array

Input sequence.

y0 : float

Desired matching value.

Returns

-------

idx0 : int

Index of the nearest value `x0` in x; e.g. x[idx0] = x0.

x0 : float

Value in `x` which is closest to `y0`.

Notes

-----

This function does support extrapolation.

"""

# Distance map

d = np.abs(x - y0)

idx0 = np.nonzero(d == d.min())[0][0]

return idx0, x[idx0]

[docs]def dsigmf(x, b1, c1, b2, c2):

"""

Difference of two fuzzy sigmoid membership functions.

Parameters

----------

x : 1d array

Independent variable.

b1 : float

Midpoint of first sigmoid; f1(b1) = 0.5

c1 : float

Width and sign of first sigmoid.

b2 : float

Midpoint of second sigmoid; f2(b2) = 0.5

c2 : float

Width and sign of second sigmoid.

Returns

-------

y : 1d array

Generated sigmoid values, defined as

y = f1 - f2

f1(x) = 1 / (1. + exp[- c1 \* (x - b1)])

f2(x) = 1 / (1. + exp[- c2 \* (x - b2)])

"""

return sigmf(x, b1, c=c1) - sigmf(x, b2, c=c2)

[docs]def gaussmf(x, mean, sigma):

"""

Gaussian fuzzy membership function.

Parameters

----------

x : 1d array or iterable

Independent variable.

mean : float

Gaussian parameter for center (mean) value.

sigma : float

Gaussian parameter for standard deviation.

Returns

-------

y : 1d array

Gaussian membership function for x

"""

return np.exp(-((x - mean) \*\* 2.) / float(sigma) \*\* 2.)

[docs]def gauss2mf(x, mean1, sigma1, mean2, sigma2):

"""

Gaussian fuzzy membership function of two combined Gaussians.

Parameters

----------

x : 1d array or iterable

Independent variable.

mean1 : float

Gaussian parameter for center (mean) value of left-side Gaussian.

Note mean1 <= mean2 reqiured.

sigma1 : float

Standard deviation of left Gaussian.

mean2 : float

Gaussian parameter for center (mean) value of right-side Gaussian.

Note mean2 >= mean1 required.

sigma2 : float

Standard deviation of right Gaussian.

Returns

-------

y : 1d array

Membership function with left side up to `mean1` defined by the first

Gaussian, and the right side above `mean2` defined by the second.

In the range mean1 <= x <= mean2 the function has value = 1.

"""

assert mean1 <= mean2, 'mean1 <= mean2 is required. See docstring.'

y = np.ones(len(x))

idx1 = x <= mean1

idx2 = x > mean2

y[idx1] = gaussmf(x[idx1], mean1, sigma1)

y[idx2] = gaussmf(x[idx2], mean2, sigma2)

return y

[docs]def gbellmf(x, a, b, c):

"""

Generalized Bell function fuzzy membership generator.

Parameters

----------

x : 1d array

Independent variable.

a : float

Bell function parameter controlling width. See Note for definition.

b : float

Bell function parameter controlling center. See Note for definition.

c : float

Bell function parameter controlling slope. See Note for definition.

Returns

-------

y : 1d array

Generalized Bell fuzzy membership function.

Notes

-----

Definition of Generalized Bell function is:

y(x) = 1 / (1 + abs([x - c] / a) \*\* [2 \* b])

"""

return 1. / (1. + np.abs((x - c) / a) \*\* (2 \* b))

[docs]def piecemf(x, abc):

"""

Piecewise linear membership function (particularly used in FIRE filters).

Parameters

----------

x : 1d array

Independent variable vector.

abc : 1d array, length 3

Defines the piecewise function. Important: if abc = [a, b, c] then

a <= b <= c is REQUIRED!

Returns

-------

y : 1d array

Piecewise fuzzy membership function for x.

Notes

-----

Piecewise definition:

y = 0, min(x) <= x <= a

y = b(x - a)/c(b - a), a <= x <= b

y = x/c, b <= x <= c

"""

a, b, c = abc

if c != x.max():

c = x.max()

assert a <= b and b <= c, '`abc` requires a <= b <= c.'

n = len(x)

y = np.zeros(n)

idx0 = \_nearest(x, 0)[0]

idxa = \_nearest(x, a)[0]

idxb = \_nearest(x, b)[0]

n = np.r\_[0:n - idx0]

y[idx0 + n] = n / float(c)

y[idx0:idxa] = 0

m = np.r\_[0:idxb - idxa]

y[idxa:idxb] = b \* m / (float(c) \* (b - a))

return y / y.max()

[docs]def pimf(x, a, b, c, d):

"""

Pi-function fuzzy membership generator.

Parameters

----------

x : 1d array

Independent variable.

a : float

Left 'foot', where the function begins to climb from zero.

b : float

Left 'ceiling', where the function levels off at 1.

c : float

Right 'ceiling', where the function begins falling from 1.

d : float

Right 'foot', where the function reattains zero.

Returns

-------

y : 1d array

Pi-function.

Notes

-----

This is equivalently a product of smf and zmf.

"""

y = np.ones(len(x))

assert a <= b and b <= c and c <= d, 'a <= b <= c <= d is required.'

idx = x <= a

y[idx] = 0

idx = np.logical\_and(a <= x, x <= (a + b) / 2.)

y[idx] = 2. \* ((x[idx] - a) / (b - a)) \*\* 2.

idx = np.logical\_and((a + b) / 2. < x, x <= b)

y[idx] = 1 - 2. \* ((x[idx] - b) / (b - a)) \*\* 2.

idx = np.logical\_and(c <= x, x < (c + d) / 2.)

y[idx] = 1 - 2. \* ((x[idx] - c) / (d - c)) \*\* 2.

idx = np.logical\_and((c + d) / 2. <= x, x <= d)

y[idx] = 2. \* ((x[idx] - d) / (d - c)) \*\* 2.

idx = x >= d

y[idx] = 0

return y

[docs]def psigmf(x, b1, c1, b2, c2):

"""

Product of two sigmoid membership functions.

Parameters

----------

x : 1d array

Data vector for independent variable.

b1 : float

Offset or bias for the first sigmoid. This is the center value of the

sigmoid, where it equals 1/2.

c1 : float

Controls 'width' of the first sigmoidal region about `b1` (magnitude),

and also which side of the function is open (sign). A positive value of

`c1` means the left side approaches zero while the right side

approaches one; a negative value of `c1` means the opposite.

b2 : float

Offset or bias for the second sigmoid. This is the center value of the

sigmoid, where it equals 1/2.

c2 : float

Controls 'width' of the second sigmoidal region about `b2` (magnitude),

and also which side of the function is open (sign). A positive value of

`c2` means the left side approaches zero while the right side

approaches one; a negative value of `c2` means the opposite.

Returns

-------

y : 1d array

Generated sigmoid values, defined as

y = f1(x) \* f2(x)

f1(x) = 1 / (1. + exp[- c1 \* (x - b1)])

f2(x) = 1 / (1. + exp[- c2 \* (x - b2)])

Notes

-----

For a smoothed rect-like function, c2 < 0 < c1. For its inverse (zero in

middle, one at edges) c1 < 0 < c2.

"""

return sigmf(x, b1, c1) \* sigmf(x, b2, c2)

def sigmoid(wx, b):

"""

Generates a sigmoid function.

Parameters

----------

wx : 2d array, (K, N)

Sum of the inner product of W and X, where W is a KxM data matrix

and X is a MxN weight matrix.

b : 1d array, length K

Bias or threshold.

Returns

-------

sigmoid : 2d array, (K, N)

Sigmoid function result.

"""

return 1. / (1. + np.exp(-(wx + np.dot(np.atleast\_2d(b).T,

np.ones((1, wx.shape[1]))))))

[docs]def sigmf(x, b, c):

"""

The basic sigmoid membership function generator.

Parameters

----------

x : 1d array

Data vector for independent variable.

b : float

Offset or bias. This is the center value of the sigmoid, where it

equals 1/2.

c : float

Controls 'width' of the sigmoidal region about `b` (magnitude); also

which side of the function is open (sign). A positive value of `a`

means the left side approaches 0.0 while the right side approaches 1.;

a negative value of `c` means the opposite.

Returns

-------

y : 1d array

Generated sigmoid values, defined as y = 1 / (1. + exp[- c \* (x - b)])

Notes

-----

These are the same values, provided separately and in the opposite order

compared to the publicly available MathWorks' Fuzzy Logic Toolbox

documentation. Pay close attention to above docstring!

"""

return 1. / (1. + np.exp(- c \* (x - b)))

[docs]def smf(x, a, b):

"""

S-function fuzzy membership generator.

Parameters

----------

x : 1d array

Independent variable.

a : float

'foot', where the function begins to climb from zero.

b : float

'ceiling', where the function levels off at 1.

Returns

-------

y : 1d array

S-function.

Notes

-----

Named such because of its S-like shape.

"""

assert a <= b, 'a <= b is required.'

y = np.ones(len(x))

idx = x <= a

y[idx] = 0

idx = np.logical\_and(a <= x, x <= (a + b) / 2.)

y[idx] = 2. \* ((x[idx] - a) / (b - a)) \*\* 2.

idx = np.logical\_and((a + b) / 2. <= x, x <= b)

y[idx] = 1 - 2. \* ((x[idx] - b) / (b - a)) \*\* 2.

return y

[docs]def trapmf(x, abcd):

"""

Trapezoidal membership function generator.

Parameters

----------

x : 1d array

Independent variable.

abcd : 1d array, length 4

Four-element vector. Ensure a <= b <= c <= d.

Returns

-------

y : 1d array

Trapezoidal membership function.

"""

assert len(abcd) == 4, 'abcd parameter must have exactly four elements.'

a, b, c, d = np.r\_[abcd]

assert a <= b and b <= c and c <= d, 'abcd requires the four elements \

a <= b <= c <= d.'

y = np.ones(len(x))

idx = np.nonzero(x <= b)[0]

y[idx] = trimf(x[idx], np.r\_[a, b, b])

idx = np.nonzero(x >= c)[0]

y[idx] = trimf(x[idx], np.r\_[c, c, d])

idx = np.nonzero(x < a)[0]

y[idx] = np.zeros(len(idx))

idx = np.nonzero(x > d)[0]

y[idx] = np.zeros(len(idx))

return y

[docs]def trimf(x, abc):

"""

Triangular membership function generator.

Parameters

----------

x : 1d array

Independent variable.

abc : 1d array, length 3

Three-element vector controlling shape of triangular function.

Requires a <= b <= c.

Returns

-------

y : 1d array

Triangular membership function.

"""

assert len(abc) == 3, 'abc parameter must have exactly three elements.'

a, b, c = np.r\_[abc] # Zero-indexing in Python

assert a <= b and b <= c, 'abc requires the three elements a <= b <= c.'

y = np.zeros(len(x))

# Left side

if a != b:

idx = np.nonzero(np.logical\_and(a < x, x < b))[0]

y[idx] = (x[idx] - a) / float(b - a)

# Right side

if b != c:

idx = np.nonzero(np.logical\_and(b < x, x < c))[0]

y[idx] = (c - x[idx]) / float(c - b)

idx = np.nonzero(x == b)

y[idx] = 1

return y

[docs]def zmf(x, a, b):

"""

Z-function fuzzy membership generator.

Parameters

----------

x : 1d array

Independent variable.

a : float

'ceiling', where the function begins falling from 1.

b : float

'foot', where the function reattains zero.

Returns

-------

y : 1d array

Z-function.

Notes

-----

Named such because of its Z-like shape.

"""

assert a <= b, 'a <= b is required.'

y = np.ones(len(x))

idx = np.logical\_and(a <= x, x < (a + b) / 2.)

y[idx] = 1 - 2. \* ((x[idx] - a) / (b - a)) \*\* 2.

idx = np.logical\_and((a + b) / 2. <= x, x <= b)

y[idx] = 2. \* ((x[idx] - b) / (b - a)) \*\* 2.

idx = x >= b

y[idx] = 0

return y

…………………………………….

“fuzzy-expert.operator” can be used for fuzzy algebraic sum:

For example:

def prob\_or(memberships: List[npt.ArrayLike]) -> npt.ArrayLike:

"""

Applies the element-wise function fn(u, v) = u + v - u \* v. Also known as algebraic-sum.

:param memberships: List of arrays of membership values.

>>> from fuzzy\_expert.operators import prob\_or

>>> x = [0.1, 0.25, 0.5, 0.75, 0.3]

>>> y = [0, 0.75, 0.5, 0.25, 0]

>>> prob\_or([x, y])

array([0.1 , 0.8125, 0.75 , 0.8125, 0.3 ])

"""

result: npt.ArrayLike = np.array(memberships[0])

for membership in memberships[1:]:

membership = np.array(membership)

result: npt.ArrayLike = result + membership - result \* membership

return np.maximum(0, np.minimum(1, result))

…………………………………………

fuzzy result must be converted back into a single number. This is known as defuzzification.

Lets give an example with a trapezoidal membership function:

import skfuzzy as fuzz

# Generate trapezoidal membership function on range [0, 1]

x = np.arange(0, 5.05, 0.1)

mfx = fuzz.trapmf(x, [2, 2.5, 3, 4.5])

# Defuzzify this membership function five ways

defuzz\_centroid = fuzz.defuzz(x, mfx, 'centroid') # Same as skfuzzy.centroid

defuzz\_bisector = fuzz.defuzz(x, mfx, 'bisector')

defuzz\_mom = fuzz.defuzz(x, mfx, 'mom')

defuzz\_som = fuzz.defuzz(x, mfx, 'som')

defuzz\_lom = fuzz.defuzz(x, mfx, 'lom')

# Collect info for vertical lines

labels = ['centroid', 'bisector', 'mean of maximum', 'min of maximum',

'max of maximum']

xvals = [defuzz\_centroid,

defuzz\_bisector,

defuzz\_mom,

defuzz\_som,

defuzz\_lom]

colors = ['r', 'b', 'g', 'c', 'm']

ymax = [fuzz.interp\_membership(x, mfx, i) for i in xvals]

……………………………………..

import torch

import torch.nn as nn

def log\_sum\_exp(x):

# See implementation detail in

# b is a shift factor

# x.size() = [N, C]:

b, \_ = torch.max(x, 1)

y = b + torch.log(torch.exp(x - b.expand\_as(x)).sum(1))

# y.size() = [N, 1]. Squeeze to [N] and return

return y.squeeze(1)

def class\_select(logits, target):

# in numpy, this would be logits[:, target].

batch\_size, num\_classes = logits.size()

if target.is\_cuda:

device = target.data.get\_device()

one\_hot\_mask = torch.autograd.Variable(torch.arange(0, num\_classes)

.long()

.repeat(batch\_size, 1)

.cuda(device)

.eq(target.data.repeat(num\_classes, 1).t()))

else:

one\_hot\_mask = torch.autograd.Variable(torch.arange(0, num\_classes)

.long()

.repeat(batch\_size, 1)

.eq(target.data.repeat(num\_classes, 1).t()))

return logits.masked\_select(one\_hot\_mask)

def cross\_entropy\_with\_weights(logits, target, weights=None):

assert logits.dim() == 2

assert not target.requires\_grad

target = target.squeeze(1) if target.dim() == 2 else target

assert target.dim() == 1

loss = log\_sum\_exp(logits) - class\_select(logits, target)

if weights is not None:

# loss.size() = [N]. Assert weights has the same shape

assert list(loss.size()) == list(weights.size())

# Weight the loss

loss = loss \* weights

return loss

class CrossEntropyLoss(nn.Module):

"""

Cross entropy with instance-wise weights. Leave `aggregate` to None to obtain a loss

vector of shape (batch\_size,).

"""

def \_\_init\_\_(self, aggregate='mean'):

super(CrossEntropyLoss, self).\_\_init\_\_()

assert aggregate in ['sum', 'mean', None]

self.aggregate = aggregate

def forward(self, input, target, weights=None):

if self.aggregate == 'sum':

return cross\_entropy\_with\_weights(input, target, weights).sum()

elif self.aggregate == 'mean':

return cross\_entropy\_with\_weights(input, target, weights).mean()

elif self.aggregate is None:

return cross\_entropy\_with\_weights(input, target, weights)

If we consider Weight: