finetune.py

```
#    Copyright 2020 Division of Medical Image Computing, German Cancer Research
Center (DKFZ), Heidelberg, Germany
#
#    Licensed under the Apache License, Version 2.0 (the "License");
#    you may not use this file except in compliance with the License.
#    You may obtain a copy of the License at
#
#        http://www.apache.org/licenses/LICENSE-2.0
#
#    Unless required by applicable law or agreed to in writing, software
#    distributed under the License is distributed on an "AS IS" BASIS,
#    WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
#    See the License for the specific language governing permissions and
#    limitations under the License.


import torch
from nnunet.training.loss_functions.TopK_loss import TopKLoss
from nnunet.training.loss_functions.crossentropy import RobustCrossEntropyLoss
from nnunet.utilities.nd_softmax import softmax_helper
from nnunet.utilities.tensor_utilities import sum_tensor
from torch import nn
import numpy as np


class GDL(nn.Module):
    def __init__(self, apply_nonlin=None, batch_dice=False, do_bg=True,
smooth=1.,
                 square=False, square_volumes=False):
        """
        square_volumes will square the weight term. The paper recommends
square_volumes=True; I don't (just an intuition)
        """
        super(GDL, self).__init__()

        self.square_volumes = square_volumes
        self.square = square
        self.do_bg = do_bg
        self.batch_dice = batch_dice
        self.apply_nonlin = apply_nonlin
        self.smooth = smooth

    def forward(self, x, y, loss_mask=None):
        shp_x = x.shape
        shp_y = y.shape

        if self.batch_dice:
            axes = [0] + list(range(2, len(shp_x)))
        else:
            axes = list(range(2, len(shp_x)))

        if len(shp_x) != len(shp_y):
            y = y.view((shp_y[0], 1, *shp_y[1:]))
        if all([i == j for i, j in zip(x.shape, y.shape)]):
            # if this is the case then gt is probably already a one hot encoding
            y_onehot = y
        else:
            gt = y.long()
            y_onehot = torch.zeros(shp_x)
            if x.device.type == "cuda":
                y_onehot = y_onehot.cuda(x.device.index)
            y_onehot.scatter_(1, gt, 1)

        if self.apply_nonlin is not None:
            x = self.apply_nonlin(x)

        if not self.do_bg:
            x = x[:, 1:]
            y_onehot = y_onehot[:, 1:]

        tp, fp, fn, _ = get_tp_fp_fn_tn(x, y_onehot, axes, loss_mask,
self.square)

        # GDL weight computation, we use 1/V
        volumes = sum_tensor(y_onehot, axes) + 1e-6 # add some eps to prevent
div by zero

        if self.square_volumes:
            volumes = volumes ** 2

        # apply weights
        tp = tp / volumes
        fp = fp / volumes
        fn = fn / volumes

        # sum over classes
        if self.batch_dice:
            axis = 0
        else:
            axis = 1

        tp = tp.sum(axis, keepdim=False)
        fp = fp.sum(axis, keepdim=False)
        fn = fn.sum(axis, keepdim=False)

        # compute dice
        dc = (2 * tp + self.smooth) / (2 * tp + fp + fn + self.smooth)

        dc = dc.mean()

        return -dc


def get_tp_fp_fn_tn(net_output, gt, axes=None, mask=None, square=False):
    """
    net_output must be (b, c, x, y(, z)))
    gt must be a label map (shape (b, 1, x, y(, z)) OR shape (b, x, y(, z))) or
one hot encoding (b, c, x, y(, z))
    if mask is provided it must have shape (b, 1, x, y(, z)))
    :param net_output:
```

```
    :param gt:
    :param axes: can be (, ) = no summation
    :param mask: mask must be 1 for valid pixels and 0 for invalid pixels
    :param square: if True then fp, tp and fn will be squared before summation
    :return:
    """
    if axes is None:
        axes = tuple(range(2, len(net_output.size())))

    shp_x = net_output.shape
    shp_y = gt.shape

    with torch.no_grad():
        if len(shp_x) != len(shp_y):
            gt = gt.view((shp_y[0], 1, *shp_y[1:]))

        if all([i == j for i, j in zip(net_output.shape, gt.shape)]):
            # if this is the case then gt is probably already a one hot encoding
            y_onehot = gt
        else:
            gt = gt.long()
            y_onehot = torch.zeros(shp_x)
            if net_output.device.type == "cuda":
                y_onehot = y_onehot.cuda(net_output.device.index)
            y_onehot.scatter_(1, gt, 1)

    tp = net_output * y_onehot
    fp = net_output * (1 - y_onehot)
    fn = (1 - net_output) * y_onehot
    tn = (1 - net_output) * (1 - y_onehot)

    if mask is not None:
        tp = torch.stack(tuple(x_i * mask[:, 0] for x_i in torch.unbind(tp, dim=1)), dim=1)
        fp = torch.stack(tuple(x_i * mask[:, 0] for x_i in torch.unbind(fp, dim=1)), dim=1)
        fn = torch.stack(tuple(x_i * mask[:, 0] for x_i in torch.unbind(fn, dim=1)), dim=1)
        tn = torch.stack(tuple(x_i * mask[:, 0] for x_i in torch.unbind(tn, dim=1)), dim=1)

    if square:
        tp = tp ** 2
        fp = fp ** 2
        fn = fn ** 2
        tn = tn ** 2

    if len(axes) > 0:
        tp = sum_tensor(tp, axes, keepdim=False)
        fp = sum_tensor(fp, axes, keepdim=False)
        fn = sum_tensor(fn, axes, keepdim=False)
        tn = sum_tensor(tn, axes, keepdim=False)

    return tp, fp, fn, tn


class SoftDiceLoss(nn.Module):
    def __init__(self, apply_nonlin=None, batch_dice=False, do_bg=True,
smooth=1.):
        """
        """
        super(SoftDiceLoss, self).__init__()

        self.do_bg = do_bg
        self.batch_dice = batch_dice
        self.apply_nonlin = apply_nonlin
        self.smooth = smooth

    def forward(self, x, y, loss_mask=None):
        shp_x = x.shape

        if self.batch_dice:
            axes = [0] + list(range(2, len(shp_x)))
        else:
            axes = list(range(2, len(shp_x)))

        if self.apply_nonlin is not None:
            x = self.apply_nonlin(x)

        tp, fp, fn, _ = get_tp_fp_fn_tn(x, y, axes, loss_mask, False)

        nominator = 2 * tp + self.smooth
        denominator = 2 * tp + fp + fn + self.smooth

        dc = nominator / (denominator + 1e-8)

        if not self.do_bg:
            if self.batch_dice:
                dc = dc[1:]
            else:
                dc = dc[:, 1:]
        dc = dc.mean()

        return -dc


class MCCLoss(nn.Module):
    def __init__(self, apply_nonlin=None, batch_mcc=False, do_bg=True,
smooth=0.0):
        """
        based on matthews correlation coefficient
        https://en.wikipedia.org/wiki/Matthews_correlation_coefficient

        Does not work. Really unstable. F this.
        """
        super(MCCLoss, self).__init__()

        self.smooth = smooth
        self.do_bg = do_bg
        self.batch_mcc = batch_mcc
        self.apply_nonlin = apply_nonlin

    def forward(self, x, y, loss_mask=None):
        shp_x = x.shape
        voxels = np.prod(shp_x[2:])
        if self.batch_mcc:
            axes = [0] + list(range(2, len(shp_x)))
        else:
            axes = list(range(2, len(shp_x)))

        if self.apply_nonlin is not None:
            x = self.apply_nonlin(x)

        tp, fp, fn, tn = get_tp_fp_fn_tn(x, y, axes, loss_mask, False)
        tp /= voxels
        fp /= voxels
        fn /= voxels
        tn /= voxels

        nominator = tp * tn - fp * fn + self.smooth
        denominator = ((tp + fp) * (tp + fn) * (tn + fp) * (tn + fn)) ** 0.5 +
self.smooth

        mcc = nominator / denominator

        if not self.do_bg:
            if self.batch_mcc:
                mcc = mcc[1:]
            else:
                mcc = mcc[:, 1:]
        mcc = mcc.mean()

        return -mcc


class SoftDiceLossSquared(nn.Module):
    def __init__(self, apply_nonlin=None, batch_dice=False, do_bg=True,
smooth=1.):
        """
        squares the terms in the denominator as proposed by Milletari et al.
        """
        super(SoftDiceLossSquared, self).__init__()

        self.do_bg = do_bg
        self.batch_dice = batch_dice
        self.apply_nonlin = apply_nonlin
        self.smooth = smooth

    def forward(self, x, y, loss_mask=None):
        shp_x = x.shape
        shp_y = y.shape

        if self.batch_dice:
            axes = [0] + list(range(2, len(shp_x)))
        else:
            axes = list(range(2, len(shp_x)))

        if self.apply_nonlin is not None:
            x = self.apply_nonlin(x)

        with torch.no_grad():
            if len(shp_x) != len(shp_y):
                y = y.view((shp_y[0], 1, *shp_y[1:]))

            if all([i == j for i, j in zip(x.shape, y.shape)]):
                # if this is the case then gt is probably already a one hot
encoding
                y_onehot = y
            else:
                y = y.long()
                y_onehot = torch.zeros(shp_x)
                if x.device.type == "cuda":
                    y_onehot = y_onehot.cuda(x.device.index)
                y_onehot.scatter_(1, y, 1).float()

        intersect = x * y_onehot
        # values in the denominator get smoothed
        denominator = x ** 2 + y_onehot ** 2

        # aggregation was previously done in get_tp_fp_fn, but needs to be done
here now (needs to be done after
        # squaring)
        intersect = sum_tensor(intersect, axes, False) + self.smooth
        denominator = sum_tensor(denominator, axes, False) + self.smooth

        dc = 2 * intersect / denominator

        if not self.do_bg:
            if self.batch_dice:
                dc = dc[1:]
            else:
                dc = dc[:, 1:]
        dc = dc.mean()

        return -dc


class DC_and_CE_loss(nn.Module):
    def __init__(self, soft_dice_kwargs, ce_kwargs, aggregate="sum",
square_dice=False, weight_ce=1, weight_dice=1,
                 log_dice=False, ignore_label=None):
        """
        CAREFUL. Weights for CE and Dice do not need to sum to one. You can set
whatever you want.
        :param soft_dice_kwargs:
        :param ce_kwargs:
        :param aggregate:
        :param square_dice:
        :param weight_ce:
        :param weight_dice:
        """
        super(DC_and_CE_loss, self).__init__()
        if ignore_label is not None:
            assert not square_dice, 'not implemented'
            ce_kwargs['reduction'] = 'none'
        self.log_dice = log_dice
        self.weight_dice = weight_dice
        self.weight_ce = weight_ce
        self.aggregate = aggregate
        self.ce = RobustCrossEntropyLoss(**ce_kwargs)
        self.ignore_label = ignore_label

        if not square_dice:
            self.dc = SoftDiceLoss(apply_nonlin=softmax_helper,
**soft_dice_kwargs)
        else:
            self.dc = SoftDiceLossSquared(apply_nonlin=softmax_helper,
**soft_dice_kwargs)

    def forward(self, net_output, target):
        """
        target must be b, c, x, y(, z) with c=1
        :param net_output:
        :param target:
        :return:
        """
        if self.ignore_label is not None:
            assert target.shape[1] == 1, 'not implemented for one hot encoding'
            mask = target != self.ignore_label
            target[~mask] = 0
            mask = mask.float()
        else:
            mask = None

        dc_loss = self.dc(net_output, target, loss_mask=mask) if
self.weight_dice != 0 else 0
        if self.log_dice:
            dc_loss = -torch.log(-dc_loss)

        ce_loss = self.ce(net_output, target[:, 0].long()) if self.weight_ce !=
0 else 0
        if self.ignore_label is not None:
            ce_loss *= mask[:, 0]
            ce_loss = ce_loss.sum() / mask.sum()

        if self.aggregate == "sum":
            result = self.weight_ce * ce_loss + self.weight_dice * dc_loss
        else:
            raise NotImplementedError("nah son") # reserved for other stuff
(later)
        return result


class DC_and_BCE_loss(nn.Module):
    def __init__(self, bce_kwargs, soft_dice_kwargs, aggregate="sum"):
        """
        DO NOT APPLY NONLINEARITY IN YOUR NETWORK!

        THIS LOSS IS INTENDED TO BE USED FOR BRATS REGIONS ONLY
        :param soft_dice_kwargs:
        :param bce_kwargs:
        :param aggregate:
        """
        super(DC_and_BCE_loss, self).__init__()

        self.aggregate = aggregate
        self.ce = nn.BCEWithLogitsLoss(**bce_kwargs)
        self.dc = SoftDiceLoss(apply_nonlin=torch.sigmoid, **soft_dice_kwargs)

    def forward(self, net_output, target):
        ce_loss = self.ce(net_output, target)
        dc_loss = self.dc(net_output, target)

        if self.aggregate == "sum":
            result = ce_loss + dc_loss
        else:
            raise NotImplementedError("nah son") # reserved for other stuff
(later)

        return result


class GDL_and_CE_loss(nn.Module):
    def __init__(self, gdl_dice_kwargs, ce_kwargs, aggregate="sum"):
        super(GDL_and_CE_loss, self).__init__()
        self.aggregate = aggregate
        self.ce = RobustCrossEntropyLoss(**ce_kwargs)
        self.dc = GDL(softmax_helper, **gdl_dice_kwargs)

    def forward(self, net_output, target):
        dc_loss = self.dc(net_output, target)
        ce_loss = self.ce(net_output, target)
        if self.aggregate == "sum":
            result = ce_loss + dc_loss
        else:
            raise NotImplementedError("nah son") # reserved for other stuff
(later)
        return result


class DC_and_topk_loss(nn.Module):
    def __init__(self, soft_dice_kwargs, ce_kwargs, aggregate="sum",
square_dice=False):
        super(DC_and_topk_loss, self).__init__()
        self.aggregate = aggregate
        self.ce = TopKLoss(**ce_kwargs)
        if not square_dice:
            self.dc = SoftDiceLoss(apply_nonlin=softmax_helper,
**soft_dice_kwargs)
        else:
            self.dc = SoftDiceLossSquared(apply_nonlin=softmax_helper,
**soft_dice_kwargs)

    def forward(self, net_output, target):
        dc_loss = self.dc(net_output, target)
        ce_loss = self.ce(net_output, target)
        if self.aggregate == "sum":
            result = ce_loss + dc_loss
        else:
            raise NotImplementedError("nah son") # reserved for other stuff
(later?)
        return result
```