**cldice.py**

```
from copy import deepcopy
from nnunet.utilities.nd_softmax import softmax_helper
from torch import nn
import torch
import numpy as np
from nnunet.network_architecture.initialization import InitWeights_He
from nnunet.network_architecture.neural_network import SegmentationNetwork
import torch.nn.functional


class ConvDropoutNormNonlin(nn.Module):
    """
    fixes a bug in ConvDropoutNormNonlin where lrelu was used regardless of
nonlin. Bad.
    """

    def __init__(self, input_channels, output_channels,
                 conv_op=nn.Conv2d, conv_kwargs=None,
                 norm_op=nn.BatchNorm2d, norm_op_kwargs=None,
                 dropout_op=nn.Dropout2d, dropout_op_kwargs=None,
                 nonlin=nn.LeakyReLU, nonlin_kwargs=None):
        super(ConvDropoutNormNonlin, self).__init__()
        if nonlin_kwargs is None:
            nonlin_kwargs = {'negative_slope': 1e-2, 'inplace': True}
        if dropout_op_kwargs is None:
            dropout_op_kwargs = {'p': 0.5, 'inplace': True}
        if norm_op_kwargs is None:
            norm_op_kwargs = {'eps': 1e-5, 'affine': True, 'momentum': 0.1}
        if conv_kwargs is None:
            conv_kwargs = {'kernel_size': 3, 'stride': 1, 'padding': 1,
'dilation': 1, 'bias': True}

        self.nonlin_kwargs = nonlin_kwargs
        self.nonlin = nonlin
        self.dropout_op = dropout_op
        self.dropout_op_kwargs = dropout_op_kwargs
        self.norm_op_kwargs = norm_op_kwargs
        self.conv_kwargs = conv_kwargs
        self.conv_op = conv_op
        self.norm_op = norm_op

        self.conv = self.conv_op(input_channels, output_channels,
**self.conv_kwargs)
        if self.dropout_op is not None and self.dropout_op_kwargs['p'] is not
None and self.dropout_op_kwargs[
            'p'] > 0:
            self.dropout = self.dropout_op(**self.dropout_op_kwargs)
        else:
            self.dropout = None
        self.instnorm = self.norm_op(output_channels, **self.norm_op_kwargs)
        self.lrelu = self.nonlin(**self.nonlin_kwargs)

    def forward(self, x):
```

```python
        x = self.conv(x)
        if self.dropout is not None:
            x = self.dropout(x)
        return self.lrelu(self.instnorm(x))


class ConvDropoutNonlinNorm(ConvDropoutNormNonlin):
    def forward(self, x):
        x = self.conv(x)
        if self.dropout is not None:
            x = self.dropout(x)
        return self.instnorm(self.lrelu(x))


class StackedConvLayers(nn.Module):
    def __init__(self, input_feature_channels, output_feature_channels, 
num_convs,
                 conv_op=nn.Conv2d, conv_kwargs=None,
                 norm_op=nn.BatchNorm2d, norm_op_kwargs=None,
                 dropout_op=nn.Dropout2d, dropout_op_kwargs=None,
                 nonlin=nn.LeakyReLU, nonlin_kwargs=None, first_stride=None, 
basic_block=ConvDropoutNormNonlin):
        '''
        stacks ConvDropoutNormLReLU layers. initial_stride will only be applied 
to first layer in the stack. The other parameters affect all layers
        :param input_feature_channels:
        :param output_feature_channels:
        :param num_convs:
        :param dilation:
        :param kernel_size:
        :param padding:
        :param dropout:
        :param initial_stride:
        :param conv_op:
        :param norm_op:
        :param dropout_op:
        :param inplace:
        :param neg_slope:
        :param norm_affine:
        :param conv_bias:
        '''
        self.input_channels = input_feature_channels
        self.output_channels = output_feature_channels

        if nonlin_kwargs is None:
            nonlin_kwargs = {'negative_slope': 1e-2, 'inplace': True}
        if dropout_op_kwargs is None:
            dropout_op_kwargs = {'p': 0.5, 'inplace': True}
        if norm_op_kwargs is None:
            norm_op_kwargs = {'eps': 1e-5, 'affine': True, 'momentum': 0.1}
        if conv_kwargs is None:
            conv_kwargs = {'kernel_size': 3, 'stride': 1, 'padding': 1, 
'dilation': 1, 'bias': True}

        self.nonlin_kwargs = nonlin_kwargs
        self.nonlin = nonlin
        self.dropout_op = dropout_op
        self.dropout_op_kwargs = dropout_op_kwargs
```

```
        self.norm_op_kwargs = norm_op_kwargs
        self.conv_kwargs = conv_kwargs
        self.conv_op = conv_op
        self.norm_op = norm_op

        if first_stride is not None:
            self.conv_kwargs_first_conv = deepcopy(conv_kwargs)
            self.conv_kwargs_first_conv['stride'] = first_stride
        else:
            self.conv_kwargs_first_conv = conv_kwargs

        super(StackedConvLayers, self).__init__()
        self.blocks = nn.Sequential(
            *([basic_block(input_feature_channels, output_feature_channels,
self.conv_op,
                           self.conv_kwargs_first_conv,
                           self.norm_op, self.norm_op_kwargs, self.dropout_op,
self.dropout_op_kwargs,
                           self.nonlin, self.nonlin_kwargs)] +
              [basic_block(output_feature_channels, output_feature_channels,
self.conv_op,
                           self.conv_kwargs,
                           self.norm_op, self.norm_op_kwargs, self.dropout_op,
self.dropout_op_kwargs,
                           self.nonlin, self.nonlin_kwargs) for _ in
range(num_convs - 1)]))

    def forward(self, x):
        return self.blocks(x)


def print_module_training_status(module):
    if isinstance(module, nn.Conv2d) or isinstance(module, nn.Conv3d) or
isinstance(module, nn.Dropout3d) or \
            isinstance(module, nn.Dropout2d) or isinstance(module, nn.Dropout)
or isinstance(module, nn.InstanceNorm3d) \
            or isinstance(module, nn.InstanceNorm2d) or isinstance(module,
nn.InstanceNorm1d) \
            or isinstance(module, nn.BatchNorm2d) or isinstance(module,
nn.BatchNorm3d) or isinstance(module,

                     nn.BatchNorm1d):
        print(str(module), module.training)


class Upsample(nn.Module):
    def __init__(self, size=None, scale_factor=None, mode='nearest',
align_corners=False):
        super(Upsample, self).__init__()
        self.align_corners = align_corners
        self.mode = mode
        self.scale_factor = scale_factor
        self.size = size

    def forward(self, x):
        return nn.functional.interpolate(x, size=self.size,
scale_factor=self.scale_factor, mode=self.mode,
                                         align_corners=self.align_corners)
```

```python
class Generic_UNet(SegmentationNetwork):
    DEFAULT_BATCH_SIZE_3D = 2
    DEFAULT_PATCH_SIZE_3D = (64, 192, 160)
    SPACING_FACTOR_BETWEEN_STAGES = 2
    BASE_NUM_FEATURES_3D = 30
    MAX_NUMPOOL_3D = 999
    MAX_NUM_FILTERS_3D = 320

    DEFAULT_PATCH_SIZE_2D = (256, 256)
    BASE_NUM_FEATURES_2D = 30
    DEFAULT_BATCH_SIZE_2D = 50
    MAX_NUMPOOL_2D = 999
    MAX_FILTERS_2D = 480

    use_this_for_batch_size_computation_2D = 19739648
    use_this_for_batch_size_computation_3D = 520000000  # 505789440

    def __init__(self, input_channels, base_num_features, num_classes, num_pool,
num_conv_per_stage=2,
                 feat_map_mul_on_downscale=2, conv_op=nn.Conv2d,
                 norm_op=nn.BatchNorm2d, norm_op_kwargs=None,
                 dropout_op=nn.Dropout2d, dropout_op_kwargs=None,
                 nonlin=nn.LeakyReLU, nonlin_kwargs=None, deep_supervision=True,
dropout_in_localization=False,
                 final_nonlin=softmax_helper,
weightInitializer=InitWeights_He(1e-2), pool_op_kernel_sizes=None,
                 conv_kernel_sizes=None,
                 upscale_logits=False, convolutional_pooling=False,
convolutional_upsampling=False,
                 max_num_features=None, basic_block=ConvDropoutNormNonlin,
                 seg_output_use_bias=False):
        """
        basically more flexible than v1, architecture is the same

        Does this look complicated? Nah bro. Functionality > usability

        This does everything you need, including world peace.

        Questions? -> f.isensee@dkfz.de
        """
        super(Generic_UNet, self).__init__()
        self.convolutional_upsampling = convolutional_upsampling
        self.convolutional_pooling = convolutional_pooling
        self.upscale_logits = upscale_logits
        if nonlin_kwargs is None:
            nonlin_kwargs = {'negative_slope': 1e-2, 'inplace': True}
        if dropout_op_kwargs is None:
            dropout_op_kwargs = {'p': 0.5, 'inplace': True}
        if norm_op_kwargs is None:
            norm_op_kwargs = {'eps': 1e-5, 'affine': True, 'momentum': 0.1}

        self.conv_kwargs = {'stride': 1, 'dilation': 1, 'bias': True}

        self.nonlin = nonlin
        self.nonlin_kwargs = nonlin_kwargs
        self.dropout_op_kwargs = dropout_op_kwargs
```

```
        self.norm_op_kwargs = norm_op_kwargs
        self.weightInitializer = weightInitializer
        self.conv_op = conv_op
        self.norm_op = norm_op
        self.dropout_op = dropout_op
        self.num_classes = num_classes
        self.final_nonlin = final_nonlin
        self._deep_supervision = deep_supervision
        self.do_ds = deep_supervision

        if conv_op == nn.Conv2d:
            upsample_mode = 'bilinear'
            pool_op = nn.MaxPool2d
            transpconv = nn.ConvTranspose2d
            if pool_op_kernel_sizes is None:
                pool_op_kernel_sizes = [(2, 2)] * num_pool
            if conv_kernel_sizes is None:
                conv_kernel_sizes = [(3, 3)] * (num_pool + 1)
        elif conv_op == nn.Conv3d:
            upsample_mode = 'trilinear'
            pool_op = nn.MaxPool3d
            transpconv = nn.ConvTranspose3d
            if pool_op_kernel_sizes is None:
                pool_op_kernel_sizes = [(2, 2, 2)] * num_pool
            if conv_kernel_sizes is None:
                conv_kernel_sizes = [(3, 3, 3)] * (num_pool + 1)
        else:
            raise ValueError("unknown convolution dimensionality, conv op: %s" %
str(conv_op))

        self.input_shape_must_be_divisible_by = np.prod(pool_op_kernel_sizes, 0,
dtype=np.int64)
        self.pool_op_kernel_sizes = pool_op_kernel_sizes
        self.conv_kernel_sizes = conv_kernel_sizes

        self.conv_pad_sizes = []
        for krnl in self.conv_kernel_sizes:
            self.conv_pad_sizes.append([1 if i == 3 else 0 for i in krnl])

        if max_num_features is None:
            if self.conv_op == nn.Conv3d:
                self.max_num_features = self.MAX_NUM_FILTERS_3D
            else:
                self.max_num_features = self.MAX_FILTERS_2D
        else:
            self.max_num_features = max_num_features

        self.conv_blocks_context = []
        self.conv_blocks_localization = []
        self.td = []
        self.tu = []
        self.seg_outputs = []

        output_features = base_num_features
        input_features = input_channels

        for d in range(num_pool):
            # determine the first stride
```

```
            if d != 0 and self.convolutional_pooling:
                first_stride = pool_op_kernel_sizes[d - 1]
            else:
                first_stride = None

            self.conv_kwargs['kernel_size'] = self.conv_kernel_sizes[d]
            self.conv_kwargs['padding'] = self.conv_pad_sizes[d]
            # add convolutions
            self.conv_blocks_context.append(StackedConvLayers(input_features,
output_features, num_conv_per_stage,
                                                              self.conv_op,
self.conv_kwargs, self.norm_op,

 self.norm_op_kwargs, self.dropout_op,

 self.dropout_op_kwargs, self.nonlin, self.nonlin_kwargs,
                                                              first_stride,
basic_block=basic_block))
            if not self.convolutional_pooling:
                self.td.append(pool_op(pool_op_kernel_sizes[d]))
            input_features = output_features
            output_features = int(np.round(output_features *
feat_map_mul_on_downscale))

            output_features = min(output_features, self.max_num_features)

        # now the bottleneck.
        # determine the first stride
        if self.convolutional_pooling:
            first_stride = pool_op_kernel_sizes[-1]
        else:
            first_stride = None

        # the output of the last conv must match the number of features from the
skip connection if we are not using
        # convolutional upsampling. If we use convolutional upsampling then the
reduction in feature maps will be
        # done by the transposed conv
        if self.convolutional_upsampling:
            final_num_features = output_features
        else:
            final_num_features = self.conv_blocks_context[-1].output_channels

        self.conv_kwargs['kernel_size'] = self.conv_kernel_sizes[num_pool]
        self.conv_kwargs['padding'] = self.conv_pad_sizes[num_pool]
        self.conv_blocks_context.append(nn.Sequential(
            StackedConvLayers(input_features, output_features,
num_conv_per_stage - 1, self.conv_op, self.conv_kwargs,
                              self.norm_op, self.norm_op_kwargs,
self.dropout_op, self.dropout_op_kwargs, self.nonlin,
                              self.nonlin_kwargs, first_stride,
basic_block=basic_block),
            StackedConvLayers(output_features, final_num_features, 1,
self.conv_op, self.conv_kwargs,
                              self.norm_op, self.norm_op_kwargs,
self.dropout_op, self.dropout_op_kwargs, self.nonlin,
                              self.nonlin_kwargs, basic_block=basic_block)))
```

```
        # if we don't want to do dropout in the localization pathway then we set
the dropout prob to zero here
        if not dropout_in_localization:
            old_dropout_p = self.dropout_op_kwargs['p']
            self.dropout_op_kwargs['p'] = 0.0

        # now lets build the localization pathway
        for u in range(num_pool):
            nfeatures_from_down = final_num_features
            nfeatures_from_skip = self.conv_blocks_context[
                -(2 + u)].output_channels  # self.conv_blocks_context[-1] is
bottleneck, so start with -2
            n_features_after_tu_and_concat = nfeatures_from_skip * 2

            # the first conv reduces the number of features to match those of
skip
            # the following convs work on that number of features
            # if not convolutional upsampling then the final conv reduces the
num of features again
            if u != num_pool - 1 and not self.convolutional_upsampling:
                final_num_features = self.conv_blocks_context[-(3 +
u)].output_channels
            else:
                final_num_features = nfeatures_from_skip

            if not self.convolutional_upsampling:
                self.tu.append(Upsample(scale_factor=pool_op_kernel_sizes[-(u +
1)], mode=upsample_mode))
            else:
                self.tu.append(transpconv(nfeatures_from_down,
nfeatures_from_skip, pool_op_kernel_sizes[-(u + 1)],
                                          pool_op_kernel_sizes[-(u + 1)],
bias=False))

            self.conv_kwargs['kernel_size'] = self.conv_kernel_sizes[- (u + 1)]
            self.conv_kwargs['padding'] = self.conv_pad_sizes[- (u + 1)]
            self.conv_blocks_localization.append(nn.Sequential(
                StackedConvLayers(n_features_after_tu_and_concat,
nfeatures_from_skip, num_conv_per_stage - 1,
                                  self.conv_op, self.conv_kwargs, self.norm_op,
self.norm_op_kwargs, self.dropout_op,
                                  self.dropout_op_kwargs, self.nonlin,
self.nonlin_kwargs, basic_block=basic_block),
                StackedConvLayers(nfeatures_from_skip, final_num_features, 1,
self.conv_op, self.conv_kwargs,
                                  self.norm_op, self.norm_op_kwargs,
self.dropout_op, self.dropout_op_kwargs,
                                  self.nonlin, self.nonlin_kwargs,
basic_block=basic_block)
            ))

        for ds in range(len(self.conv_blocks_localization)):
            self.seg_outputs.append(conv_op(self.conv_blocks_localization[ds]
[-1].output_channels, num_classes,
                                            1, 1, 0, 1, 1, seg_output_use_bias))

        self.upscale_logits_ops = []
        cum_upsample = np.cumprod(np.vstack(pool_op_kernel_sizes), axis=0)[::-1]
```

```
        for usl in range(num_pool - 1):
            if self.upscale_logits:

 self.upscale_logits_ops.append(Upsample(scale_factor=tuple([int(i) for i in
cum_upsample[usl + 1]]),
                                                        mode=upsample_mode))
            else:
                self.upscale_logits_ops.append(lambda x: x)

        if not dropout_in_localization:
            self.dropout_op_kwargs['p'] = old_dropout_p

        # register all modules properly
        self.conv_blocks_localization =
nn.ModuleList(self.conv_blocks_localization)
        self.conv_blocks_context = nn.ModuleList(self.conv_blocks_context)
        self.td = nn.ModuleList(self.td)
        self.tu = nn.ModuleList(self.tu)
        self.seg_outputs = nn.ModuleList(self.seg_outputs)
        if self.upscale_logits:
            self.upscale_logits_ops = nn.ModuleList(
                self.upscale_logits_ops)  # lambda x:x is not a Module so we
need to distinguish here

        if self.weightInitializer is not None:
            self.apply(self.weightInitializer)
            # self.apply(print_module_training_status)

    def forward(self, x):
        skips = []
        seg_outputs = []
        for d in range(len(self.conv_blocks_context) - 1):
            x = self.conv_blocks_context[d](x)
            skips.append(x)
            if not self.convolutional_pooling:
                x = self.td[d](x)

        x = self.conv_blocks_context[-1](x)

        for u in range(len(self.tu)):
            x = self.tu[u](x)
            x = torch.cat((x, skips[-(u + 1)]), dim=1)
            x = self.conv_blocks_localization[u](x)
            seg_outputs.append(self.final_nonlin(self.seg_outputs[u](x)))

        if self._deep_supervision and self.do_ds:
            return tuple([seg_outputs[-1]] + [i(j) for i, j in
                                              zip(list(self.upscale_logits_ops)
[::-1], seg_outputs[:-1][::-1])])
        else:
            return seg_outputs[-1]

    @staticmethod
    def compute_approx_vram_consumption(patch_size, num_pool_per_axis,
base_num_features, max_num_features,
                                        num_modalities, num_classes,
pool_op_kernel_sizes, deep_supervision=False,
                                        conv_per_stage=2):
```

```
        """
        This only applies for num_conv_per_stage and
convolutional_upsampling=True
        not real vram consumption. just a constant term to which the vram
consumption will be approx proportional
        (+ offset for parameter storage)
        :param deep_supervision:
        :param patch_size:
        :param num_pool_per_axis:
        :param base_num_features:
        :param max_num_features:
        :param num_modalities:
        :param num_classes:
        :param pool_op_kernel_sizes:
        :return:
        """
        if not isinstance(num_pool_per_axis, np.ndarray):
            num_pool_per_axis = np.array(num_pool_per_axis)

        npool = len(pool_op_kernel_sizes)

        map_size = np.array(patch_size)
        tmp = np.int64((conv_per_stage * 2 + 1) * np.prod(map_size,
dtype=np.int64) * base_num_features +
                       num_modalities * np.prod(map_size, dtype=np.int64) +
                       num_classes * np.prod(map_size, dtype=np.int64))

        num_feat = base_num_features

        for p in range(npool):
            for pi in range(len(num_pool_per_axis)):
                map_size[pi] /= pool_op_kernel_sizes[p][pi]
            num_feat = min(num_feat * 2, max_num_features)
            num_blocks = (conv_per_stage * 2 + 1) if p < (npool - 1) else
conv_per_stage  # conv_per_stage + conv_per_stage for the convs of encode/decode
and 1 for transposed conv
            tmp += num_blocks * np.prod(map_size, dtype=np.int64) * num_feat
            if deep_supervision and p < (npool - 2):
                tmp += np.prod(map_size, dtype=np.int64) * num_classes
            # print(p, map_size, num_feat, tmp)
        return tmp
```