neural_network.py

```
import numpy as np
from batchgenerators.augmentations.utils import pad_nd_image
from nnunet.utilities.random_stuff import no_op
from nnunet.utilities.to_torch import to_cuda, maybe_to_torch
from torch import nn
import torch
from scipy.ndimage.filters import gaussian_filter
from typing import Union, Tuple, List

from torch.cuda.amp import autocast


class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()

    def get_device(self):
        if next(self.parameters()).device.type == "cpu":
            return "cpu"
        else:
            return next(self.parameters()).device.index

    def set_device(self, device):
        if device == "cpu":
            self.cpu()
        else:
            self.cuda(device)

    def forward(self, x):
        raise NotImplementedError


class SegmentationNetwork(NeuralNetwork):
    def __init__(self):
        super(NeuralNetwork, self).__init__()

        # if we have 5 pooling then our patch size must be divisible by 2**5
        self.input_shape_must_be_divisible_by = None  # for example in a 2d
network that does 5 pool in x and 6 pool
        # in y this would be (32, 64)

        # we need to know this because we need to know if we are a 2d or a 3d
netowrk
        self.conv_op = None  # nn.Conv2d or nn.Conv3d

        # this tells us how many channels we have in the output. Important for
preallocation in inference
        self.num_classes = None  # number of channels in the output

        # depending on the loss, we do not hard code a nonlinearity into the
architecture. To aggregate predictions
        # during inference, we need to apply the nonlinearity, however. So it is
important to let the newtork know what
```

```
        # to apply in inference. For the most part this will be softmax
        self.inference_apply_nonlin = lambda x: x  # softmax_helper

        # This is for saving a gaussian importance map for inference. It weights 
voxels higher that are closer to the
        # center. Prediction at the borders are often less accurate and are thus 
downweighted. Creating these Gaussians
        # can be expensive, so it makes sense to save and reuse them.
        self._gaussian_3d = self._patch_size_for_gaussian_3d = None
        self._gaussian_2d = self._patch_size_for_gaussian_2d = None

    def predict_3D(self, x: np.ndarray, do_mirroring: bool, mirror_axes: 
Tuple[int, ...] = (0, 1, 2),
                   use_sliding_window: bool = False,
                   step_size: float = 0.5, patch_size: Tuple[int, ...] = None, 
regions_class_order: Tuple[int, ...] = None,
                   use_gaussian: bool = False, pad_border_mode: str = 
"constant",
                   pad_kwargs: dict = None, all_in_gpu: bool = False,
                   verbose: bool = True, mixed_precision: bool = True) -> 
Tuple[np.ndarray, np.ndarray]:
        """
        Use this function to predict a 3D image. It does not matter whether the 
network is a 2D or 3D U-Net, it will
        detect that automatically and run the appropriate code.

        When running predictions, you need to specify whether you want to run 
fully convolutional of sliding window
        based inference. We very strongly recommend you use sliding window with 
the default settings.

        It is the responsibility of the user to make sure the network is in the 
proper mode (eval for inference!). If
        the network is not in eval mode it will print a warning.

        :param x: Your input data. Must be a nd.ndarray of shape (c, x, y, z).
        :param do_mirroring: If True, use test time data augmentation in the form 
of mirroring
        :param mirror_axes: Determines which axes to use for mirroing. Per 
default, mirroring is done along all three
        axes
        :param use_sliding_window: if True, run sliding window prediction. 
Heavily recommended! This is also the default
        :param step_size: When running sliding window prediction, the step size 
determines the distance between adjacent
        predictions. The smaller the step size, the denser the predictions (and 
the longer it takes!). Step size is given
        as a fraction of the patch_size. 0.5 is the default and means that wen 
advance by patch_size * 0.5 between
        predictions. step_size cannot be larger than 1!
        :param patch_size: The patch size that was used for training the network. 
Do not use different patch sizes here,
        this will either crash or give potentially less accurate segmentations
        :param regions_class_order: Fabian only
        :param use_gaussian: (Only applies to sliding window prediction) If True, 
uses a Gaussian importance weighting
         to weigh predictions closer to the center of the current patch higher 
than those at the borders. The reason
```

```
         behind this is that the segmentation accuracy decreases towards the
borders. Default (and recommended): True
        :param pad_border_mode: leave this alone
        :param pad_kwargs: leave this alone
        :param all_in_gpu: experimental. You probably want to leave this as is
it
        :param verbose: Do you want a wall of text? If yes then set this to True
        :param mixed_precision: if True, will run inference in mixed precision
with autocast()
        :return:
        """
        torch.cuda.empty_cache()

        assert step_size <= 1, 'step_size must be smaller than 1. Otherwise
there will be a gap between consecutive ' \
                               'predictions'

        if verbose: print("debug: mirroring", do_mirroring, "mirror_axes",
mirror_axes)

        if pad_kwargs is None:
            pad_kwargs = {'constant_values': 0}

        # A very long time ago the mirror axes were (2, 3, 4) for a 3d network.
This is just to intercept any old
        # code that uses this convention
        if len(mirror_axes):
            if self.conv_op == nn.Conv2d:
                if max(mirror_axes) > 1:
                    raise ValueError("mirror axes. duh")
            if self.conv_op == nn.Conv3d:
                if max(mirror_axes) > 2:
                    raise ValueError("mirror axes. duh")

        if self.training:
            print('WARNING! Network is in train mode during inference. This may
be intended, or not...')

        assert len(x.shape) == 4, "data must have shape (c,x,y,z)"

        if mixed_precision:
            context = autocast
        else:
            context = no_op

        with context():
            with torch.no_grad():
                if self.conv_op == nn.Conv3d:
                    if use_sliding_window:
                        res = self._internal_predict_3D_3Dconv_tiled(x,
step_size, do_mirroring, mirror_axes, patch_size,

regions_class_order, use_gaussian, pad_border_mode,

pad_kwargs=pad_kwargs, all_in_gpu=all_in_gpu,

verbose=verbose)
                    else:
                        res = self._internal_predict_3D_3Dconv(x, patch_size,
do_mirroring, mirror_axes, regions_class_order,
                                                                               pad_border_mode,
pad_kwargs=pad_kwargs, verbose=verbose)
                elif self.conv_op == nn.Conv2d:
                    if use_sliding_window:
                        res = self._internal_predict_3D_2Dconv_tiled(x,
patch_size, do_mirroring, mirror_axes, step_size,

regions_class_order, use_gaussian, pad_border_mode,
                                                                                     pad_kwargs,
all_in_gpu, False)
                    else:
                        res = self._internal_predict_3D_2Dconv(x, patch_size,
do_mirroring, mirror_axes, regions_class_order,
                                                                               pad_border_mode,
pad_kwargs, all_in_gpu, False)
                else:
                    raise RuntimeError("Invalid conv op, cannot determine what
dimensionality (2d/3d) the network is")

        return res

    def predict_2D(self, x, do_mirroring: bool, mirror_axes: tuple = (0, 1, 2),
use_sliding_window: bool = False,
                   step_size: float = 0.5, patch_size: tuple = None,
regions_class_order: tuple = None,
                   use_gaussian: bool = False, pad_border_mode: str =
"constant",
                   pad_kwargs: dict = None, all_in_gpu: bool = False,
                   verbose: bool = True, mixed_precision: bool = True) ->
Tuple[np.ndarray, np.ndarray]:
        """
        Use this function to predict a 2D image. If this is a 3D U-Net it will
crash because you cannot predict a 2D
        image with that (you dummy).

        When running predictions, you need to specify whether you want to run
fully convolutional of sliding window
        based inference. We very strongly recommend you use sliding window with
the default settings.

        It is the responsibility of the user to make sure the network is in the
proper mode (eval for inference!). If
        the network is not in eval mode it will print a warning.

        :param x: Your input data. Must be a nd.ndarray of shape (c, x, y).
        :param do_mirroring: If True, use test time data augmentation in the form
of mirroring
        :param mirror_axes: Determines which axes to use for mirroing. Per
default, mirroring is done along all three
        axes
        :param use_sliding_window: if True, run sliding window prediction.
Heavily recommended! This is also the default
        :param step_size: When running sliding window prediction, the step size
determines the distance between adjacent
        predictions. The smaller the step size, the denser the predictions (and
the longer it takes!). Step size is given
```

```
        as a fraction of the patch_size. 0.5 is the default and means that wen
advance by patch_size * 0.5 between
        predictions. step_size cannot be larger than 1!
        :param patch_size: The patch size that was used for training the network.
Do not use different patch sizes here,
        this will either crash or give potentially less accurate segmentations
        :param regions_class_order: Fabian only
        :param use_gaussian: (Only applies to sliding window prediction) If True,
uses a Gaussian importance weighting
         to weigh predictions closer to the center of the current patch higher
than those at the borders. The reason
         behind this is that the segmentation accuracy decreases towards the
borders. Default (and recommended): True
        :param pad_border_mode: leave this alone
        :param pad_kwargs: leave this alone
        :param all_in_gpu: experimental. You probably want to leave this as is
it
        :param verbose: Do you want a wall of text? If yes then set this to True
        :return:
        """
        torch.cuda.empty_cache()

        assert step_size <= 1, 'step_size must be smaler than 1. Otherwise there
will be a gap between consecutive ' \
                               'predictions'

        if self.conv_op == nn.Conv3d:
            raise RuntimeError("Cannot predict 2d if the network is 3d. Dummy.")

        if verbose: print("debug: mirroring", do_mirroring, "mirror_axes",
mirror_axes)

        if pad_kwargs is None:
            pad_kwargs = {'constant_values': 0}

        # A very long time ago the mirror axes were (2, 3) for a 2d network.
This is just to intercept any old
        # code that uses this convention
        if len(mirror_axes):
            if max(mirror_axes) > 1:
                raise ValueError("mirror axes. duh")

        if self.training:
            print('WARNING! Network is in train mode during inference. This may
be intended, or not...')

        assert len(x.shape) == 3, "data must have shape (c,x,y)"

        if mixed_precision:
            context = autocast
        else:
            context = no_op

        with context():
            with torch.no_grad():
                if self.conv_op == nn.Conv2d:
                    if use_sliding_window:
                        res = self._internal_predict_2D_2Dconv_tiled(x,
step_size, do_mirroring, mirror_axes, patch_size,

regions_class_order, use_gaussian, pad_border_mode,
                                                                   pad_kwargs,
all_in_gpu, verbose)
                    else:
                        res = self._internal_predict_2D_2Dconv(x, patch_size,
do_mirroring, mirror_axes, regions_class_order,
                                                             pad_border_mode,
pad_kwargs, verbose)
                else:
                    raise RuntimeError("Invalid conv op, cannot determine what
dimensionality (2d/3d) the network is")

        return res

    @staticmethod
    def _get_gaussian(patch_size, sigma_scale=1. / 8) -> np.ndarray:
        tmp = np.zeros(patch_size)
        center_coords = [i // 2 for i in patch_size]
        sigmas = [i * sigma_scale for i in patch_size]
        tmp[tuple(center_coords)] = 1
        gaussian_importance_map = gaussian_filter(tmp, sigmas, 0,
mode='constant', cval=0)
        gaussian_importance_map = gaussian_importance_map /
np.max(gaussian_importance_map) * 1
        gaussian_importance_map = gaussian_importance_map.astype(np.float32)

        # gaussian_importance_map cannot be 0, otherwise we may end up with
nans!
        gaussian_importance_map[gaussian_importance_map == 0] = np.min(
            gaussian_importance_map[gaussian_importance_map != 0])

        return gaussian_importance_map

    @staticmethod
    def _compute_steps_for_sliding_window(patch_size: Tuple[int, ...],
image_size: Tuple[int, ...], step_size: float) -> List[List[int]]:
        assert [i >= j for i, j in zip(image_size, patch_size)], "image size
must be as large or larger than patch_size"
        assert 0 < step_size <= 1, 'step_size must be larger than 0 and smaller
or equal to 1'

        # our step width is patch_size*step_size at most, but can be narrower.
For example if we have image size of
        # 110, patch size of 64 and step_size of 0.5, then we want to make 3
steps starting at coordinate 0, 23, 46
        target_step_sizes_in_voxels = [i * step_size for i in patch_size]

        num_steps = [int(np.ceil((i - k) / j)) + 1 for i, j, k in
zip(image_size, target_step_sizes_in_voxels, patch_size)]

        steps = []
        for dim in range(len(patch_size)):
            # the highest step value for this dimension is
            max_step_value = image_size[dim] - patch_size[dim]
            if num_steps[dim] > 1:
                actual_step_size = max_step_value / (num_steps[dim] - 1)
            else:
                actual_step_size = 99999999999  # does not matter because there
is only one step at 0

            steps_here = [int(np.round(actual_step_size * i)) for i in
range(num_steps[dim])]

            steps.append(steps_here)

        return steps

    def _internal_predict_3D_3Dconv_tiled(self, x: np.ndarray, step_size: float,
do_mirroring: bool, mirror_axes: tuple,
                                          patch_size: tuple,
regions_class_order: tuple, use_gaussian: bool,
                                          pad_border_mode: str, pad_kwargs:
dict, all_in_gpu: bool,
                                          verbose: bool) -> Tuple[np.ndarray,
np.ndarray]:
        # better safe than sorry
        assert len(x.shape) == 4, "x must be (c, x, y, z)"

        if verbose: print("step_size:", step_size)
        if verbose: print("do mirror:", do_mirroring)

        assert patch_size is not None, "patch_size cannot be None for tiled
prediction"

        # for sliding window inference the image must at least be as large as
the patch size. It does not matter
        # whether the shape is divisible by 2**num_pool as long as the patch
size is
        data, slicer = pad_nd_image(x, patch_size, pad_border_mode, pad_kwargs,
True, None)
        data_shape = data.shape  # still c, x, y, z

        # compute the steps for sliding window
        steps = self._compute_steps_for_sliding_window(patch_size,
data_shape[1:], step_size)
        num_tiles = len(steps[0]) * len(steps[1]) * len(steps[2])

        if verbose:
            print("data shape:", data_shape)
            print("patch size:", patch_size)
            print("steps (x, y, and z):", steps)
            print("number of tiles:", num_tiles)

        # we only need to compute that once. It can take a while to compute this
due to the large sigma in
        # gaussian_filter
        if use_gaussian and num_tiles > 1:
            if self._gaussian_3d is None or not all(
                    [i == j for i, j in zip(patch_size,
self._patch_size_for_gaussian_3d)]):
                if verbose: print('computing Gaussian')
                gaussian_importance_map = self._get_gaussian(patch_size,
sigma_scale=1. / 8)
```

```
                self._gaussian_3d = gaussian_importance_map
                self._patch_size_for_gaussian_3d = patch_size
            else:
                if verbose: print("using precomputed Gaussian")
                gaussian_importance_map = self._gaussian_3d

            gaussian_importance_map = torch.from_numpy(gaussian_importance_map)

            #predict on cpu if cuda not available
            if torch.cuda.is_available():
                gaussian_importance_map =
gaussian_importance_map.cuda(self.get_device(), non_blocking=True)

        else:
            gaussian_importance_map = None

        if all_in_gpu:
            # If we run the inference in GPU only (meaning all tensors are
allocated on the GPU, this reduces
            # CPU-GPU communication but required more GPU memory) we need to
preallocate a few things on GPU

            if use_gaussian and num_tiles > 1:
                # half precision for the outputs should be good enough. If the
outputs here are half, the
                # gaussian_importance_map should be as well
                gaussian_importance_map = gaussian_importance_map.half()

                # make sure we did not round anything to 0
                gaussian_importance_map[gaussian_importance_map == 0] =
gaussian_importance_map[
                    gaussian_importance_map != 0].min()

                add_for_nb_of_preds = gaussian_importance_map
            else:
                add_for_nb_of_preds = torch.ones(data.shape[1:],
device=self.get_device())

            if verbose: print("initializing result array (on GPU)")
            aggregated_results = torch.zeros([self.num_classes] +
list(data.shape[1:]), dtype=torch.half,
                                             device=self.get_device())

            if verbose: print("moving data to GPU")
            data = torch.from_numpy(data).cuda(self.get_device(),
non_blocking=True)

            if verbose: print("initializing result_numsamples (on GPU)")
            aggregated_nb_of_predictions = torch.zeros([self.num_classes] +
list(data.shape[1:]), dtype=torch.half,
                                                       device=self.get_device())

        else:
            if use_gaussian and num_tiles > 1:
                add_for_nb_of_preds = self._gaussian_3d
            else:
                add_for_nb_of_preds = np.ones(data.shape[1:], dtype=np.float32)
            aggregated_results = np.zeros([self.num_classes] +
list(data.shape[1:]), dtype=np.float32)
            aggregated_nb_of_predictions = np.zeros([self.num_classes] +
list(data.shape[1:]), dtype=np.float32)

        for x in steps[0]:
            lb_x = x
            ub_x = x + patch_size[0]
            for y in steps[1]:
                lb_y = y
                ub_y = y + patch_size[1]
                for z in steps[2]:
                    lb_z = z
                    ub_z = z + patch_size[2]

                    predicted_patch = self._internal_maybe_mirror_and_pred_3D(
                        data[None, :, lb_x:ub_x, lb_y:ub_y, lb_z:ub_z],
mirror_axes, do_mirroring,
                        gaussian_importance_map)[0]

                    if all_in_gpu:
                        predicted_patch = predicted_patch.half()
                    else:
                        predicted_patch = predicted_patch.cpu().numpy()

                    aggregated_results[:, lb_x:ub_x, lb_y:ub_y, lb_z:ub_z] +=
predicted_patch
                    aggregated_nb_of_predictions[:, lb_x:ub_x, lb_y:ub_y,
lb_z:ub_z] += add_for_nb_of_preds

        # we reverse the padding here (remeber that we padded the input to be at
least as large as the patch size
        slicer = tuple(
            [slice(0, aggregated_results.shape[i]) for i in
             range(len(aggregated_results.shape) - (len(slicer) - 1))] +
slicer[1:])
        aggregated_results = aggregated_results[slicer]
        aggregated_nb_of_predictions = aggregated_nb_of_predictions[slicer]

        # computing the class_probabilities by dividing the aggregated result
with result_numsamples
        class_probabilities = aggregated_results / aggregated_nb_of_predictions

        if regions_class_order is None:
            predicted_segmentation = class_probabilities.argmax(0)
        else:
            if all_in_gpu:
                class_probabilities_here =
class_probabilities.detach().cpu().numpy()
            else:
                class_probabilities_here = class_probabilities
            predicted_segmentation =
np.zeros(class_probabilities_here.shape[1:], dtype=np.float32)
            for i, c in enumerate(regions_class_order):
                predicted_segmentation[class_probabilities_here[i] > 0.5] = c

        if all_in_gpu:
            if verbose: print("copying results to CPU")
            if regions_class_order is None:
                predicted_segmentation =
predicted_segmentation.detach().cpu().numpy()

            class_probabilities = class_probabilities.detach().cpu().numpy()

        if verbose: print("prediction done")
        return predicted_segmentation, class_probabilities

    def _internal_predict_2D_2Dconv(self, x: np.ndarray, min_size: Tuple[int,
int], do_mirroring: bool,
                                    mirror_axes: tuple = (0, 1, 2),
regions_class_order: tuple = None,
                                    pad_border_mode: str = "constant",
pad_kwargs: dict = None,
                                    verbose: bool = True) -> Tuple[np.ndarray,
np.ndarray]:
        """
        This one does fully convolutional inference. No sliding window
        """
        assert len(x.shape) == 3, "x must be (c, x, y)"

        assert self.input_shape_must_be_divisible_by is not None,
'input_shape_must_be_divisible_by must be set to ' \
                                                                  'run
_internal_predict_2D_2Dconv'
        if verbose: print("do mirror:", do_mirroring)

        data, slicer = pad_nd_image(x, min_size, pad_border_mode, pad_kwargs,
True,
                                    self.input_shape_must_be_divisible_by)

        predicted_probabilities =
self._internal_maybe_mirror_and_pred_2D(data[None], mirror_axes, do_mirroring,
                                                                          None)
[0]

        slicer = tuple(
            [slice(0, predicted_probabilities.shape[i]) for i in
range(len(predicted_probabilities.shape) -

(len(slicer) - 1))] + slicer[1:])
        predicted_probabilities = predicted_probabilities[slicer]

        if regions_class_order is None:
            predicted_segmentation = predicted_probabilities.argmax(0)
            predicted_segmentation =
predicted_segmentation.detach().cpu().numpy()
            predicted_probabilities =
predicted_probabilities.detach().cpu().numpy()
        else:
            predicted_probabilities =
predicted_probabilities.detach().cpu().numpy()
            predicted_segmentation = np.zeros(predicted_probabilities.shape[1:],
dtype=np.float32)
            for i, c in enumerate(regions_class_order):
                predicted_segmentation[predicted_probabilities[i] > 0.5] = c
        return predicted_segmentation, predicted_probabilities

    def _internal_predict_3D_3Dconv(self, x: np.ndarray, min_size: Tuple[int,
...], do_mirroring: bool,
                                    mirror_axes: tuple = (0, 1, 2),
regions_class_order: tuple = None,
                                    pad_border_mode: str = "constant",
pad_kwargs: dict = None,
                                    verbose: bool = True) -> Tuple[np.ndarray,
np.ndarray]:
        """
        This one does fully convolutional inference. No sliding window
        """
        assert len(x.shape) == 4, "x must be (c, x, y, z)"

        assert self.input_shape_must_be_divisible_by is not None,
'input_shape_must_be_divisible_by must be set to ' \
                                                                  'run
_internal_predict_3D_3Dconv'
        if verbose: print("do mirror:", do_mirroring)

        data, slicer = pad_nd_image(x, min_size, pad_border_mode, pad_kwargs,
True,
                                    self.input_shape_must_be_divisible_by)

        predicted_probabilities =
self._internal_maybe_mirror_and_pred_3D(data[None], mirror_axes, do_mirroring,
                                                                          None)
[0]

        slicer = tuple(
            [slice(0, predicted_probabilities.shape[i]) for i in
range(len(predicted_probabilities.shape) -

(len(slicer) - 1))] + slicer[1:])
        predicted_probabilities = predicted_probabilities[slicer]

        if regions_class_order is None:
            predicted_segmentation = predicted_probabilities.argmax(0)
            predicted_segmentation =
predicted_segmentation.detach().cpu().numpy()
            predicted_probabilities =
predicted_probabilities.detach().cpu().numpy()
        else:
            predicted_probabilities =
predicted_probabilities.detach().cpu().numpy()
            predicted_segmentation = np.zeros(predicted_probabilities.shape[1:],
dtype=np.float32)
            for i, c in enumerate(regions_class_order):
                predicted_segmentation[predicted_probabilities[i] > 0.5] = c

        return predicted_segmentation, predicted_probabilities

    def _internal_maybe_mirror_and_pred_3D(self, x: Union[np.ndarray,
torch.tensor], mirror_axes: tuple,
                                           do_mirroring: bool = True,
                                                        mult: np.ndarray or torch.tensor =
None) -> torch.tensor:
        assert len(x.shape) == 5, 'x must be (b, c, x, y, z)'

        # if cuda available:
        #   everything in here takes place on the GPU. If x and mult are not yet
on GPU this will be taken care of here
        #   we now return a cuda tensor! Not numpy array!

        x = maybe_to_torch(x)
        result_torch = torch.zeros([1, self.num_classes] + list(x.shape[2:]),
                                   dtype=torch.float)

        if torch.cuda.is_available():
            x = to_cuda(x, gpu_id=self.get_device())
            result_torch = result_torch.cuda(self.get_device(),
non_blocking=True)

        if mult is not None:
            mult = maybe_to_torch(mult)
            if torch.cuda.is_available():
                mult = to_cuda(mult, gpu_id=self.get_device())

        if do_mirroring:
            mirror_idx = 8
            num_results = 2 ** len(mirror_axes)
        else:
            mirror_idx = 1
            num_results = 1

        for m in range(mirror_idx):
            if m == 0:
                pred = self.inference_apply_nonlin(self(x))
                result_torch += 1 / num_results * pred

            if m == 1 and (2 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (4, ))))
                result_torch += 1 / num_results * torch.flip(pred, (4,))

            if m == 2 and (1 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (3, ))))
                result_torch += 1 / num_results * torch.flip(pred, (3,))

            if m == 3 and (2 in mirror_axes) and (1 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (4, 3))))
                result_torch += 1 / num_results * torch.flip(pred, (4, 3))

            if m == 4 and (0 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (2, ))))
                result_torch += 1 / num_results * torch.flip(pred, (2,))

            if m == 5 and (0 in mirror_axes) and (2 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (4, 2))))
                result_torch += 1 / num_results * torch.flip(pred, (4, 2))

            if m == 6 and (0 in mirror_axes) and (1 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (3, 2))))
                result_torch += 1 / num_results * torch.flip(pred, (3, 2))
            if m == 7 and (0 in mirror_axes) and (1 in mirror_axes) and (2 in
mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (4, 3,
2))))
                result_torch += 1 / num_results * torch.flip(pred, (4, 3, 2))

        if mult is not None:
            result_torch[:, :] *= mult

        return result_torch

    def _internal_maybe_mirror_and_pred_2D(self, x: Union[np.ndarray,
torch.tensor], mirror_axes: tuple,
                                           do_mirroring: bool = True,
                                           mult: np.ndarray or torch.tensor =
None) -> torch.tensor:
        # if cuda available:
        #   everything in here takes place on the GPU. If x and mult are not yet
on GPU this will be taken care of here
        #   we now return a cuda tensor! Not numpy array!

        assert len(x.shape) == 4, 'x must be (b, c, x, y)'

        x = maybe_to_torch(x)
        result_torch = torch.zeros([x.shape[0], self.num_classes] +
list(x.shape[2:]), dtype=torch.float)

        if torch.cuda.is_available():
            x = to_cuda(x, gpu_id=self.get_device())
            result_torch = result_torch.cuda(self.get_device(),
non_blocking=True)

        if mult is not None:
            mult = maybe_to_torch(mult)
            if torch.cuda.is_available():
                mult = to_cuda(mult, gpu_id=self.get_device())

        if do_mirroring:
            mirror_idx = 4
            num_results = 2 ** len(mirror_axes)
        else:
            mirror_idx = 1
            num_results = 1

        for m in range(mirror_idx):
            if m == 0:
                pred = self.inference_apply_nonlin(self(x))
                result_torch += 1 / num_results * pred

            if m == 1 and (1 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (3, ))))
                result_torch += 1 / num_results * torch.flip(pred, (3, ))

            if m == 2 and (0 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (2, ))))
                result_torch += 1 / num_results * torch.flip(pred, (2, ))
            if m == 3 and (0 in mirror_axes) and (1 in mirror_axes):
                pred = self.inference_apply_nonlin(self(torch.flip(x, (3, 2))))
                result_torch += 1 / num_results * torch.flip(pred, (3, 2))

        if mult is not None:
            result_torch[:, :] *= mult

        return result_torch

    def _internal_predict_2D_2Dconv_tiled(self, x: np.ndarray, step_size: float,
do_mirroring: bool, mirror_axes: tuple,
                                          patch_size: tuple,
regions_class_order: tuple, use_gaussian: bool,
                                          pad_border_mode: str, pad_kwargs:
dict, all_in_gpu: bool,
                                          verbose: bool) -> Tuple[np.ndarray,
np.ndarray]:
        # better safe than sorry
        assert len(x.shape) == 3, "x must be (c, x, y)"

        if verbose: print("step_size:", step_size)
        if verbose: print("do mirror:", do_mirroring)

        assert patch_size is not None, "patch_size cannot be None for tiled
prediction"

        # for sliding window inference the image must at least be as large as
the patch size. It does not matter
        # whether the shape is divisible by 2**num_pool as long as the patch
size is
        data, slicer = pad_nd_image(x, patch_size, pad_border_mode, pad_kwargs,
True, None)
        data_shape = data.shape  # still c, x, y

        # compute the steps for sliding window
        steps = self._compute_steps_for_sliding_window(patch_size,
data_shape[1:], step_size)
        num_tiles = len(steps[0]) * len(steps[1])

        if verbose:
            print("data shape:", data_shape)
            print("patch size:", patch_size)
            print("steps (x, y, and z):", steps)
            print("number of tiles:", num_tiles)

        # we only need to compute that once. It can take a while to compute this
due to the large sigma in
        # gaussian_filter
        if use_gaussian and num_tiles > 1:
            if self._gaussian_2d is None or not all(
                    [i == j for i, j in zip(patch_size,
self._patch_size_for_gaussian_2d)]):
                if verbose: print('computing Gaussian')
                gaussian_importance_map = self._get_gaussian(patch_size,
sigma_scale=1. / 8)

                self._gaussian_2d = gaussian_importance_map
                self._patch_size_for_gaussian_2d = patch_size
            else:
                if verbose: print("using precomputed Gaussian")
                gaussian_importance_map = self._gaussian_2d

            gaussian_importance_map = torch.from_numpy(gaussian_importance_map)
            if torch.cuda.is_available():
                gaussian_importance_map = 
gaussian_importance_map.cuda(self.get_device(), non_blocking=True)

        else:
            gaussian_importance_map = None

        if all_in_gpu:
            # If we run the inference in GPU only (meaning all tensors are 
allocated on the GPU, this reduces
            # CPU-GPU communication but required more GPU memory) we need to 
preallocate a few things on GPU

            if use_gaussian and num_tiles > 1:
                # half precision for the outputs should be good enough. If the 
outputs here are half, the
                # gaussian_importance_map should be as well
                gaussian_importance_map = gaussian_importance_map.half()

                # make sure we did not round anything to 0
                gaussian_importance_map[gaussian_importance_map == 0] = 
gaussian_importance_map[
                    gaussian_importance_map != 0].min()

                add_for_nb_of_preds = gaussian_importance_map
            else:
                add_for_nb_of_preds = torch.ones(data.shape[1:], 
device=self.get_device())

            if verbose: print("initializing result array (on GPU)")
            aggregated_results = torch.zeros([self.num_classes] + 
list(data.shape[1:]), dtype=torch.half,
                                             device=self.get_device())

            if verbose: print("moving data to GPU")
            data = torch.from_numpy(data).cuda(self.get_device(), 
non_blocking=True)

            if verbose: print("initializing result_numsamples (on GPU)")
            aggregated_nb_of_predictions = torch.zeros([self.num_classes] + 
list(data.shape[1:]), dtype=torch.half,
                                                       device=self.get_device())
        else:
            if use_gaussian and num_tiles > 1:
                add_for_nb_of_preds = self._gaussian_2d
            else:
                add_for_nb_of_preds = np.ones(data.shape[1:], dtype=np.float32)
            aggregated_results = np.zeros([self.num_classes] + 
list(data.shape[1:]), dtype=np.float32)
            aggregated_nb_of_predictions = np.zeros([self.num_classes] + 
list(data.shape[1:]), dtype=np.float32)

        for x in steps[0]:
            lb_x = x
            ub_x = x + patch_size[0]
            for y in steps[1]:
                lb_y = y
                ub_y = y + patch_size[1]

                predicted_patch = self._internal_maybe_mirror_and_pred_2D(
                    data[None, :, lb_x:ub_x, lb_y:ub_y], mirror_axes,
do_mirroring,
                    gaussian_importance_map)[0]

                if all_in_gpu:
                    predicted_patch = predicted_patch.half()
                else:
                    predicted_patch = predicted_patch.cpu().numpy()

                aggregated_results[:, lb_x:ub_x, lb_y:ub_y] += predicted_patch
                aggregated_nb_of_predictions[:, lb_x:ub_x, lb_y:ub_y] +=
add_for_nb_of_preds

        # we reverse the padding here (remeber that we padded the input to be at
least as large as the patch size
        slicer = tuple(
            [slice(0, aggregated_results.shape[i]) for i in
             range(len(aggregated_results.shape) - (len(slicer) - 1))] +
slicer[1:])
        aggregated_results = aggregated_results[slicer]
        aggregated_nb_of_predictions = aggregated_nb_of_predictions[slicer]

        # computing the class_probabilities by dividing the aggregated result
with result_numsamples
        class_probabilities = aggregated_results / aggregated_nb_of_predictions

        if regions_class_order is None:
            predicted_segmentation = class_probabilities.argmax(0)
        else:
            if all_in_gpu:
                class_probabilities_here =
class_probabilities.detach().cpu().numpy()
            else:
                class_probabilities_here = class_probabilities
            predicted_segmentation =
np.zeros(class_probabilities_here.shape[1:], dtype=np.float32)
            for i, c in enumerate(regions_class_order):
                predicted_segmentation[class_probabilities_here[i] > 0.5] = c

        if all_in_gpu:
            if verbose: print("copying results to CPU")

            if regions_class_order is None:
                predicted_segmentation =
predicted_segmentation.detach().cpu().numpy()

            class_probabilities = class_probabilities.detach().cpu().numpy()

        if verbose: print("prediction done")
        return predicted_segmentation, class_probabilities
```

```
    def _internal_predict_3D_2Dconv(self, x: np.ndarray, min_size: Tuple[int,
int], do_mirroring: bool,
                                    mirror_axes: tuple = (0, 1),
regions_class_order: tuple = None,
                                    pad_border_mode: str = "constant",
pad_kwargs: dict = None,
                                    all_in_gpu: bool = False, verbose: bool =
True) -> Tuple[np.ndarray, np.ndarray]:
        if all_in_gpu:
            raise NotImplementedError
        assert len(x.shape) == 4, "data must be c, x, y, z"
        predicted_segmentation = []
        softmax_pred = []
        for s in range(x.shape[1]):
            pred_seg, softmax_pres = self._internal_predict_2D_2Dconv(
                x[:, s], min_size, do_mirroring, mirror_axes,
regions_class_order, pad_border_mode, pad_kwargs, verbose)
            predicted_segmentation.append(pred_seg[None])
            softmax_pred.append(softmax_pres[None])
        predicted_segmentation = np.vstack(predicted_segmentation)
        softmax_pred = np.vstack(softmax_pred).transpose((1, 0, 2, 3))
        return predicted_segmentation, softmax_pred

    def predict_3D_pseudo3D_2Dconv(self, x: np.ndarray, min_size: Tuple[int,
int], do_mirroring: bool,
                                   mirror_axes: tuple = (0, 1),
regions_class_order: tuple = None,
                                   pseudo3D_slices: int = 5, all_in_gpu: bool =
False,
                                   pad_border_mode: str = "constant",
pad_kwargs: dict = None,
                                   verbose: bool = True) -> Tuple[np.ndarray,
np.ndarray]:
        if all_in_gpu:
            raise NotImplementedError
        assert len(x.shape) == 4, "data must be c, x, y, z"
        assert pseudo3D_slices % 2 == 1, "pseudo3D_slices must be odd"
        extra_slices = (pseudo3D_slices - 1) // 2

        shp_for_pad = np.array(x.shape)
        shp_for_pad[1] = extra_slices

        pad = np.zeros(shp_for_pad, dtype=np.float32)
        data = np.concatenate((pad, x, pad), 1)

        predicted_segmentation = []
        softmax_pred = []
        for s in range(extra_slices, data.shape[1] - extra_slices):
            d = data[:, (s - extra_slices):(s + extra_slices + 1)]
            d = d.reshape((-1, d.shape[-2], d.shape[-1]))
            pred_seg, softmax_pres = \
                self._internal_predict_2D_2Dconv(d, min_size, do_mirroring,
mirror_axes,
                                                 regions_class_order,
pad_border_mode, pad_kwargs, verbose)
            predicted_segmentation.append(pred_seg[None])
            softmax_pred.append(softmax_pres[None])
        predicted_segmentation = np.vstack(predicted_segmentation)
        softmax_pred = np.vstack(softmax_pred).transpose((1, 0, 2, 3))

        return predicted_segmentation, softmax_pred

    def _internal_predict_3D_2Dconv_tiled(self, x: np.ndarray, patch_size:
Tuple[int, int], do_mirroring: bool,
                                          mirror_axes: tuple = (0, 1),
step_size: float = 0.5,
                                          regions_class_order: tuple = None,
use_gaussian: bool = False,
                                          pad_border_mode: str = "edge",
pad_kwargs: dict =None,
                                          all_in_gpu: bool = False,
                                          verbose: bool = True) ->
Tuple[np.ndarray, np.ndarray]:
        if all_in_gpu:
            raise NotImplementedError

        assert len(x.shape) == 4, "data must be c, x, y, z"

        predicted_segmentation = []
        softmax_pred = []

        for s in range(x.shape[1]):
            pred_seg, softmax_pres = self._internal_predict_2D_2Dconv_tiled(
                x[:, s], step_size, do_mirroring, mirror_axes, patch_size,
regions_class_order, use_gaussian,
                pad_border_mode, pad_kwargs, all_in_gpu, verbose)

            predicted_segmentation.append(pred_seg[None])
            softmax_pred.append(softmax_pres[None])

        predicted_segmentation = np.vstack(predicted_segmentation)
        softmax_pred = np.vstack(softmax_pred).transpose((1, 0, 2, 3))

        return predicted_segmentation, softmax_pred


if __name__ == '__main__':
    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(162, 529, 529), 0.5))
    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(162, 529, 529), 1))
    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(162, 529, 529), 0.1))

    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(60, 448, 224), 1))
    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(60, 448, 224), 0.5))

    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(30, 224, 224), 1))
    print(SegmentationNetwork._compute_steps_for_sliding_window((30, 224, 224),
(30, 224, 224), 0.125))
```

```
    print(SegmentationNetwork._compute_steps_for_sliding_window((123, 54, 123),
(246, 162, 369), 0.25))
```