Code Printout of UMamba_Moe

Pradeep Rajan

May 3, 2024

Introduction

This document contains the source code of four main files. The MambaBot_Moe Listing is where we have implemented the U-Net, Mamba Network and Mixture of experts as well. The Gating Mechanism is defined at the start of the code and, the Mixture of Experts and Mamba model at the Encoder part of the code. In this listing the MoE is implemented at the bottleneck part of the U-Net, where it can have a significant impact on the processing of complex features before the decoding phase begins. The MambaEnc_Moe Listing also has the U-Net, Mamba Network and Mixture of Experts defined as well. Here the MoE are implemented at every stage of the encoding process, enhancing the model's ability to handle diverse and complex image features effectively.

The nnUNetTrainerUMambaBotMoe and nnUNetTrainerUMambaEncMoe listings are custom training classes which is an extension of the nnUNetTrainer. The nnUnetTrainer handles all the complex matters. As the name suggest, they are designed to train 'UMambaBot' and 'UMambaEnc', respectively. The build network architecture method dynamically constructs the UMambaBotMoe network architecture. It does this by calling get_umamba_bot_moe_from_plans, which helps configure the model. The same goes for UMambaEncMoe as well.

Code listings

In what follows, we list the source code that has been modified to implement the Mixture of Experts in the U-Mamba network.

Listings

Listing 1: MambaBot_Moe

```
import numpy as np
import torch
from torch import nn
from typing import Union, Type, List, Tuple

from dynamic_network_architectures.building_blocks.helper import
    get_matching_convtransp
from dynamic_network_architectures.building_blocks.residual_encoders import
    ResidualEncoder
from dynamic_network_architectures.building_blocks.plain_conv_encoder import
    PlainConvEncoder
from dynamic_network_architectures.building_blocks.residual import
    StackedResidualBlocks
from dynamic_network_architectures.building_blocks.residual_encoders import
    ResidualEncoder
from dynamic_network_architectures.building_blocks.residual import BasicBlockD,
    BottleneckD
from torch.nn.modules.conv import _ConvNd
from torch.nn.modules.dropout import _DropoutNd
from dynamic_network_architectures.building_blocks.helper import
    convert_conv_op_to_dim
```

```
from nnunetv2.utilities.plans_handling.plans_handler import ConfigurationManager,
    PlansManager
from dynamic_network_architectures.building_blocks.helper import
    get_matching_instancenorm, convert_dim_to_conv_op
from nnunetv2.utilities.network_initialization import InitWeights_He
from mamba_ssm import Mamba


class GatingMechanism(nn.Module):
    def __init__(self, d_model, num_experts, hidden_size=64):
        super(GatingMechanism, self).__init__()
        self.d_model = d_model
        self.num_experts = num_experts
        self.gating_network = nn.Sequential(
            nn.Linear(d_model, hidden_size),
            nn.Linear(hidden_size, num_experts)
        )

    def forward(self, x):
        gates = torch.softmax(self.gating_network(x), dim=-1)
        return gates

class UNetResDecoder(nn.Module):
    def __init__(self,
                 encoder: Union[PlainConvEncoder, ResidualEncoder],
                 num_classes: int,
                 n_conv_per_stage: Union[int, Tuple[int, ...], List[int]],
                 deep_supervision, nonlin_first: bool = False):
        """
        This class needs the skips of the encoder as input in its forward.

        the encoder goes all the way to the bottleneck, so that's where the
            decoder picks up. stages in the decoder
        are sorted by order of computation, so the first stage has the lowest
            resolution and takes the bottleneck
        features and the lowest skip as inputs
        the decoder has two (three) parts in each stage:
        1) conv transpose to upsample the feature maps of the stage below it (or
            the bottleneck in case of the first stage)
        2) n_conv_per_stage conv blocks to let the two inputs get to know each
            other and merge
        3) (optional if deep_supervision=True) a segmentation output Todo: enable
             upsample logits?
        :param encoder:
        :param num_classes:
        :param n_conv_per_stage:
        :param deep_supervision:
        """
        super().__init__()
        self.deep_supervision = deep_supervision
        self.encoder = encoder
        self.num_classes = num_classes
        n_stages_encoder = len(encoder.output_channels)
        if isinstance(n_conv_per_stage, int):
            n_conv_per_stage = [n_conv_per_stage] * (n_stages_encoder - 1)
        assert len(n_conv_per_stage) == n_stages_encoder - 1, "n_conv_per_stage
            must have as many entries as we have " \
                                                          "resolution stages - 1
                                                              (n_stages in encoder
                                                               - 1), " \
```

```python
                                                            "here: %d" %
                                                                n_stages_encoder

        transpconv_op = get_matching_convtransp(conv_op=encoder.conv_op)

        # we start with the bottleneck and work out way up
        stages = []
        transpconvs = []
        seg_layers = []
        for s in range(1, n_stages_encoder):
            input_features_below = encoder.output_channels[-s]
            input_features_skip = encoder.output_channels[-(s + 1)]
            stride_for_transpconv = encoder.strides[-s]
            transpconvs.append(transpconv_op(
                input_features_below, input_features_skip, stride_for_transpconv,
                    stride_for_transpconv,
                bias=encoder.conv_bias
            ))
            # input features to conv is 2x input_features_skip (concat
                input_features_skip with transpconv output)
            stages.append(StackedResidualBlocks(
                n_blocks = n_conv_per_stage[s-1],
                conv_op = encoder.conv_op,
                input_channels = 2 * input_features_skip,
                output_channels = input_features_skip,
                kernel_size = encoder.kernel_sizes[-(s + 1)],
                initial_stride = 1,
                conv_bias = encoder.conv_bias,
                norm_op = encoder.norm_op,
                norm_op_kwargs = encoder.norm_op_kwargs,
                dropout_op = encoder.dropout_op,
                dropout_op_kwargs = encoder.dropout_op_kwargs,
                nonlin = encoder.nonlin,
                nonlin_kwargs = encoder.nonlin_kwargs,
            ))

            # we always build the deep supervision outputs so that we can always
                load parameters. If we don't do this
            # then a model trained with deep_supervision=True could not easily be
                 loaded at inference time where
            # deep supervision is not needed. It's just a convenience thing
            seg_layers.append(encoder.conv_op(input_features_skip, num_classes,
                1, 1, 0, bias=True))

        self.stages = nn.ModuleList(stages)
        self.transpconvs = nn.ModuleList(transpconvs)
        self.seg_layers = nn.ModuleList(seg_layers)

    def forward(self, skips):
        """
        we expect to get the skips in the order they were computed, so the
            bottleneck should be the last entry
        :param skips:
        :return:
        """
        lres_input = skips[-1]
        seg_outputs = []
        for s in range(len(self.stages)):
            x = self.transpconvs[s](lres_input)
            x = torch.cat((x, skips[-(s+2)]), 1)
            x = self.stages[s](x)
```

```
            if self.deep_supervision:
                seg_outputs.append(self.seg_layers[s](x))
            elif s == (len(self.stages) - 1):
                seg_outputs.append(self.seg_layers[-1](x))
            lres_input = x

        # invert seg outputs so that the largest segmentation prediction is
            returned first
        seg_outputs = seg_outputs[::-1]

        if not self.deep_supervision:
            r = seg_outputs[0]
        else:
            r = seg_outputs
        return r

    def compute_conv_feature_map_size(self, input_size):
        """
        IMPORTANT: input_size is the input_size of the encoder!
        :param input_size:
        :return:
        """
        # first we need to compute the skip sizes. Skip bottleneck because all
            output feature maps of our ops will at
        # least have the size of the skip above that (therefore -1)
        skip_sizes = []
        for s in range(len(self.encoder.strides) - 1):
            skip_sizes.append([i // j for i, j in zip(input_size, self.encoder.
                strides[s])])
            input_size = skip_sizes[-1]
        # print(skip_sizes)

        assert len(skip_sizes) == len(self.stages)

        # our ops are the other way around, so let's match things up
        output = np.int64(0)
        for s in range(len(self.stages)):
            # print(skip_sizes[-(s+1)], self.encoder.output_channels[-(s+2)])
            # conv blocks
            output += self.stages[s].compute_conv_feature_map_size(skip_sizes[-(s
                +1)])
            # trans conv
            output += np.prod([self.encoder.output_channels[-(s+2)], *skip_sizes
                [-(s+1)]], dtype=np.int64)
            # segmentation
            if self.deep_supervision or (s == (len(self.stages) - 1)):
                output += np.prod([self.num_classes, *skip_sizes[-(s+1)]], dtype=
                    np.int64)
        return output

class UMambaBotMoe(nn.Module):
    def __init__(self,
                 input_channels: int,
                 n_stages: int,
                 features_per_stage: Union[int, List[int], Tuple[int, ...]],
                 conv_op: Type[_ConvNd],
                 kernel_sizes: Union[int, List[int], Tuple[int, ...]],
                 strides: Union[int, List[int], Tuple[int, ...]],
                 n_conv_per_stage: Union[int, List[int], Tuple[int, ...]],
                 num_classes: int,
                 n_conv_per_stage_decoder: Union[int, Tuple[int, ...], List[int
```

```
                ]],
                conv_bias: bool = False,
                norm_op: Union[None, Type[nn.Module]] = None,
                norm_op_kwargs: dict = None,
                dropout_op: Union[None, Type[_DropoutNd]] = None,
                dropout_op_kwargs: dict = None,
                nonlin: Union[None, Type[torch.nn.Module]] = None,
                nonlin_kwargs: dict = None,
                deep_supervision: bool = False,
                block: Union[Type[BasicBlockD], Type[BottleneckD]] = BasicBlockD
                    ,
                bottleneck_channels: Union[int, List[int], Tuple[int, ...]] =
                    None,
                stem_channels: int = None,
                num_experts: int = 1
                ):
        super().__init__()
        self.num_experts = num_experts
        n_blocks_per_stage = n_conv_per_stage
        if isinstance(n_blocks_per_stage, int):
            n_blocks_per_stage = [n_blocks_per_stage] * n_stages
        if isinstance(n_conv_per_stage_decoder, int):
            n_conv_per_stage_decoder = [n_conv_per_stage_decoder] * (n_stages -
                1)
        assert len(n_blocks_per_stage) == n_stages, "n_blocks_per_stage must have
             as many entries as we have " \
                                                  f"resolution stages. here: {
                                                      n_stages}. " \
                                                  f"n_blocks_per_stage: {
                                                      n_blocks_per_stage}"
        assert len(n_conv_per_stage_decoder) == (n_stages - 1), "
            n_conv_per_stage_decoder must have one less entries " \
                                                    f"as we have resolution
                                                        stages. here: {n_stages} "
                                                         \
                                                    f"stages, so it should have {
                                                        n_stages - 1} entries. " \
                                                    f"n_conv_per_stage_decoder: {
                                                        n_conv_per_stage_decoder}"
        self.encoder = ResidualEncoder(input_channels, n_stages,
            features_per_stage, conv_op, kernel_sizes, strides,
                                       n_blocks_per_stage, conv_bias, norm_op,
                                           norm_op_kwargs, dropout_op,
                                       dropout_op_kwargs, nonlin, nonlin_kwargs,
                                           block, bottleneck_channels,
                                       return_skips=True, disable_default_stem=
                                           False, stem_channels=stem_channels)
        # layer norm
        self.ln = nn.LayerNorm(features_per_stage[-1])
        self.mamba_experts = nn.ModuleList([Mamba(
                        d_model=features_per_stage[-1],
                        d_state=16,
                        d_conv=4,
                        expand=2,
                ) for _ in range(num_experts)])
        self.gating_mechanism = GatingMechanism(features_per_stage[-1],
            num_experts)
        self.decoder = UNetResDecoder(self.encoder, num_classes,
            n_conv_per_stage_decoder, deep_supervision)

    def determine_dimensionality(self, x):
```

```python
        if x.dim() == 4:
            return 2
        elif x.dim() == 5:
            return 3
        else:
            raise ValueError("Unsupported input dimension. Expected 4D (2D images
               ) or 5D (3D images)")


    def forward(self, x):
        self.dimensionality = self.determine_dimensionality(x)
        skips = self.encoder(x)
        middle_feature = skips[-1]

        if self.dimensionality == 2:
            B, C, H, W = middle_feature.shape
            flat_dim = H * W
        elif self.dimensionality == 3:
            B, C, H, W, D = middle_feature.shape
            flat_dim = H * W * D

        middle_feature_flat = middle_feature.view(B, C, flat_dim).transpose(-1,
            -2)
        middle_feature_flat = self.ln(middle_feature_flat)
        gates = self.gating_mechanism(middle_feature_flat)
        expert_outputs = [expert(middle_feature_flat) for expert in self.
            mamba_experts]
        expert_outputs = torch.stack(expert_outputs, dim=0)
        gated_outputs = torch.einsum('bnm,ebnc->bnc', gates, expert_outputs)
        gated_outputs = gated_outputs.transpose(-1, -2).view(B, C, *
            middle_feature.shape[2:])
        skips[-1] = gated_outputs
        return self.decoder(skips)

    def compute_conv_feature_map_size(self, input_size):
        assert len(input_size) == convert_conv_op_to_dim(self.encoder.conv_op), "
            just give the image size without color/feature channels or "
        return self.encoder.compute_conv_feature_map_size(input_size) + self.
            decoder.compute_conv_feature_map_size(input_size)


def get_umamba_bot_moe_from_plans(plans_manager: PlansManager,
                           dataset_json: dict,
                           configuration_manager: ConfigurationManager,
                           num_input_channels: int,
                           deep_supervision: bool = True):
    """
    we may have to change this in the future to accommodate other plans ->
        network mappings

    num_input_channels can differ depending on whether we do cascade. Its best to
         make this info available in the
    trainer rather than inferring it again from the plans here.
    """
    num_stages = len(configuration_manager.conv_kernel_sizes)

    dim = len(configuration_manager.conv_kernel_sizes[0])
    conv_op = convert_dim_to_conv_op(dim)

    label_manager = plans_manager.get_label_manager(dataset_json)
```

```
segmentation_network_class_name = 'UMambaBotMoe'
network_class = UMambaBotMoe
kwargs = {
    'UMambaBotMoe': {
        'conv_bias': True,
        'norm_op': get_matching_instancenorm(conv_op),
        'norm_op_kwargs': {'eps': 1e-5, 'affine': True},
        'dropout_op': None, 'dropout_op_kwargs': None,
        'nonlin': nn.LeakyReLU, 'nonlin_kwargs': {'inplace': True},
    }
}

conv_or_blocks_per_stage = {
    'n_conv_per_stage': configuration_manager.n_conv_per_stage_encoder,
    'n_conv_per_stage_decoder': configuration_manager.
       n_conv_per_stage_decoder
}

model = network_class(
    input_channels=num_input_channels,
    n_stages=num_stages,
    features_per_stage=[min(configuration_manager.UNet_base_num_features * 2
       ** i,
                            configuration_manager.unet_max_num_features) for
                               i in range(num_stages)],
    conv_op=conv_op,
    kernel_sizes=configuration_manager.conv_kernel_sizes,
    strides=configuration_manager.pool_op_kernel_sizes,
    num_classes=label_manager.num_segmentation_heads,
    deep_supervision=deep_supervision,
    **conv_or_blocks_per_stage,
    **kwargs[segmentation_network_class_name]
)
model.apply(InitWeights_He(1e-2))

return model
```

Listing 2: nnUNetTrainerUMambaBotMoe

```
from nnunetv2.training.nnUNetTrainer.nnUNetTrainer import nnUNetTrainer
from nnunetv2.utilities.plans_handling.plans_handler import ConfigurationManager,
    PlansManager
from torch import nn
from nnunetv2.nets.UmambaMoe import get_umamba_bot_moe_from_plans


class nnUNetTrainerUMambaBotMoe(nnUNetTrainer):
    """
    Residual Encoder + UMmaba Bottleneck + Residual Decoder + Skip Connections
    """
    @staticmethod
    def build_network_architecture(plans_manager: PlansManager,
                                   dataset_json,
                                   configuration_manager: ConfigurationManager,
                                   num_input_channels,
                                   enable_deep_supervision: bool = True) -> nn.
                                      Module:

        model = get_umamba_bot_moe_from_plans(plans_manager, dataset_json,
           configuration_manager,
                                      num_input_channels, deep_supervision=
```

```
                                            enable_deep_supervision)

        print("UMambaBotMoe: {}".format(model))

        return model
```

Listing 3: MambaEnc_Moe

```
import numpy as np
import torch
from torch import nn
from typing import Union, Type, List, Tuple

from dynamic_network_architectures.building_blocks.helper import
    get_matching_convtransp
from dynamic_network_architectures.building_blocks.plain_conv_encoder import
    PlainConvEncoder

from dynamic_network_architectures.building_blocks.simple_conv_blocks import
    StackedConvBlocks
from dynamic_network_architectures.building_blocks.residual import
    StackedResidualBlocks

from dynamic_network_architectures.building_blocks.helper import
    maybe_convert_scalar_to_list, get_matching_pool_op
from dynamic_network_architectures.building_blocks.residual import BasicBlockD,
    BottleneckD
from torch.nn.modules.conv import _ConvNd
from torch.nn.modules.dropout import _DropoutNd
from torch.cuda.amp import autocast
from dynamic_network_architectures.building_blocks.helper import
    convert_conv_op_to_dim
from nnunetv2.utilities.plans_handling.plans_handler import ConfigurationManager,
     PlansManager
from dynamic_network_architectures.building_blocks.helper import
    get_matching_instancenorm, convert_dim_to_conv_op
from dynamic_network_architectures.initialization.weight_init import
    init_last_bn_before_add_to_0
from nnunetv2.utilities.network_initialization import InitWeights_He
from mamba_ssm import Mamba

class MambaLayer(nn.Module):
    def __init__(self, dim, num_experts=4, d_state=16, d_conv=4, expand=2):
        super().__init__()
        self.dim = dim
        self.num_experts = num_experts
        self.norm = nn.LayerNorm(dim)
        self.experts = nn.ModuleList([
            Mamba(d_model=dim, d_state=d_state, d_conv=d_conv, expand=expand)
            for _ in range(num_experts)
        ])
        self.gating_network = nn.Linear(dim, num_experts)

    def forward(self, x):
        B, C, *rest = x.shape
        #assert C == self.dim, "Channel dimension mismatch"
        n_tokens = np.prod(rest)
        x_flat = x.reshape(B, C, n_tokens).transpose(-1, -2)
        x_norm = self.norm(x_flat)
        gates = torch.softmax(self.gating_network(x_norm), dim=-1)
        expert_outputs = [expert(x_norm) for expert in self.experts]
```

```
        expert_outputs = torch.stack(expert_outputs, dim=0)
        gates_expanded = gates.unsqueeze(-1)
        combined_output = torch.einsum('bnec,ebnc->bnc', gates_expanded,
            expert_outputs)
        out = combined_output.transpose(-1, -2).reshape(B, C, *rest)
        return out


class ResidualMambaEncoder(nn.Module):
    def __init__(self,
                 input_channels: int,
                 n_stages: int,
                 features_per_stage: Union[int, List[int], Tuple[int, ...]],
                 conv_op: Type[_ConvNd],
                 kernel_sizes: Union[int, List[int], Tuple[int, ...]],
                 strides: Union[int, List[int], Tuple[int, ...], Tuple[Tuple[int,
                     ...], ...]],
                 n_blocks_per_stage: Union[int, List[int], Tuple[int, ...]],
                 conv_bias: bool = False,
                 norm_op: Union[None, Type[nn.Module]] = None,
                 norm_op_kwargs: dict = None,
                 dropout_op: Union[None, Type[_DropoutNd]] = None,
                 dropout_op_kwargs: dict = None,
                 nonlin: Union[None, Type[torch.nn.Module]] = None,
                 nonlin_kwargs: dict = None,
                 block: Union[Type[BasicBlockD], Type[BottleneckD]] = BasicBlockD
                     ,
                 bottleneck_channels: Union[int, List[int], Tuple[int, ...]] =
                    None,
                 return_skips: bool = False,
                 disable_default_stem: bool = False,
                 stem_channels: int = None,
                 pool_type: str = 'conv',
                 stochastic_depth_p: float = 0.0,
                 squeeze_excitation: bool = False,
                 squeeze_excitation_reduction_ratio: float = 1. / 16,
                 num_experts: int = 1
                 ):
        super().__init__()
        if isinstance(kernel_sizes, int):
            kernel_sizes = [kernel_sizes] * n_stages
        if isinstance(features_per_stage, int):
            features_per_stage = [features_per_stage] * n_stages
        if isinstance(n_blocks_per_stage, int):
            n_blocks_per_stage = [n_blocks_per_stage] * n_stages
        if isinstance(strides, int):
            strides = [strides] * n_stages
        if bottleneck_channels is None or isinstance(bottleneck_channels, int):
            bottleneck_channels = [bottleneck_channels] * n_stages
        assert len(
            bottleneck_channels) == n_stages, "bottleneck_channels must be None
                or have as many entries as we have resolution stages (n_stages)"
        assert len(
            kernel_sizes) == n_stages, "kernel_sizes must have as many entries as
                 we have resolution stages (n_stages)"
        assert len(
            n_blocks_per_stage) == n_stages, "n_conv_per_stage must have as many
                entries as we have resolution stages (n_stages)"
        assert len(
            features_per_stage) == n_stages, "features_per_stage must have as
                many entries as we have resolution stages (n_stages)"
```

```
assert len(strides) == n_stages, "strides must have as many entries as we
    have resolution stages (n_stages). " \
                                 "Important: first entry is recommended
                                    to be 1, else we run strided conv
                                    drectly on the input"

pool_op = get_matching_pool_op(conv_op, pool_type=pool_type) if pool_type
    != 'conv' else None

# build a stem, Todo maybe we need more flexibility for this in the
   future. For now, if you need a custom
#  stem you can just disable the stem and build your own.
#  THE STEM DOES NOT DO STRIDE/POOLING IN THIS IMPLEMENTATION
if not disable_default_stem:
    if stem_channels is None:
        stem_channels = features_per_stage[0]
    self.stem = StackedConvBlocks(1, conv_op, input_channels,
       stem_channels, kernel_sizes[0], 1, conv_bias,
                                  norm_op, norm_op_kwargs, dropout_op,
                                     dropout_op_kwargs, nonlin,
                                     nonlin_kwargs)
    input_channels = stem_channels
else:
    self.stem = None

# now build the network
stages = []
mamba_layers = []
for s in range(n_stages):
    stride_for_conv = strides[s] if pool_op is None else 1

    stage = StackedResidualBlocks(
        n_blocks_per_stage[s], conv_op, input_channels,
           features_per_stage[s], kernel_sizes[s], stride_for_conv,
        conv_bias, norm_op, norm_op_kwargs, dropout_op, dropout_op_kwargs
           , nonlin, nonlin_kwargs,
        block=block, bottleneck_channels=bottleneck_channels[s],
           stochastic_depth_p=stochastic_depth_p,
        squeeze_excitation=squeeze_excitation,
        squeeze_excitation_reduction_ratio=
           squeeze_excitation_reduction_ratio
    )

    if pool_op is not None:
        stage = nn.Sequential(pool_op(strides[s]), stage)

    stages.append(stage)
    input_channels = features_per_stage[s]

    mamba_layers.append(MambaLayer(input_channels))

#self.stages = nn.Sequential(*stages)
self.stages = nn.ModuleList(stages)
self.output_channels = features_per_stage
self.strides = [maybe_convert_scalar_to_list(conv_op, i) for i in strides
   ]
self.return_skips = return_skips

# we store some things that a potential decoder needs
self.conv_op = conv_op
self.norm_op = norm_op
```

```
        self.norm_op_kwargs = norm_op_kwargs
        self.nonlin = nonlin
        self.nonlin_kwargs = nonlin_kwargs
        self.dropout_op = dropout_op
        self.dropout_op_kwargs = dropout_op_kwargs
        self.conv_bias = conv_bias
        self.kernel_sizes = kernel_sizes

        self.mamba_layers = nn.ModuleList(mamba_layers)

    def forward(self, x):
        if self.stem is not None:
            x = self.stem(x)
        ret = []
        #for s in self.stages:
        for s in range(len(self.stages)):
            #x = s(x)
            x = self.stages[s](x)
            x = self.mamba_layers[s](x)
            ret.append(x)
        if self.return_skips:
            return ret
        else:
            return ret[-1]

    def compute_conv_feature_map_size(self, input_size):
        if self.stem is not None:
            output = self.stem.compute_conv_feature_map_size(input_size)
        else:
            output = np.int64(0)

        for s in range(len(self.stages)):
            output += self.stages[s].compute_conv_feature_map_size(input_size)
            input_size = [i // j for i, j in zip(input_size, self.strides[s])]

        return output

class UNetResDecoder(nn.Module):
    def __init__(self,
                 encoder: Union[PlainConvEncoder, ResidualMambaEncoder],
                 num_classes: int,
                 n_conv_per_stage: Union[int, Tuple[int, ...], List[int]],
                 deep_supervision, nonlin_first: bool = False):
        """
        This class needs the skips of the encoder as input in its forward.

        the encoder goes all the way to the bottleneck, so that's where the
            decoder picks up. stages in the decoder
        are sorted by order of computation, so the first stage has the lowest
            resolution and takes the bottleneck
        features and the lowest skip as inputs
        the decoder has two (three) parts in each stage:
        1) conv transpose to upsample the feature maps of the stage below it (or
            the bottleneck in case of the first stage)
        2) n_conv_per_stage conv blocks to let the two inputs get to know each
            other and merge
        3) (optional if deep_supervision=True) a segmentation output Todo: enable
             upsample logits?
        :param encoder:
        :param num_classes:
        :param n_conv_per_stage:
```

```
    :param deep_supervision:
    """
    super().__init__()
    self.deep_supervision = deep_supervision
    self.encoder = encoder
    self.num_classes = num_classes
    n_stages_encoder = len(encoder.output_channels)
    if isinstance(n_conv_per_stage, int):
        n_conv_per_stage = [n_conv_per_stage] * (n_stages_encoder - 1)
    assert len(n_conv_per_stage) == n_stages_encoder - 1, "n_conv_per_stage
       must have as many entries as we have " \
                                                       "resolution stages - 1
                                                          (n_stages in encoder
                                                           - 1), " \
                                                       "here: %d" %
                                                          n_stages_encoder

    transpconv_op = get_matching_convtransp(conv_op=encoder.conv_op)

    # we start with the bottleneck and work out way up
    stages = []
    transpconvs = []
    seg_layers = []
    for s in range(1, n_stages_encoder):
        input_features_below = encoder.output_channels[-s]
        input_features_skip = encoder.output_channels[-(s + 1)]
        stride_for_transpconv = encoder.strides[-s]
        transpconvs.append(transpconv_op(
            input_features_below, input_features_skip, stride_for_transpconv,
                stride_for_transpconv,
            bias=encoder.conv_bias
        ))
        # input features to conv is 2x input_features_skip (concat
           input_features_skip with transpconv output)
        stages.append(StackedResidualBlocks(
            n_blocks = n_conv_per_stage[s-1],
            conv_op = encoder.conv_op,
            input_channels = 2 * input_features_skip,
            output_channels = input_features_skip,
            kernel_size = encoder.kernel_sizes[-(s + 1)],
            initial_stride = 1,
            conv_bias = encoder.conv_bias,
            norm_op = encoder.norm_op,
            norm_op_kwargs = encoder.norm_op_kwargs,
            dropout_op = encoder.dropout_op,
            dropout_op_kwargs = encoder.dropout_op_kwargs,
            nonlin = encoder.nonlin,
            nonlin_kwargs = encoder.nonlin_kwargs,
        ))
        # we always build the deep supervision outputs so that we can always
           load parameters. If we don't do this
        # then a model trained with deep_supervision=True could not easily be
            loaded at inference time where
        # deep supervision is not needed. It's just a convenience thing
        seg_layers.append(encoder.conv_op(input_features_skip, num_classes,
           1, 1, 0, bias=True))

    self.stages = nn.ModuleList(stages)
    self.transpconvs = nn.ModuleList(transpconvs)
    self.seg_layers = nn.ModuleList(seg_layers)
```

```python
def forward(self, skips):
    """
    we expect to get the skips in the order they were computed, so the
        bottleneck should be the last entry
    :param skips:
    :return:
    """
    lres_input = skips[-1]
    seg_outputs = []
    for s in range(len(self.stages)):
        x = self.transpconvs[s](lres_input)
        x = torch.cat((x, skips[-(s+2)]), 1)
        x = self.stages[s](x)
        if self.deep_supervision:
            seg_outputs.append(self.seg_layers[s](x))
        elif s == (len(self.stages) - 1):
            seg_outputs.append(self.seg_layers[-1](x))
        lres_input = x

    # invert seg outputs so that the largest segmentation prediction is
        returned first
    seg_outputs = seg_outputs[::-1]

    if not self.deep_supervision:
        r = seg_outputs[0]
    else:
        r = seg_outputs
    return r

def compute_conv_feature_map_size(self, input_size):
    """
    IMPORTANT: input_size is the input_size of the encoder!
    :param input_size:
    :return:
    """
    # first we need to compute the skip sizes. Skip bottleneck because all
        output feature maps of our ops will at
    # least have the size of the skip above that (therefore -1)
    skip_sizes = []
    for s in range(len(self.encoder.strides) - 1):
        skip_sizes.append([i // j for i, j in zip(input_size, self.encoder.
            strides[s])])
        input_size = skip_sizes[-1]
    # print(skip_sizes)

    assert len(skip_sizes) == len(self.stages)

    # our ops are the other way around, so let's match things up
    output = np.int64(0)
    for s in range(len(self.stages)):
        # print(skip_sizes[-(s+1)], self.encoder.output_channels[-(s+2)])
        # conv blocks
        output += self.stages[s].compute_conv_feature_map_size(skip_sizes[-(s
            +1)])
        # trans conv
        output += np.prod([self.encoder.output_channels[-(s+2)], *skip_sizes
            [-(s+1)]], dtype=np.int64)
        # segmentation
        if self.deep_supervision or (s == (len(self.stages) - 1)):
            output += np.prod([self.num_classes, *skip_sizes[-(s+1)]], dtype=
                np.int64)
```

```python
        return output

class UMambaEncMoe(nn.Module):
    def __init__(self,
                 input_channels: int,
                 n_stages: int,
                 features_per_stage: Union[int, List[int], Tuple[int, ...]],
                 conv_op: Type[_ConvNd],
                 kernel_sizes: Union[int, List[int], Tuple[int, ...]],
                 strides: Union[int, List[int], Tuple[int, ...]],
                 n_conv_per_stage: Union[int, List[int], Tuple[int, ...]],
                 num_classes: int,
                 n_conv_per_stage_decoder: Union[int, Tuple[int, ...], List[int
                    ]],
                 conv_bias: bool = False,
                 norm_op: Union[None, Type[nn.Module]] = None,
                 norm_op_kwargs: dict = None,
                 dropout_op: Union[None, Type[_DropoutNd]] = None,
                 dropout_op_kwargs: dict = None,
                 nonlin: Union[None, Type[torch.nn.Module]] = None,
                 nonlin_kwargs: dict = None,
                 deep_supervision: bool = False,
                 block: Union[Type[BasicBlockD], Type[BottleneckD]] = BasicBlockD
                    ,
                 bottleneck_channels: Union[int, List[int], Tuple[int, ...]] =
                    None,
                 stem_channels: int = None
                 ):
        super().__init__()
        n_blocks_per_stage = n_conv_per_stage
        if isinstance(n_blocks_per_stage, int):
            n_blocks_per_stage = [n_blocks_per_stage] * n_stages
        if isinstance(n_conv_per_stage_decoder, int):
            n_conv_per_stage_decoder = [n_conv_per_stage_decoder] * (n_stages -
                1)
        assert len(n_blocks_per_stage) == n_stages, "n_blocks_per_stage must have
             as many entries as we have " \
                                                    f"resolution stages. here: {
                                                        n_stages}. " \
                                                    f"n_blocks_per_stage: {
                                                        n_blocks_per_stage}"
        assert len(n_conv_per_stage_decoder) == (n_stages - 1), "
            n_conv_per_stage_decoder must have one less entries " \
                                                    f"as we have resolution stages.
                                                        here: {n_stages} " \
                                                f"stages, so it should have {
                                                    n_stages - 1} entries. " \
                                                    f"n_conv_per_stage_decoder: {
                                                        n_conv_per_stage_decoder}"
        self.encoder = ResidualMambaEncoder(input_channels, n_stages,
            features_per_stage, conv_op, kernel_sizes, strides,
                                         n_blocks_per_stage, conv_bias, norm_op,
                                            norm_op_kwargs, dropout_op,
                                         dropout_op_kwargs, nonlin, nonlin_kwargs,
                                            block, bottleneck_channels,
                                         return_skips=True, disable_default_stem=
                                            False, stem_channels=stem_channels)
        self.decoder = UNetResDecoder(self.encoder, num_classes,
            n_conv_per_stage_decoder, deep_supervision)

    def forward(self, x):
```

```python
        skips = self.encoder(x)
        return self.decoder(skips)

    def compute_conv_feature_map_size(self, input_size):
        assert len(input_size) == convert_conv_op_to_dim(self.encoder.conv_op), "
            just give the image size without color/feature channels or "
        return self.encoder.compute_conv_feature_map_size(input_size) + self.
            decoder.compute_conv_feature_map_size(input_size)


def get_umamba_enc_moe_from_plans(plans_manager: PlansManager,
                            dataset_json: dict,
                            configuration_manager: ConfigurationManager,
                            num_input_channels: int,
                            deep_supervision: bool = True):
    """
    we may have to change this in the future to accommodate other plans ->
        network mappings

    num_input_channels can differ depending on whether we do cascade. Its best to
         make this info available in the
    trainer rather than inferring it again from the plans here.
    """
    num_stages = len(configuration_manager.conv_kernel_sizes)

    dim = len(configuration_manager.conv_kernel_sizes[0])
    conv_op = convert_dim_to_conv_op(dim)

    label_manager = plans_manager.get_label_manager(dataset_json)

    segmentation_network_class_name = 'UMambaEncMoe'
    network_class = UMambaEncMoe
    kwargs = {
        'UMambaEncMoe': {
            'conv_bias': True,
            'norm_op': get_matching_instancenorm(conv_op),
            'norm_op_kwargs': {'eps': 1e-5, 'affine': True},
            'dropout_op': None, 'dropout_op_kwargs': None,
            'nonlin': nn.LeakyReLU, 'nonlin_kwargs': {'inplace': True},
        }
    }

    conv_or_blocks_per_stage = {
        'n_conv_per_stage': configuration_manager.n_conv_per_stage_encoder,
        'n_conv_per_stage_decoder': configuration_manager.
            n_conv_per_stage_decoder
    }

    model = network_class(
        input_channels=num_input_channels,
        n_stages=num_stages,
        features_per_stage=[min(configuration_manager.UNet_base_num_features * 2
            ** i,
                                configuration_manager.unet_max_num_features) for
                                    i in range(num_stages)],
        conv_op=conv_op,
        kernel_sizes=configuration_manager.conv_kernel_sizes,
        strides=configuration_manager.pool_op_kernel_sizes,
        num_classes=label_manager.num_segmentation_heads,
        deep_supervision=deep_supervision,
        **conv_or_blocks_per_stage,
```

```
        **kwargs[segmentation_network_class_name]
    )
    model.apply(InitWeights_He(1e-2))
    if network_class == UMambaEncMoe:
        model.apply(init_last_bn_before_add_to_0)

    return model
```

Listing 4: nnUNetTrainerUMambaEncMoe

```
import torch
from nnunetv2.training.nnUNetTrainer.nnUNetTrainer import nnUNetTrainer
from nnunetv2.utilities.plans_handling.plans_handler import ConfigurationManager,
    PlansManager
from torch import nn

from nnunetv2.nets.UMambaEncMoe import get_umamba_enc_moe_from_plans

class nnUNetTrainerUMambaEncMoe(nnUNetTrainer):

    def __init__(self, plans: dict, configuration: str, fold: int, dataset_json:
        dict, unpack_dataset: bool = True,
                 device: torch.device = torch.device('cuda')):
        super().__init__(plans, configuration, fold, dataset_json, unpack_dataset
            , device)

    @staticmethod
    def build_network_architecture(plans_manager: PlansManager,
                                   dataset_json,
                                   configuration_manager: ConfigurationManager,
                                   num_input_channels,
                                   enable_deep_supervision: bool = True) -> nn.
                                       Module:

        model = get_umamba_enc_moe_from_plans(plans_manager, dataset_json,
            configuration_manager,
                                      num_input_channels, deep_supervision=
                                          enable_deep_supervision)

        print("UMambaEncMoe: {}".format(model))

        return model
```