debug

June 17, 2023

[ ]:

```
from utils.metrics import metStat

a = metStat()

print(f"{a:.4f}")
```

[ ]:

```
class a():
    def __init__(self):
        self.b = 1

test = a()

setattr(test, 'c', 2)
getattr(test, 'c')
```

[ ]:

```
import torch
from model import EncoderUNetModel

a = EncoderUNetModel((16, 128,128), 3, 32, 128, 2, (8,), time_embed= None, pool␣
 ↪= "none", dims= 3)

pics = torch.randn(1, 3, 16, 128, 128)
a(pics).shape
```

[ ]:

```
from torch import nn
hidden_dim = 128
row_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
col_embed = nn.Parameter(torch.rand(50, hidden_dim // 2))
H, W = 16, 16
print(col_embed[:W].unsqueeze(0).repeat(H, 1, 1).shape)
print(row_embed[:H].unsqueeze(1).repeat(1, W, 1).shape)
pos = torch.cat([
            col_embed[:W].unsqueeze(0).repeat(H, 1, 1),
            row_embed[:H].unsqueeze(1).repeat(1, W, 1),
        ], dim=-1).flatten(0, 1).unsqueeze(1)
print(pos.shape)
```

[ ]:

```
import torch
from model import Regression
a = Regression(in_size = (16, 128, 128), out_size = (4, 32, 32), out_channels =␣
 ↪8)
pics = torch.randn(1, 32, 16, 128, 128)
a(pics).shape
a = Regression(in_size = (16, 128, 128), out_size = (4, 32, 32), out_channels =␣
 ↪(3, 3))
pics = torch.randn(1, 32, 16, 128, 128)
for i in a(pics):
    print(i.shape)
```

[ ]:

```
import torch
a = torch.ones(1)
b = a.clone()
a.zero_()
print(b)
poscar.pos2boxncls()
```

[ ]:

```
import torch
from torch import nn
from typing import Optional, Callable, Tuple
import math
import einops
import matplotlib.pyplot as plt
from functools import partial
from torch.nn import functional as F
from  torchvision.ops import MLP

# github https://github.com/tatp22/multidim-positional-encoding/blob/master/
 ↪positional_encodings/torch_encodings.py
class Sinnembed_SpacialEncoding(nn.Module):
    def __init__(self, channels):
        """
        :param channels: The last dimension of the tensor you want to apply pos␣
 ↪emb to.
        """
        super().__init__()
        self.org_channels = channels
        channels = int(math.ceil(channels / 6) * 2)
        if channels % 2:
            channels += 1
        self.channels = channels
        inv_freq = 1.0 / (10000 ** (torch.arange(0, channels, 2).float() /␣
 ↪channels))
        self.register_buffer("inv_freq", inv_freq)
        self.cached_penc = None

    def forward(self, tensor):
        """
        :param tensor: A 5d tensor of size (batch_size, x, y, z, ch)
        :return: Positional Encoding Matrix of size (batch_size, x, y, z, ch)
        """
        if len(tensor.shape) != 5:
            raise RuntimeError("The input tensor has to be 5d!")

        if self.cached_penc is not None and self.cached_penc.shape == tensor.
 ↪shape:
            return self.cached_penc

        self.cached_penc = None
        batch_size, orig_ch, z, x, y = tensor.shape
        pos_x = torch.arange(x, device=tensor.device).type(self.inv_freq.type())
        pos_y = torch.arange(y, device=tensor.device).type(self.inv_freq.type())
        pos_z = torch.arange(z, device=tensor.device).type(self.inv_freq.type())
        sin_inp_x = torch.einsum("i,j->ji", pos_x, self.inv_freq)
        sin_inp_y = torch.einsum("i,j->ji", pos_y, self.inv_freq)
        sin_inp_z = torch.einsum("i,j->ji", pos_z, self.inv_freq)
        emb_x = self._get_emb(sin_inp_x)[:,None,:,None].repeat(1,z, 1, y)
        emb_y = self._get_emb(sin_inp_y)[:,None,None,:].repeat(1,z, x, 1)
        emb_z = self._get_emb(sin_inp_z)[:,:,None,None].repeat(1,1, x, y)
        emb = torch.zeros((self.channels * 3, z, x, y), device=tensor.device).
 ↪type(
            tensor.type()
        )
        emb[:self.channels] = emb_x
        emb[self.channels : 2 * self.channels] = emb_y
        emb[2 * self.channels :] = emb_z

        self.cached_penc = emb[None, :orig_ch, :, :, :].repeat(batch_size, 1,␣
 ↪1, 1, 1)
        return self.cached_penc

    def _get_emb(self, sin_inp):
        """
        Gets a base embedding for one dimension with sin and cos intertwined
        """
        emb = torch.cat((sin_inp.sin(), sin_inp.cos()), dim=0)
        return emb


class Sinnembed_Position(nn.Module):
    """This module is used to embed the position of the input image.
    The same position will have the same embedding.

    >>> pos = torch.randn(B, 300, dim)
    >>> pos_emb = Sinnembed_Position(dim)(pos)
    >>> pos_emb.shape
    torch.Size([B, 300, dim * hidden_dim])

    """

    def __init__(self, dim: int = 3, hidden_dim: int = 128, position_format:␣
↪str = None, tensor_format: str = "BNC"):
        """Embed the sigmoid position with sine and cosine functions.

        Args:
            dim (int, optional): the dim of the model. Defaults to 3.
            hidden_dim (int, optional): hidden channel. Defaults to 128.
            position_format (str, optional): the format of the position tensor.␣
↪Defaults to "ZXY" for 3d, "XY" for 2d.
            tensor_format (str, optional): the format of the input tensor.␣
↪Defaults to "BNC".

        Outs:
            pos_emb (torch.Tensor): the position embedding tensor. shape (B, N,␣
↪hidden_dim * dim)
        """
        super().__init__()
        self.scale = 2 * math.pi
        self.dim = dim
        self.hidden_dim = hidden_dim
        dim_t = torch.arange(hidden_dim).float()
        dim_t = 10000 ** (dim * (dim_t // dim) / 128)
        self.register_buffer('dim_t', dim_t)

        self.tensor_format = tensor_format
        self.position_format = position_format or "ZXY"
        if self.dim == 2:
            self.position_format = position_format or "XY"
        elif self.dim == 3:
            self.position_format = position_format or "ZXY"
        elif self.dim == 1:
            self.position_format = position_format or "X"
        else:
            raise ValueError("dim must be 1, 2 or 3")

    def forward(self, pos_tensor: torch.Tensor) -> torch.Tensor:
        embed = []
        index = self.tensor_format.find("C")
        pos_tensor = pos_tensor.transpose(index, -1)
        for i, axis in enumerate(self.position_format):
            i_embed = pos_tensor[..., (i,)] * self.scale / self.dim_t
            i_embed[..., 0::2].sin_()
            i_embed[..., 1::2].cos_()
            i_embed.transpose_(index, -1)
            embed.append(i_embed)
        return torch.cat(embed, dim=index)

class EmbedMultiHeadAttention(nn.Module):
    """Embed Multi-Head Attention module, embed q, k, v after linear projection.
 ↪"""
    def __init__(self, embed_dim = 128, num_heads = 8, kdim: Optional[int] =␣
 ↪None, vdim: Optional[int] = None, dropout: float = 0.0, bias: bool = True,␣
 ↪tensor_format = "BNC"):
        super().__init__()
        self.num_heads = num_heads
        self.tensor_format = tensor_format
        self.dropout = dropout
        self.bias = bias
        self.embed_dim = embed_dim
        self.kdim = kdim or embed_dim
        self.vdim = vdim or embed_dim
        self.head_dim = embed_dim // num_heads
        assert self.head_dim * num_heads == self.embed_dim, "embed_dim must be␣
 ↪divisible by num_heads"

        self.q_proj = nn.Linear(self.embed_dim, self.embed_dim, bias = self.
 ↪bias)
        self.k_proj = nn.Linear(self.embed_dim, self.kdim, bias = self.bias)
        self.v_proj = nn.Linear(self.embed_dim, self.vdim, bias = self.bias)
        self.out_proj = nn.Linear(self.embed_dim, self.embed_dim, bias = self.
 ↪bias)

        self._reset_parameters()

    def _reset_parameters(self):
        nn.init.xavier_uniform_(self.q_proj.weight)
        nn.init.xavier_uniform_(self.k_proj.weight)
        nn.init.xavier_uniform_(self.v_proj.weight)
        nn.init.xavier_uniform_(self.out_proj.weight)

        nn.init.constant_(self.q_proj.bias, 0.)
        nn.init.constant_(self.out_proj.bias, 0.)

        nn.init.normal_(self.k_proj.bias)
        nn.init.normal_(self.v_proj.bias)

    def forward(self, q: torch.Tensor, k: torch.Tensor, v: torch.Tensor,
                q_pos: Optional[torch.Tensor] = None,
                k_pos: Optional[torch.Tensor] = None,
                v_pos: Optional[torch.Tensor] = None,
                q_emb: Optional[torch.Tensor] = None,
                k_emb: Optional[torch.Tensor] = None,
                v_emb: Optional[torch.Tensor] = None,
                mask: torch.Tensor = None):
        """_summary_

        Args:
            q (torch.Tensor): quary tensor, normally shape (B, N, C)
            k (torch.Tensor): key tensor, normally shape (B, N, C)
            v (torch.Tensor): value tensor, normally shape (B, N, C)
            q_emb (Optional[torch.Tensor]): embedding after projection.␣
↪Defaults to None.
            mask (_type_, optional): _description_. Defaults to None.

        Returns:
            _type_: _description_
        """
        if self.tensor_format != "BNC":
            q, k, v, q_pos, k_pos, v_pos, q_emb, k_emb, v_emb = [einops.
↪rearrange(x, (f"{' '.join(self.tensor_format)}", "B N C")) if x is not None␣
↪else None for x in (q, k, v, q_pos, k_pos, v_pos, q_emb, k_emb, v_emb)]
        q, k, v = self.q_proj(q), self.k_proj(k), self.v_proj(v)
        if q_pos is not None:
            q = q + q_pos
        if k_pos is not None:
            k = k + k_pos
        if v_pos is not None:
            v = v + v_pos

        # Turn the shape from normally "B N C" to "B H N D"
        q, k, v, q_emb, k_emb, v_emb = [einops.rearrange(x, "B N (H D) -> B H N␣
↪D", H = self.num_heads) if x is not None else None for x in (q, k, v, q_emb,␣
↪k_emb, v_emb)]

        if q_emb is not None:
            q = torch.cat([q, q_emb], dim = -1)
        if k_emb is not None:
            k = torch.cat([k, k_emb], dim = -1)
        if v_emb is not None:
            v = torch.cat([v, v_emb], dim = -1)
        out = F.scaled_dot_product_attention(q, k, v, attn_mask = mask,␣
 ↪dropout_p = self.dropout)
        out = einops.rearrange(out, f'B H N D -> {" ".join(self.tensor_format).
 ↪replace("C", "(H D)")}')

        return out

class CondTransformerDecoderLayer(nn.Module):
    def __init__(self,
                 dim = 3,
                 hidden_dim: int = 256,
                 dim_feedforward: int = 2048,
                 dropout: float = 0.1,
                 num_heads: int = 8,
                 norm: Callable = nn.LayerNorm,
                 return_intermediate: bool = False, is_first = False):
        super().__init__()
        self.is_first = is_first
        # Decoder Self-Attention
        self.sa_qpos_proj = nn.Linear(hidden_dim, hidden_dim)
        self.sa_kpos_proj = nn.Linear(hidden_dim, hidden_dim)
        self.self_attn = EmbedMultiHeadAttention(hidden_dim, num_heads,␣
 ↪dropout=dropout)
        self.cross_attn = EmbedMultiHeadAttention(embed_dim = hidden_dim,␣
 ↪num_heads = num_heads, vdim = hidden_dim, dropout=dropout)
        self.dropout = nn.Dropout(dropout)
        self.norm1 = norm(hidden_dim)
        self.norm2 = norm(hidden_dim)
        self.norm3 = norm(hidden_dim)
        self.ffn = MLP(hidden_dim, [dim_feedforward, hidden_dim], dropout =␣
 ↪dropout)

        # Decoder Cross-Attention
        if is_first:
            self.ca_qpos_proj = nn.Linear(hidden_dim, hidden_dim)
        self.ca_kpos_proj = nn.Linear(hidden_dim, hidden_dim)
        self.ca_qpos_sin_proj = nn.Linear(hidden_dim, hidden_dim)

    def forward(self, encoder_emb, decoder_emb, query_pos, pos, query_sin_emb):
        # Self-Attention
        q_pos = self.sa_qpos_proj(query_pos)
        k_pos = self.sa_kpos_proj(query_pos)
        x = self.self_attn(q = decoder_emb, k = decoder_emb, v = decoder_emb,␣
 ↪q_pos = q_pos, k_pos = k_pos)
        x = decoder_emb + self.dropout(x)
        x = self.norm1(x)
        # Cross Attention
        k_pos = self.ca_kpos_proj(pos)
        query_sin_emb = self.ca_qpos_sin_proj(query_sin_emb)
        if self.is_first:
            q_pos = self.ca_qpos_proj(query_pos)
            out = self.cross_attn(q = x, k = encoder_emb, v = encoder_emb,␣
 ↪q_pos = q_pos, k_pos = k_pos, q_emb = query_sin_emb, k_emb = k_pos)
        else:
            out = self.cross_attn(q = x, k = encoder_emb, v = encoder_emb,␣
 ↪q_emb = query_sin_emb, k_emb = k_pos)

        x = x + self.dropout(out)
        x = self.norm2(x)

        out = self.ffn(x)

        x = x + self.dropout(out)

        x = self.norm3(x)

        return x

class CondTransformerDecoder(nn.Module):
    def __init__(self, dim = 3, hidden_dim = 384, num_layers = 6,␣
 ↪return_intermediate = False, dropout = 0.1):
        super().__init__()
        self.layers = nn.ModuleList([])
        self.query_scale = MLP(hidden_dim, [hidden_dim, hidden_dim], dropout =␣
 ↪0)
        self.ref_point_head = MLP(hidden_dim, [hidden_dim, dim], dropout = 0)
        self.sin_emb = Sinnembed_Position(dim = dim, hidden_dim= hidden_dim //␣
 ↪dim,tensor_format = "BNC")
        self.norm = nn.LayerNorm(hidden_dim)
        for i in range(num_layers):
            self.layers.append(
                CondTransformerDecoderLayer(dim = dim, hidden_dim = hidden_dim,␣
 ↪dropout = dropout, is_first=(i==0)))

    def forward(self, encoder_emb: torch.Tensor, decoder_emb: torch.Tensor,␣
 ↪query_pos: torch.Tensor, pos: torch.Tensor, is_encode: bool = True):
        """_summary_

        Args:
            encoder_emb (torch.Tensor): shape: (B, N, C)
            decoder_emb (torch.Tensor): shape: (B, N, C)
            query_pos (torch.Tensor): learnable feature shape: (B, N, C) or (N,␣
 ↪C)
            pos (torch.Tensor): Sine spacial embeding shape: (B, N, C) or (N, C)

        Returns:
            _type_: _description_
        """
        x = decoder_emb
        if query_pos.dim() == 2:
            query_pos = query_pos.unsqueeze(0).repeat(x.shape[0], 1, 1)
        if pos.dim() == 2:
            pos = pos.unsqueeze(0).repeat(x.shape[0], 1, 1)

        ref_points = self.ref_point_head(decoder_emb).sigmoid()

        for i, layer in enumerate(self.layers):
            if i == 0:
                pos_trans = 1
            else:
                pos_trans = self.query_scale(x)

            query_sin_emb = self.sin_emb(ref_points)
            query_sin_emb = query_sin_emb * pos_trans
            if not is_encode:
                encoder_emb = x
            x = layer(encoder_emb = encoder_emb, decoder_emb = x, query_pos =␣
 ↪query_pos, pos = pos, query_sin_emb = query_sin_emb)

        x = self.norm(x)
        return x

from model.op import conv_nd
class CondTransformer(nn.Module):
    def __init__(self, hidden_dim = 192, dim_feedforward = 2048, image_dim =␣
 ↪256, dropout = 0.1, encoder_layers = 6, decoder_layers = 6, num_heads = 8,␣
 ↪num_queries = 300, dim = 3, target_size = (3.0, 25.0, 25.0), cls_num = 3,␣
 ↪feature_mapsize = (4, 16, 16)):
        super().__init__()
        self.hidden_dim = hidden_dim
        self.image_dim = image_dim
        self.feature_mapsize = feature_mapsize
        self.num_queries = num_queries
        self.cls_num = cls_num
        self.register_buffer("target_size",torch.tensor(target_size))
        self.encoder = nn.Sequential()
        for _ in range(encoder_layers):
            self.encoder.append(nn.TransformerEncoderLayer(hidden_dim, nhead = num_heads, dim_feedforward= dim_feedforward, dropout = dropout, batch_first = True))
        self.decoder = CondTransformerDecoder(dim = dim, hidden_dim = hidden_dim, num_layers = decoder_layers, dropout = dropout)

    def forward(self, encoder_emb: torch.Tensor, decoder_emb: torch.Tensor, query_pos: torch.Tensor, pos: torch.Tensor, is_encode: bool = False):
        if is_encode:
            encoder_emb = self.encoder(encoder_emb)

        decoder_emb = self.decoder(encoder_emb, decoder_emb, query_pos, pos, is_encode = is_encode)

        return decoder_emb

from model import EncoderUNetModel
class atomcDETR(nn.Module):
    def __init__(self,
                 fea_ch = 256,
                 tran_ch = 192,
                 real_size = (3.0, 25.0, 25.0),
                 feature_mapsize = (4, 16, 16),
                 num_queries = 300,
                 order = ("O", "H", "none")):
        super().__init__()
        self.register_buffer("real_size", torch.tensor(real_size))
        self.register_buffer("feature_mapsize", torch.tensor(feature_mapsize))
        self.fea_ch = fea_ch
        self.tran_ch = tran_ch
        self.num_queries = num_queries
        self.order = order
        self.cls_num = len(order)

        self.spac_emb = Sinnembed_SpacialEncoding(self.fea_ch)
        self.spac_proj = conv_nd(3, self.fea_ch, self.tran_ch, kernel_size = 1)
        self.label_emb = nn.Linear(self.cls_num + 3, self.tran_ch)

        self.backbone = EncoderUNetModel(image_size=(16, 128, 128),
                                         in_channels = 1,
                                         model_channels = 32,
                                         out_channels = 256,
                                         num_res_blocks= 2,
                                         channel_mult=(1,2,4,8),
                                         attention_resolutions=tuple(),
                                         dims = 3,
                                         pool = "none"
```

```
                                        )
        self.transformer = CondTransformer(hidden_dim = 192,
                                           dim_feedforward = 2048,
                                           image_dim = 256,
                                        )

        self.cls = nn.Sequential(
            MLP(self.tran_ch, [self.tran_ch, self.cls_num]),
            nn.Softmax(dim = 2))

        self.reg = nn.Sequential(
            MLP(self.tran_ch, [self.tran_ch, 3]),
            nn.Sigmoid())

    def forward(self, img: Optional[torch.Tensor] = None, label:␣
↪Optional[Tuple[torch.Tensor]] = None):
        """_summary_
        Args:
            label (Tuple[torch.Tensor]): providing the label to predict, should␣
↪be formatted as (B, N), (B, N, C), C is Z, X, Y. and Z,X,Y are in real_size
            feature_map (torch.Tensor): _description_. Defaults to None.
        """
        assert (label or img) is not None, "Either label or img should be␣
↪provided"
        if label is None:
            # give many random atoms
            B = img.shape[0]
            CLS = torch.randint(0, self.cls_num, (B, self.num_queries), dtype =␣
↪torch.long, device = self.real_size.device)
            REG = torch.empty(B, self.num_queries, 3, device = self.real_size.
↪device).uniform_(0, 1) * self.real_size
        else:
            B = CLS.shape[0]
            CLS, REG = label

        object_quary = self.label_emb(torch.cat([F.one_hot(CLS).float(), REG],␣
↪dim = 2)) # B x N x C
        spacial_emb = self.spac_emb(torch.empty(B, self.tran_ch, *self.
↪feature_mapsize, device = self.real_size.device)).flatten(2).permute(0, 2,␣
↪1) # B x N' x C

        if img is None:
            encoder_emb = object_quary
        else:
            encoder_emb = self.spac_proj(self.backbone(img)).flatten(2).
↪permute(0, 2, 1) # B x N' x C
```

```
        decoder_emb = self.transformer(encoder_emb + spacial_emb, object_quary,␣
 ↪object_quary, spacial_emb, is_encode = img is not None)

        pred_cls = self.cls(decoder_emb)
        pred_reg = (self.reg(decoder_emb) * 1.2 - 0.1) * self.real_size
        return pred_cls, pred_reg

img = torch.randn(1, 1, 16, 128, 128).cuda()
from model.utils import structure
r = atomcDETR().cuda()
pd_c, pd_o = r(img)
```

[ ]:

```
from datasets import AFMDataset_DETR
from torch.utils.data import DataLoader
dataloader = AFMDataset_DETR("../data/bulkice", file_list = "train.filelist")
dl = DataLoader(dataloader, batch_size = 1, shuffle = True)
it_dl = iter(dl)
img, (gt_o, gt_c), fn = next(it_dl)
gt_o, gt_c = gt_o.cuda(), gt_c.cuda()
print(gt_o.shape, gt_c.shape)
```

[ ]:

```
import numpy as np
import torch
from typing import Dict, Tuple, Union, Optional
import cv2
def CZXY2POS(x: torch.Tensor, order = ("none", "O", "H")):
    """x is in CZXY format : N x (cls_num, z, x, y) or (c1, c2, c3, z, x, y))"""
    if x.shape[1] != 4:
        assert x.shape[1] == len(order) + 3, "x should be in CZXY format"
        cls = x[:, :len(order)].argmax(dim = 1, keepdim = True)
        x = torch.cat([cls, x[:, len(order):]], dim = 1)
    dic = {}
    for i, o in enumerate(order):
        if "NONE" in o.upper():
            continue
        mask = x[:,0] == i
        dic[o] = x[mask, 1:]
    return dic

def plotAtom(bg: torch.Tensor | np.ndarray, points_dict: Dict[str, torch.
 ↪Tensor] | torch.Tensor, order = ("none", "O", "H"), real_size = (3.0, 25.0,␣
 ↪25.0), color = {"O": (255, 0, 0), "H": (255, 255, 255)}, radius = {"O": 0.7,␣
 ↪"H": 0.4}):
    """bg should be HW format, points_dict should be in CZXY format"""
    if isinstance(bg, torch.Tensor):
        bg = bg.cpu().numpy()
    if isinstance(points_dict, torch.Tensor):
        points_dict = CZXY2POS(points_dict, order)
    bg = bg.copy()
    real_size = np.array(real_size)
    zoom = (0, *bg.shape[:2]) / real_size
    points_dict = {k: v.detach().cpu().numpy() * zoom for k, v in points_dict.
 ↪items()}
    for elem, pos in points_dict.items():
        pos = pos.astype(np.uint8)[...,:0:-1] # y x format
        c = color[elem]
        r = int(radius[elem] * zoom[1])
        for p in pos:
            bg = cv2.circle(bg, p, r, c, -1)
            bg = cv2.circle(bg, p, r, (255, 255, 255), 1)
    return bg

test_atom = torch.cat((gt_c[0], gt_o[0]), dim = 1)
img = plotAtom(np.zeros((300,300, 3), dtype = np.uint8), test_atom)
import matplotlib.pyplot as plt
plt.imshow(img)
```

[ ]:
```
from scipy.optimize import linear_sum_assignment
import torch
from torch import nn
class RCNNLoss(nn.Module):
    def __init__(self, dis_eff = 1, con_eff = 1, none_dist = 0, real_box = (3.,␣
 ↪25., 25.), alpha = 0.25, gamma = 2):
        super().__init__()
        self.dis_eff = dis_eff
        self.con_eff = con_eff
        self.none_dist = none_dist
        self.register_buffer('real_box', torch.tensor(real_box))
        self.register_buffer('unit_length',torch.pow(torch.prod(self.real_box),␣
 ↪1/3))
        self.alpha = alpha
        self.gamma = gamma


    def forward(self,
                pd_bbox,        #bbox: (B, N, 3)     format: (Z, X, Y)
                pd_onehots,        #clss: (B, N, 3)     format: (none, O, H)
                gt_bbox,    #gt_bbox: (B, N, 3)   format: (Z, X, Y)
                gt_clss     #gt_clss: (B, N, 1)   format: (index)
                ):
        B, N, D = pd_bbox.shape
        # distance loss
        pd_bbox = pd_bbox
        gt_bbox = gt_bbox
        pd_bbox = pd_bbox * self.real_box
        gt_bbox = gt_bbox * self.real_box
        DIS_met = torch.cdist(pd_bbox, gt_bbox, p = 2)
        DIS_met = DIS_met / self.unit_length

        # confidence loss
        pd_clss = torch.argmax(pd_onehots, dim = -1, keepdim= True)            ␣
 ↪  # (B, N, 1)
        pd_mask = pd_clss != 0
        gt_mask = gt_clss != 0

        #DIS_cost
        # all None to None distance are set to 0
        mask = torch.einsum('B N I, B M I -> B N M', ~pd_mask, ~gt_mask)
        DIS_met = torch.where(mask, 0, DIS_met)
        # all predict to None distance are set to none_dist
        #mask = torch.einsum('B N I, B M I -> B N M', pd_mask, gt_mask)
        #DIS_met = torch.where(~mask, += self.none_dist                         ␣
 ↪    # can change

        pd_clss = pd_clss
        gt_clss = gt_clss.view(B, 1, N).repeat(1, N, 1) # (B, N, N)
        CON_met = torch.gather(pd_onehots, 2, gt_clss)    # (B, N, N)
        with torch.no_grad():
            cal_cost = (DIS_met + CON_met).detach().cpu().numpy()
            match_result = tuple(linear_sum_assignment(cost) for cost in␣
 ↪cal_cost)
        # CLS_cost = positive_loss - negative_loss
        CLS_cost = - ((1 - CON_met) ** 2) * (CON_met + 1e-5).log()
        # cost metrix
        COST = CLS_cost * self.con_eff + DIS_met * self.dis_eff / self.
 ↪unit_length
        # find match

        LOSS = []
        for b, (i, j) in enumerate(match_result):
            LOSS.append(COST[b, i, j].mean())
        LOSS = torch.stack(LOSS).mean()
        return LOSS, match_result

loss = RCNNLoss().cuda()
```

[ ]:

```
opt = torch.optim.Adam(r.parameters(), lr = 1e-5)
for i, (img, (gt_o, gt_c), fn) in enumerate(it_dl):
    img, gt_o, gt_c = img.cuda(), gt_o.cuda(), gt_c.cuda()
    pd_c, pd_o = r(img)
    opt.zero_grad()
    L, M = loss(pd_o, pd_c, gt_o, gt_c)
    L.backward()
    g = torch.nn.utils.clip_grad_norm_(r.parameters(), 10, error_if_nonfinite=␣
 ↪True)
    print(L, g, end = "")
    opt.step()
    if i % 10 == 0:
        bg = img[0,(0,0,0),0].permute(1,2,0).detach().cpu().numpy() * 255
        bg = bg.astype(np.uint8)
        gt_atom = torch.cat((gt_c[0], gt_o[0]), dim = 1)
        img = plotAtom(bg, gt_atom)
        pd_atom = torch.cat((pd_c[0], pd_o[0]), dim = 1)
        img2 = plotAtom(bg, pd_atom)
        fig, (ax1, ax2) = plt.subplots(1, 2)
        ax1.imshow(img)
        ax2.imshow(img2)
        plt.show()
```

[ ]:
```
print(cv2.circle(bg.copy(), (50,50), 1, (255,255,255), -1))
```

[ ]:
```
import torch
from torch import nn
from torch.nn import functional as F
import math
import matplotlib.pyplot as plt
import numpy as np

class Sinnembed_SpacialEncoding(nn.Module):
    def __init__(self, channels):
        """
        :param channels: The last dimension of the tensor you want to apply pos␣
 ↪emb to.
        """
        super().__init__()
        self.org_channels = channels
        channels = int(math.ceil(channels / 6) * 2)
        if channels % 2:
            channels += 1
        self.channels = channels
        inv_freq = 1.0 / (10000 ** (torch.arange(0, channels, 2).float() /␣
 ↪channels))
        self.register_buffer("inv_freq", inv_freq)
        self.cached_penc = None

    def forward(self, tensor):
        """
        :param tensor: A 5d tensor of size (batch_size, x, y, z, ch)
        :return: Positional Encoding Matrix of size (batch_size, x, y, z, ch)
        """
        if len(tensor.shape) != 5:
            raise RuntimeError("The input tensor has to be 5d!")

        if self.cached_penc is not None and self.cached_penc.shape == tensor.
↪shape:
            return self.cached_penc

        self.cached_penc = None
        batch_size, orig_ch, z, x, y = tensor.shape
        pos_x = torch.arange(x, device=tensor.device).type(self.inv_freq.type())
        pos_y = torch.arange(y, device=tensor.device).type(self.inv_freq.type())
        pos_z = torch.arange(z, device=tensor.device).type(self.inv_freq.type())
        sin_inp_x = torch.einsum("i,j->ji", pos_x, self.inv_freq)
        sin_inp_y = torch.einsum("i,j->ji", pos_y, self.inv_freq)
        sin_inp_z = torch.einsum("i,j->ji", pos_z, self.inv_freq)
        emb_x = self._get_emb(sin_inp_x)[:,None,:,None].repeat(1,z, 1, y)
        emb_y = self._get_emb(sin_inp_y)[:,None,None,:].repeat(1,z, x, 1)
        emb_z = self._get_emb(sin_inp_z)[:,:,None,None].repeat(1,1, x, y)
        emb = torch.zeros((self.channels * 3, z, x, y), device=tensor.device).
↪type(
            tensor.type()
        )
        emb[:self.channels] = emb_x
        emb[self.channels : 2 * self.channels] = emb_y
        emb[2 * self.channels :] = emb_z

        self.cached_penc = emb[None, :orig_ch, :, :, :].repeat(batch_size, 1,␣
↪1, 1, 1)
        return self.cached_penc,emb_x

    def _get_emb(self, sin_inp):
        """
        Gets a base embedding for one dimension with sin and cos intertwined
        """
        emb = torch.cat((sin_inp.sin(), sin_inp.cos()), dim=0)
        return emb

class PositionalEncoding3D(nn.Module):
    def __init__(self, channels):
        """
        :param channels: The last dimension of the tensor you want to apply pos␣
↪emb to.
        """
        super(PositionalEncoding3D, self).__init__()
        self.org_channels = channels
        channels = int(np.ceil(channels / 6) * 2)
        if channels % 2:
            channels += 1
        self.channels = channels
        inv_freq = 1.0 / (10000 ** (torch.arange(0, channels, 2).float() /␣
 ↪channels))
        self.register_buffer("inv_freq", inv_freq)
        self.cached_penc = None

    def forward(self, tensor):
        """
        :param tensor: A 5d tensor of size (batch_size, x, y, z, ch)
        :return: Positional Encoding Matrix of size (batch_size, x, y, z, ch)
        """
        if len(tensor.shape) != 5:
            raise RuntimeError("The input tensor has to be 5d!")

        if self.cached_penc is not None and self.cached_penc.shape == tensor.
 ↪shape:
            return self.cached_penc

        self.cached_penc = None
        batch_size, x, y, z, orig_ch = tensor.shape
        pos_x = torch.arange(x, device=tensor.device).type(self.inv_freq.type())
        pos_y = torch.arange(y, device=tensor.device).type(self.inv_freq.type())
        pos_z = torch.arange(z, device=tensor.device).type(self.inv_freq.type())
        sin_inp_x = torch.einsum("i,j->ij", pos_x, self.inv_freq)
        sin_inp_y = torch.einsum("i,j->ij", pos_y, self.inv_freq)
        sin_inp_z = torch.einsum("i,j->ij", pos_z, self.inv_freq)
        emb_x = get_emb(sin_inp_x).unsqueeze(1).unsqueeze(1)
        emb_y = get_emb(sin_inp_y).unsqueeze(1)
        emb_z = get_emb(sin_inp_z)
        emb = torch.zeros((x, y, z, self.channels * 3), device=tensor.device).
 ↪type(
            tensor.type()
        )
        emb[:, :, :, : self.channels] = emb_x
        emb[:, :, :, self.channels : 2 * self.channels] = emb_y
        emb[:, :, :, 2 * self.channels :] = emb_z

        self.cached_penc = emb[None, :, :, :, :orig_ch].repeat(batch_size, 1,␣
 ↪1, 1, 1)
        return self.cached_penc

def get_emb(sin_inp):
    """
    Gets a base embedding for one dimension with sin and cos intertwined
    """
    emb = torch.stack((sin_inp.sin(), sin_inp.cos()), dim=-1)
    return torch.flatten(emb, -2, -1)


class PositionalEncodingPermute3D(nn.Module):
    def __init__(self, channels):
        """
        Accepts (batchsize, ch, x, y, z) instead of (batchsize, x, y, z, ch)
        """
        super(PositionalEncodingPermute3D, self).__init__()
        self.penc = PositionalEncoding3D(channels)

    def forward(self, tensor):
        tensor = tensor.permute(0, 2, 3, 4, 1)
        enc = self.penc(tensor)
        return enc.permute(0, 4, 1, 2, 3)

    @property
    def org_channels(self):
        return self.penc.org_channels

a = PositionalEncodingPermute3D(256)
c = Sinnembed_SpacialEncoding(256)
b = torch.zeros((1, 256, 1, 256, 256))
e= a(b)
r, _ = c(b)
fig, (ax1, ax2) = plt.subplots(1 ,2)
ax1.imshow(e[0,100,0])
ax2.imshow(r[0,100,0])
```

[ ]:

```
import torch
a = torch.ones(1,10)
b = torch.zeros(1,10)
c =
```

[ ]:

```
from model.op import MultiHeadAttentionWithNoLinear
import torch
from torch import nn
a = nn.MultiheadAttention(24, 24, 0, kdim =24, vdim = 16, batch_first=True)
b = MultiHeadAttentionWithNoLinear()
q = torch.rand(1, 10, 24)
k = torch.rand(1, 30, 24)
v = torch.rand(1, 30, 16)
a(q,k,v)[0].shape
b(q,k,v).shape
```

[ ]:

```
from copy import deepcopy
deepcopy(None)
```

[ ]:

```
import torch
a = torch.arange(0, 10)
a = a.unsqueeze(0).repeat(10,1)
c = torch.tensor(((1,2), (3,4)))
a[(0,1), c]

a = torch.randn(6, 10)
b = torch.randint(0, 10, (6, 10))
c = torch.arange(0, 10).unsqueeze(0).repeat(6,1)
print(c)
print(a)
a.sparse_mask
e = torch.randn(10)
e[c]
```

[ ]:

```
import torch
from typing import Iterable, Callable
def apply(x: Iterable, func: Callable):
    for i in x:
        func(i)
    return x
a = torch.randn(1, 3)
b = torch.randn(1, 3)
c = torch._nested_tensor_from_tensor_list([a,b])
d = torch.ones(3) * 10
print(c)
apply(c, lambda x: x.div_(d))
print(c)
```

[ ]:

```
import torch
a = torch.arange(0, 10)
b = a.unsqueeze(0).repeat(6,1)
b = b.transpose(0,1)
print(b.reshape(10,6).reshape(60))
c = a.repeat(6)
print(c[1])
# print(b.shape)
# print(b.view(60))
# print(a.repeat(6))
```

[ ]:

```
[i for i in range(10) for _ in range(6)]
```

[ ]:

```
from torch import nn
nn.TransformerDecoder
```

[ ]:

```
from torch import nn
import torch
class Sinnembed_Position(nn.Module):
    """This module is used to embed the position of the input image.

    The same position will have the same embedding.

    >>> pos = torch.randn(B, 300, dim)
    >>> pos_emb = Sinnembed_Position(dim)(pos)
    >>> pos_emb.shape
    torch.Size([B, 300, dim * hidden_dim])

    """

    def __init__(self, dim: int = 3,
                 hidden_dim: int = 128,
                 position_format: str = None,
                 temperature: float = 10000,
                 normalize: bool = False,
                 eps = 1e-6,
                 scale: float = None,
                 ):
        """Embed the sigmoid position with sine and cosine functions.

        Args:
            dim (int, optional): the dim of the model. Defaults to 3.
            hidden_dim (int, optional): hidden channel. Defaults to 128.
            position_format (str, optional): the format of the position tensor.␣
↪Defaults to "ZXY" for 3d, "XY" for 2d.
            tensor_format (str, optional): the format of the input tensor.␣
↪Defaults to "BNC".

        Outs:
            pos_emb (torch.Tensor): the position embedding tensor. shape (B, N,␣
↪hidden_dim * dim)
        """
        super().__init__()
        self.scale = scale or 2 * math.pi
        self.temperature = temperature
        self.normalize = normalize
        self.dim = dim
        self.hidden_dim = hidden_dim
        self.eps = eps
        dim_t = torch.arange(hidden_dim).float()
        dim_t = temperature ** (dim * (dim_t // dim) / hidden_dim)
        self.register_buffer('dim_t', dim_t)
    def forward(self, pos_tensor: torch.Tensor) -> torch.Tensor:
        embed = []
        B, N, axis = pos_tensor.shape
        axis_embed = pos_tensor * self.scale
        i_embed = pos_tensor[..., (i,)] * self.scale / self.dim_t
        i_embed[..., 0::2].sin_()
        i_embed[..., 1::2].cos_()
        i_embed.transpose_(index, -1)
        embed.append(i_embed)
        return torch.cat(embed, dim=index)
```

[ ]:

```
import torch
a = torch.randn(1, 100, 3, 1)
dim = torch.arange(10)
b = torch.randn(1, 100)
(a * dim).shape

a = torch.arange(3).unsqueeze(0)
print(a.shape)

from model.op import Sinnembed_Position
SP = Sinnembed_Position(hidden_dim=256)
pos = torch.arange(0, 300)
pos = pos.unsqueeze(0).unsqueeze(-1).repeat(3, 1, 20)
print(pos.shape)

import matplotlib.pyplot as plt
emb = SP(pos)
plt.imshow(emb[0])
```

[ ]:

```
import torch
a = torch.zeros(3, 3, 3)
b = torch.ones(3, 1, 3)
a + b
```

[ ]:

```
import torch
a = torch.ones(10,10).float()
b = torch.zeros(1, 10).float()
torch.cat([a,b], dim=-1).shape
```

[ ]:

```
import torch
a = torch.meshgrid(torch.arange(0, 10), torch.arange(0, 10), torch.arange(0,␣
 ↪10), indexing='ij')
a = torch.stack(a, dim=-1)
b = torch.meshgrid(torch.arange(0, 10), torch.arange(0, 10), torch.arange(0,␣
 ↪10), indexing='xy')
b = torch.stack(b, dim=-1)
```

[ ]:

```
from model.op import sinnembed_position
import matplotlib.pyplot as plt
import torch
from einops import pack, rearrange
a = torch.rand((100, 3))
b = sinnembed_position(a, 128)
fea = torch.randn_like(b)
fig, axs = plt.subplots(1, 4)
axs[0].imshow(fea)
axs[1].imshow(b)
axs[2].imshow(torch.cat([fea, b], dim=-2).view(100, 256))
out = torch.cat([fea, b], dim=-1)
print(out.shape)
out = rearrange(out, "h (w c) -> h (c w)", w = 2)
axs[3].imshow(out)
plt.show()
```

[ ]:

```
a = torch.randn(2, 10)
b = torch.randn(1, 10)
pack([a, b], pattern = "n *")
```

[ ]:

```
import os
import shutil
loade = "/home/supercgor/gitfile/data/middle/pls2/bulk_slice_box"
path = "/home/supercgor/gitfile/data/middle/datas/pls2/bulk_slice_box/"
for p in os.listdir(loade):
    if p == "cut_info.txt":
        continue
    for i in os.listdir(f"{path}/{p}"):
        j = 4 - int(i.split(".")[0])
        shutil.copyfile(f"{loade}/{p}/{i}", f"{path}/{p}/{j}.png")
        print( f"{path}/{p}/{j}.png")
        #os.rename(f"{path}/{p}/{i}", f"{path}/{p}/{j}.png")
```

[ ]:

```
from model.unet import UNetModel
a = UNetModel(image_size=(16, 128, 128),
              in_channels=1,
              model_channels=32,
              out_channels=32,
              num_res_blocks=1,
              attention_resolutions=(4, 8),
              dropout=0,
              channel_mult=(1, 2, 4, 8),
              dims = 3,
              time_embed=None)
```

[ ]:

```
import torch
test = torch.randn(1, 1, 16, 128, 128)
out = a(test, debug = True)
```

```
shapes [torch.Size([1, 1, 16, 128, 128]), torch.Size([1, 32, 16, 128, 128]),
torch.Size([1, 32, 16, 128, 128]), torch.Size([1, 64, 8, 64, 64]),
torch.Size([1, 64, 8, 64, 64]), torch.Size([1, 128, 4, 32, 32]), torch.Size([1,
128, 4, 32, 32]), torch.Size([1, 256, 2, 16, 16]), torch.Size([1, 256, 2, 16,
16]), torch.Size([1, 256, 2, 16, 16]), torch.Size([1, 256, 2, 16, 16]),
torch.Size([1, 128, 4, 32, 32]), torch.Size([1, 128, 4, 32, 32]), torch.Size([1,
64, 8, 64, 64]), torch.Size([1, 64, 8, 64, 64]), torch.Size([1, 32, 16, 128,
128]), torch.Size([1, 32, 16, 128, 128]), torch.Size([1, 32, 16, 128, 128])]
```

[ ]:

```
from model.utils import structure
print(out.shape)
structure(a)
```

```
torch.Size([1, 32, 16, 128, 128])
--------------------------------------------------------------------------------
--------------------
|                     weight name                    |          weight shape
|   number   |
--------------------------------------------------------------------------------
--------------------
| input_blocks.0.conv.weight                         | 32, 1, 3, 3, 3
| 864        |
| input_blocks.0.conv.bias                           | 32,
| 32         |
| input_blocks.1.res0.in_layers.norm.weight          | 32,
| 32         |
| input_blocks.1.res0.in_layers.norm.bias            | 32,
| 32         |
| input_blocks.1.res0.in_layers.conv.weight          | 32, 32, 3, 3, 3
| 27648      |
| input_blocks.1.res0.in_layers.conv.bias            | 32,
| 32         |
| input_blocks.1.res0.out_layers.norm.weight         | 32,
| 32         |
| input_blocks.1.res0.out_layers.norm.bias           | 32,
| 32         |
| input_blocks.1.res0.out_layers.conv.weight         | 32, 32, 3, 3, 3
| 27648      |
| input_blocks.1.res0.out_layers.conv.bias           | 32,
| 32         |
| input_blocks.2.down_conv.op.weight                 | 64, 32, 3, 3, 3
| 55296      |
| input_blocks.2.down_conv.op.bias                   | 64,
| 64         |
| input_blocks.3.res0.in_layers.norm.weight          | 64,
| 64         |
| input_blocks.3.res0.in_layers.norm.bias            | 64,
| 64         |
| input_blocks.3.res0.in_layers.conv.weight          | 64, 64, 3, 3, 3
| 110592     |
| input_blocks.3.res0.in_layers.conv.bias            | 64,
| 64         |
| input_blocks.3.res0.out_layers.norm.weight         | 64,
| 64         |
| input_blocks.3.res0.out_layers.norm.bias           | 64,
| 64         |
| input_blocks.3.res0.out_layers.conv.weight         | 64, 64, 3, 3, 3
| 110592     |
| input_blocks.3.res0.out_layers.conv.bias           | 64,
| 64         |
| input_blocks.4.down_conv.op.weight                 | 128, 64, 3, 3, 3
| 221184     |
| input_blocks.4.down_conv.op.bias                   | 128,
| 128        |
| input_blocks.5.res0.in_layers.norm.weight          | 128,
| 128        |
| input_blocks.5.res0.in_layers.norm.bias            | 128,
| 128        |
| input_blocks.5.res0.in_layers.conv.weight          | 128, 128, 3, 3, 3
| 442368     |
| input_blocks.5.res0.in_layers.conv.bias            | 128,
| 128        |
| input_blocks.5.res0.out_layers.norm.weight         | 128,
| 128        |
| input_blocks.5.res0.out_layers.norm.bias           | 128,
| 128        |
| input_blocks.5.res0.out_layers.conv.weight         | 128, 128, 3, 3, 3
| 442368     |
| input_blocks.5.res0.out_layers.conv.bias           | 128,
| 128        |
| input_blocks.5.attn0.norm.weight                   | 128,
| 128        |
| input_blocks.5.attn0.norm.bias                     | 128,
| 128        |
| input_blocks.5.attn0.qkv.weight                    | 384, 128, 1
| 49152      |
| input_blocks.5.attn0.qkv.bias                      | 384,
| 384        |
| input_blocks.5.attn0.proj_out.weight               | 128, 128, 1
| 16384      |
| input_blocks.5.attn0.proj_out.bias                 | 128,
| 128        |
| input_blocks.6.down_conv.op.weight                 | 256, 128, 3, 3, 3
| 884736     |
| input_blocks.6.down_conv.op.bias                   | 256,
| 256        |
| input_blocks.7.res0.in_layers.norm.weight          | 256,
| 256        |
| input_blocks.7.res0.in_layers.norm.bias            | 256,
| 256        |
| input_blocks.7.res0.in_layers.conv.weight          | 256, 256, 3, 3, 3
| 1769472    |
| input_blocks.7.res0.in_layers.conv.bias            | 256,
| 256        |
| input_blocks.7.res0.out_layers.norm.weight         | 256,
| 256        |
| input_blocks.7.res0.out_layers.norm.bias           | 256,
| 256        |
| input_blocks.7.res0.out_layers.conv.weight         | 256, 256, 3, 3, 3
| 1769472    |
| input_blocks.7.res0.out_layers.conv.bias           | 256,
| 256        |
| input_blocks.7.attn0.norm.weight                   | 256,
| 256        |
| input_blocks.7.attn0.norm.bias                     | 256,
| 256        |
| input_blocks.7.attn0.qkv.weight                    | 768, 256, 1
| 196608     |
| input_blocks.7.attn0.qkv.bias                      | 768,
| 768        |
| input_blocks.7.attn0.proj_out.weight               | 256, 256, 1
| 65536      |
| input_blocks.7.attn0.proj_out.bias                 | 256,
| 256        |
| middle_block.res0.in_layers.norm.weight            | 256,
| 256        |
| middle_block.res0.in_layers.norm.bias              | 256,
| 256        |
| middle_block.res0.in_layers.conv.weight            | 256, 256, 3, 3, 3
| 1769472    |
| middle_block.res0.in_layers.conv.bias              | 256,
| 256        |
| middle_block.res0.out_layers.norm.weight           | 256,
| 256        |
| middle_block.res0.out_layers.norm.bias             | 256,
| 256        |
| middle_block.res0.out_layers.conv.weight           | 256, 256, 3, 3, 3
| 1769472    |
| middle_block.res0.out_layers.conv.bias             | 256,
| 256        |
| middle_block.attn.norm.weight                      | 256,
| 256        |
| middle_block.attn.norm.bias                        | 256,
| 256        |
| middle_block.attn.qkv.weight                       | 768, 256, 1
| 196608     |
| middle_block.attn.qkv.bias                         | 768,
| 768        |
| middle_block.attn.proj_out.weight                  | 256, 256, 1
| 65536      |
| middle_block.attn.proj_out.bias                    | 256,
| 256        |
| output_blocks.0.res0.in_layers.norm.weight         | 512,
| 512        |
| output_blocks.0.res0.in_layers.norm.bias           | 512,
| 512        |
| output_blocks.0.res0.in_layers.conv.weight         | 256, 512, 3, 3, 3
| 3538944    |
| output_blocks.0.res0.in_layers.conv.bias           | 256,
| 256        |
| output_blocks.0.res0.out_layers.norm.weight        | 256,
| 256        |
| output_blocks.0.res0.out_layers.norm.bias          | 256,
| 256        |
| output_blocks.0.res0.out_layers.conv.weight        | 256, 256, 3, 3, 3
| 1769472    |
| output_blocks.0.res0.out_layers.conv.bias          | 256,
| 256        |
| output_blocks.0.res0.skip_connection.weight        | 256, 512, 1, 1, 1
| 131072     |
| output_blocks.0.res0.skip_connection.bias          | 256,
| 256        |
| output_blocks.0.attn0.norm.weight                  | 256,
| 256        |
| output_blocks.0.attn0.norm.bias                    | 256,
| 256        |
| output_blocks.0.attn0.qkv.weight                   | 768, 256, 1
| 196608     |
| output_blocks.0.attn0.qkv.bias                     | 768,
| 768        |
| output_blocks.0.attn0.proj_out.weight              | 256, 256, 1
| 65536      |
| output_blocks.0.attn0.proj_out.bias                | 256,
| 256        |
| output_blocks.1.res1.in_layers.norm.weight         | 512,
| 512        |
| output_blocks.1.res1.in_layers.norm.bias           | 512,
| 512        |
| output_blocks.1.res1.in_layers.conv.weight         | 256, 512, 3, 3, 3
| 3538944    |
| output_blocks.1.res1.in_layers.conv.bias           | 256,
| 256        |
| output_blocks.1.res1.out_layers.norm.weight        | 256,
| 256        |
| output_blocks.1.res1.out_layers.norm.bias          | 256,
| 256        |
| output_blocks.1.res1.out_layers.conv.weight        | 256, 256, 3, 3, 3
| 1769472    |
| output_blocks.1.res1.out_layers.conv.bias          | 256,
| 256        |
| output_blocks.1.res1.skip_connection.weight        | 256, 512, 1, 1, 1
| 131072     |
| output_blocks.1.res1.skip_connection.bias          | 256,
| 256        |
| output_blocks.1.attn1.norm.weight                  | 256,
| 256        |
| output_blocks.1.attn1.norm.bias                    | 256,
| 256        |
| output_blocks.1.attn1.qkv.weight                   | 768, 256, 1
| 196608     |
| output_blocks.1.attn1.qkv.bias                     | 768,
| 768        |
| output_blocks.1.attn1.proj_out.weight              | 256, 256, 1
| 65536      |
| output_blocks.1.attn1.proj_out.bias                | 256,
| 256        |
| output_blocks.1.up.conv.weight                     | 128, 256, 3, 3, 3
| 884736     |
| output_blocks.1.up.conv.bias                       | 128,
| 128        |
| output_blocks.2.res0.in_layers.norm.weight         | 256,
| 256        |
| output_blocks.2.res0.in_layers.norm.bias           | 256,
| 256        |
| output_blocks.2.res0.in_layers.conv.weight         | 128, 256, 3, 3, 3
| 884736     |
| output_blocks.2.res0.in_layers.conv.bias           | 128,
| 128        |
| output_blocks.2.res0.out_layers.norm.weight        | 128,
| 128        |
| output_blocks.2.res0.out_layers.norm.bias          | 128,
| 128        |
| output_blocks.2.res0.out_layers.conv.weight        | 128, 128, 3, 3, 3
| 442368     |
| output_blocks.2.res0.out_layers.conv.bias          | 128,
| 128        |
| output_blocks.2.res0.skip_connection.weight        | 128, 256, 1, 1, 1
| 32768      |
| output_blocks.2.res0.skip_connection.bias          | 128,
| 128        |
| output_blocks.2.attn0.norm.weight                  | 128,
| 128        |
| output_blocks.2.attn0.norm.bias                    | 128,
| 128        |
| output_blocks.2.attn0.qkv.weight                   | 384, 128, 1
| 49152      |
| output_blocks.2.attn0.qkv.bias                     | 384,
| 384        |
| output_blocks.2.attn0.proj_out.weight              | 128, 128, 1
| 16384      |
| output_blocks.2.attn0.proj_out.bias                | 128,
| 128        |
| output_blocks.3.res1.in_layers.norm.weight         | 256,
| 256        |
| output_blocks.3.res1.in_layers.norm.bias           | 256,
| 256        |
| output_blocks.3.res1.in_layers.conv.weight         | 128, 256, 3, 3, 3
| 884736     |
| output_blocks.3.res1.in_layers.conv.bias           | 128,
| 128        |
| output_blocks.3.res1.out_layers.norm.weight        | 128,
| 128        |
| output_blocks.3.res1.out_layers.norm.bias          | 128,
| 128        |
| output_blocks.3.res1.out_layers.conv.weight        | 128, 128, 3, 3, 3
| 442368     |
| output_blocks.3.res1.out_layers.conv.bias          | 128,
| 128        |
| output_blocks.3.res1.skip_connection.weight        | 128, 256, 1, 1, 1
| 32768      |
| output_blocks.3.res1.skip_connection.bias          | 128,
| 128        |
| output_blocks.3.attn1.norm.weight                  | 128,
| 128        |
| output_blocks.3.attn1.norm.bias                    | 128,
| 128        |
| output_blocks.3.attn1.qkv.weight                   | 384, 128, 1
| 49152      |
| output_blocks.3.attn1.qkv.bias                     | 384,
| 384        |
| output_blocks.3.attn1.proj_out.weight              | 128, 128, 1
| 16384      |
| output_blocks.3.attn1.proj_out.bias                | 128,
| 128        |
| output_blocks.3.up.conv.weight                     | 64, 128, 3, 3, 3
| 221184     |
| output_blocks.3.up.conv.bias                       | 64,
| 64         |
| output_blocks.4.res0.in_layers.norm.weight         | 128,
| 128        |
| output_blocks.4.res0.in_layers.norm.bias           | 128,
| 128        |
| output_blocks.4.res0.in_layers.conv.weight         | 64, 128, 3, 3, 3
| 221184     |
| output_blocks.4.res0.in_layers.conv.bias           | 64,
| 64         |
| output_blocks.4.res0.out_layers.norm.weight        | 64,
| 64         |
| output_blocks.4.res0.out_layers.norm.bias          | 64,
| 64         |
| output_blocks.4.res0.out_layers.conv.weight        | 64, 64, 3, 3, 3
| 110592     |
| output_blocks.4.res0.out_layers.conv.bias          | 64,
| 64         |
| output_blocks.4.res0.skip_connection.weight        | 64, 128, 1, 1, 1
| 8192       |
| output_blocks.4.res0.skip_connection.bias          | 64,
| 64         |
| output_blocks.5.res1.in_layers.norm.weight         | 128,
| 128        |
| output_blocks.5.res1.in_layers.norm.bias           | 128,
| 128        |
| output_blocks.5.res1.in_layers.conv.weight         | 64, 128, 3, 3, 3
| 221184     |
| output_blocks.5.res1.in_layers.conv.bias           | 64,
| 64         |
| output_blocks.5.res1.out_layers.norm.weight        | 64,
| 64         |
| output_blocks.5.res1.out_layers.norm.bias          | 64,
| 64         |
| output_blocks.5.res1.out_layers.conv.weight        | 64, 64, 3, 3, 3
| 110592     |
| output_blocks.5.res1.out_layers.conv.bias          | 64,
| 64         |
| output_blocks.5.res1.skip_connection.weight        | 64, 128, 1, 1, 1
| 8192       |
| output_blocks.5.res1.skip_connection.bias          | 64,
| 64         |
| output_blocks.5.up.conv.weight                     | 32, 64, 3, 3, 3
| 55296      |
| output_blocks.5.up.conv.bias                       | 32,
| 32         |
| output_blocks.6.res0.in_layers.norm.weight         | 64,
| 64         |
| output_blocks.6.res0.in_layers.norm.bias           | 64,
| 64         |
| output_blocks.6.res0.in_layers.conv.weight         | 32, 64, 3, 3, 3
| 55296      |
| output_blocks.6.res0.in_layers.conv.bias           | 32,
| 32         |
| output_blocks.6.res0.out_layers.norm.weight        | 32,
| 32         |
| output_blocks.6.res0.out_layers.norm.bias          | 32,
| 32         |
| output_blocks.6.res0.out_layers.conv.weight        | 32, 32, 3, 3, 3
| 27648      |
| output_blocks.6.res0.out_layers.conv.bias          | 32,
| 32         |
| output_blocks.6.res0.skip_connection.weight        | 32, 64, 1, 1, 1
| 2048       |
| output_blocks.6.res0.skip_connection.bias          | 32,
| 32         |
| output_blocks.7.res1.in_layers.norm.weight         | 64,
| 64         |
| output_blocks.7.res1.in_layers.norm.bias           | 64,
| 64         |
| output_blocks.7.res1.in_layers.conv.weight         | 32, 64, 3, 3, 3
| 55296      |
| output_blocks.7.res1.in_layers.conv.bias           | 32,
| 32         |
| output_blocks.7.res1.out_layers.norm.weight        | 32,
| 32         |
| output_blocks.7.res1.out_layers.norm.bias          | 32,
| 32         |
| output_blocks.7.res1.out_layers.conv.weight        | 32, 32, 3, 3, 3
| 27648      |
| output_blocks.7.res1.out_layers.conv.bias          | 32,
| 32         |
| output_blocks.7.res1.skip_connection.weight        | 32, 64, 1, 1, 1
| 2048       |
| output_blocks.7.res1.skip_connection.bias          | 32,
| 32         |
| out.norm.weight                                    | 32,
| 32         |
| out.norm.bias                                      | 32,
| 32         |
| out.conv.weight                                    | 32, 32, 3, 3, 3
| 27648      |
| out.conv.bias                                      | 32,
| 32         |
--------------------------------------------------------------------------------
--------------------
The total number of parameters: 26306176
The parameters of Model UNetModel: 26.306176M
The memory used now: 0.00MB
--------------------------------------------------------------------------------
--------------------
```