def forward(self, x: List[Tensor],  
 batch\_data\_samples: SampleList) -> Tuple[List[Tensor]]:  
 *"""Forward function.  
  
 Args:  
 x (list[Tensor]): Multi scale Features from the  
 upstream network, each is a 4D-tensor.  
 batch\_data\_samples (List[:obj:`DetDataSample`]): The Data  
 Samples. It usually includes information such as  
 `gt\_instance`, `gt\_panoptic\_seg` and `gt\_sem\_seg`.  
  
 Returns:  
 tuple[list[Tensor]]: A tuple contains two elements.  
  
 - cls\_pred\_list (list[Tensor)]: Classification logits \  
 for each decoder layer. Each is a 3D-tensor with shape \  
 (batch\_size, num\_queries, cls\_out\_channels). \  
 Note `cls\_out\_channels` should includes background.  
 - mask\_pred\_list (list[Tensor]): Mask logits for each \  
 decoder layer. Each with shape (batch\_size, num\_queries, \  
 h, w).  
 """* feat,adapter\_c=x[0],x[1]  
 device = x[0][0].device  
 adapter\_c0=adapter\_c[0]  
 batch\_img\_metas = [  
 data\_sample.metainfo for data\_sample in batch\_data\_samples  
 ]  
 batch\_size = len(batch\_img\_metas)  
 #可能的分解方向  
 mask\_features, multi\_scale\_memorys = self.pixel\_decoder(feat)  
 # adapter\_c = adapter\_c.flatten(2).permute(0, 2, 1)  
 adapter\_feat, adapter\_memo = self.pixel\_decoder(adapter\_c)  
  
 ######adapter初始化  
  
 # adapter\_feat=adapter\_feat.flatten(2).permute(0, 2, 1)  
 # conv\_layer = nn.Conv1d(in\_channels=1600, out\_channels=100, kernel\_size=1).to(adapter\_feat.device)  
 # # 对输入进行卷积操作，得到 (4, 100, 256)  
 # output = conv\_layer(adapter\_feat)  
 # # 归一化操作  
 # normalized\_output = nn.functional.normalize(output, p=2, dim=2)  
  
 src\_flatten = []  
 mask\_flatten = []  
 spatial\_shapes = []  
 masks = [torch.zeros((src.size(0), src.size(2), src.size(3)), device=src.device, dtype=torch.bool) for src in adapter\_c]  
 for i in range(self.num\_transformer\_feat\_level):  
 idx = self.num\_transformer\_feat\_level - 1 - i  
 spatial\_shapes.append(adapter\_c[idx].shape[-2:])  
 src\_flatten.append(self.decoder\_input\_projs[idx](adapter\_c[idx]).flatten(2).transpose(1, 2))  
 mask\_flatten.append(masks[i].flatten(1))  
 #附加  
 adapter\_memo[i] = adapter\_memo[i].flatten(2)  
 src\_flatten = torch.cat(src\_flatten, 1) # bs, \sum{hxw}, c  
 mask\_flatten = torch.cat(mask\_flatten, 1) # bs, \sum{hxw}  
 spatial\_shapes = torch.as\_tensor(spatial\_shapes, dtype=torch.long, device=src\_flatten.device)  
  
 #topk方法  
 adapter\_memo = torch.cat(adapter\_memo, 2)  
  
 output\_memory, output\_proposals = gen\_encoder\_output\_proposals(src\_flatten, mask\_flatten, spatial\_shapes)  
 output\_memory = self.enc\_output\_norm(self.enc\_output(output\_memory))  
  
 # output\_memory, output\_proposals = gen\_encoder\_output\_proposals(src\_flatten, mask\_flatten, spatial\_shapes)  
 enc\_outputs\_class\_unselected = self.cls\_embed(adapter\_memo.transpose(1, 2))  
 topk = self.num\_queries  
 topk\_proposals = torch.topk(enc\_outputs\_class\_unselected.max(-1)[0], topk, dim=1)[1]  
  
 tgt\_undetach = torch.gather(output\_memory, 1,  
 topk\_proposals.unsqueeze(-1).repeat(1, 1, self.feat\_channels)) # unsigmoid  
 tgt = tgt\_undetach.detach()  
  
  
 # multi\_scale\_memorys (from low resolution to high resolution)  
 decoder\_inputs = []  
 decoder\_positional\_encodings = []  
 for i in range(self.num\_transformer\_feat\_level):  
 decoder\_input = self.decoder\_input\_projs[i](multi\_scale\_memorys[i]) #(4,256,2,2)  
 # shape (batch\_size, c, h, w) -> (batch\_size, h\*w, c) (4,4,256)  
 decoder\_input = decoder\_input.flatten(2).permute(0, 2, 1)  
 level\_embed = self.level\_embed.weight[i].view(1, 1, -1)  
 decoder\_input = decoder\_input + level\_embed  
 # shape (batch\_size, c, h, w) -> (batch\_size, h\*w, c)  
 mask = decoder\_input.new\_zeros(  
 (batch\_size,) + multi\_scale\_memorys[i].shape[-2:],  
 dtype=torch.bool)  
 decoder\_positional\_encoding = self.decoder\_positional\_encoding(  
 mask)  
 decoder\_positional\_encoding = decoder\_positional\_encoding.flatten(  
 2).permute(0, 2, 1)  
 decoder\_inputs.append(decoder\_input)  
 decoder\_positional\_encodings.append(decoder\_positional\_encoding)  
 # shape (num\_queries, c) -> (batch\_size, num\_queries, c)  
 query\_feat = self.query\_feat.weight.unsqueeze(0).repeat(  
 (batch\_size, 1, 1))  
 query\_embed = self.query\_embed.weight.unsqueeze(0).repeat(  
 (batch\_size, 1, 1))  
  
 cls\_pred\_list = []  
 mask\_pred\_list = []  
 cls\_pred, mask\_pred, attn\_mask = self.\_forward\_head(  
 query\_feat, mask\_features, multi\_scale\_memorys[0].shape[-2:])  
 cls\_pred\_list.append(cls\_pred)  
 mask\_pred\_list.append(mask\_pred)  
  
 for i in range(self.num\_transformer\_decoder\_layers):  
 level\_idx = i % self.num\_transformer\_feat\_level  
 # if a mask is all True(all background), then set it all False.  
 attn\_mask[torch.where(  
 attn\_mask.sum(-1) == attn\_mask.shape[-1])] = False  
  
 # cross\_attn + self\_attn  
 layer = self.transformer\_decoder.layers[i]  
 query\_feat = layer(  
 query=query\_feat,  
 key=decoder\_inputs[level\_idx],  
 value=decoder\_inputs[level\_idx],  
 query\_pos=query\_embed,  
 key\_pos=decoder\_positional\_encodings[level\_idx],  
 cross\_attn\_mask=attn\_mask,  
 query\_key\_padding\_mask=None,  
 # here we do not apply masking on padded region  
 key\_padding\_mask=None)  
  
 cls\_pred, mask\_pred, attn\_mask = self.\_forward\_head(  
 query\_feat, mask\_features, multi\_scale\_memorys[  
 (i + 1) % self.num\_transformer\_feat\_level].shape[-2:])  
  
 cls\_pred\_list.append(cls\_pred)  
 mask\_pred\_list.append(mask\_pred)  
  
 return cls\_pred\_list, mask\_pred\_list