HaLViT: Half of the Weights are Enough

Supplementary Material

1. Pytorch Implementation

We've developed a PyTorch-based implementation of HaLViT, structuring the model by separately defining the Attention and Feed-Forward layers as distinct classes. These components are then integrated within the overarching HaLViT class. This modular approach enhances the clarity and maintainability of the code, allowing for more straightforward modifications and extensions to individual model components. By encapsulating the attention mechanism and feed-forward network in separate classes, we facilitate easier experimentation with different architectural variations and optimizations. Ultimately, this organization paves the way for more efficient development cycles and a clearer understanding of the model's inner workings, laying a solid foundation for further research and innovation in transformer-based architectures.

```
import torch
from torch import nn
import math
from einops import rearrange, repeat
from torchvision.ops import StochasticDepth
from einops.layers.torch import Rearrange
from mmpretrain.registry import MODELS
import numpy as np

def pair(t):
    return t if isinstance(t, tuple) else (t, t)

#Feed Forward Layer
class FF(nn.Module):
    def __init__(self, dim, hidden_dim, dropout=0.,sdepth=0.):
        super().__init__()
        self.w = nn.Parameter(torch.empty(dim,hidden_dim))
        self.b1 = nn.Parameter(torch.empty(hidden_dim))
        self.b2 = nn.Parameter(torch.empty(dim))
        self.act = nn.GELU()
        self.dropout = nn.Dropout(dropout)
        self.ln1 = nn.LayerNorm(dim)
        self.sd = StochasticDepth(sdepth, "row")

        nn.init.kaiming_uniform_(self.w)
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.w)
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(self.b1, -bound, bound)

        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.w.t())
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(self.b2, -bound, bound)

    def forward(self, x):
        x = self.ln1(x)
        x = self.dropout(self.act(torch.matmul(x,self.w)+self.b1))
        x = self.dropout(torch.matmul(x, self.w.t())+self.b2)
        return self.sd(x)
```

```
#Attention Layer
class Attention(nn.Module):
    def __init__(self, dim=768, heads=12, dim_head=64, dropout = 0.,sdepth=0.):
        super().__init__()
        inner_dim = dim_head *  heads
        self.inner_dim = inner_dim

        self.heads = heads
        self.q = nn.Parameter(torch.empty(dim,inner_dim))

        self.inpnorm = nn.LayerNorm(dim)
        self.qnorm = nn.LayerNorm(inner_dim)
        self.knorm = nn.LayerNorm(inner_dim)
        self.bias = nn.Parameter(torch.empty(dim))


        self.kv = nn.Parameter(torch.empty(dim,inner_dim))

        self.act = nn.GELU()


        self.attend = nn.Softmax(dim = -1)
        self.dropout = nn.Dropout(dropout)

        self.sd = StochasticDepth(sdepth, "row")


        self.scale = dim_head ** -0.5

        for param in self.parameters():
            if not isinstance(param, nn.LayerNorm) and len(param.shape)>1:
                nn.init.kaiming_uniform_(param)
        fan_in, _ = torch.nn.init._calculate_fan_in_and_fan_out(self.q.t())
        bound = 1 / math.sqrt(fan_in) if fan_in > 0 else 0
        torch.nn.init.uniform_(self.bias, -bound, bound)
    def forward(self, x):
        x = self.inpnorm(x)
        q,k,v= torch.tensor_split(torch.matmul(x,
                          torch.cat([self.q,self.kv,self.kv.t()],1)),
                              (self.inner_dim,2*self.inner_dim,),dim=-1)
        q = self.qnorm(q)
        k = self.knorm(k)
        q,k,v = map(lambda t: rearrange(t, 'b n (h d) -> b h n d',
                          h = self.heads), (q,k,v))

        dots = torch.matmul(q, k.transpose(-1, -2)) * self.scale

        attn = self.attend(dots)
        attn = self.dropout(attn)

        # m = torch.matmul(torch.cat(self.act(self.dropout(m)),self.mlp.t())


        out = torch.matmul(attn, v)
        out = rearrange(out, 'b h n d -> b n (h d)')
        out = torch.matmul(out,self.q.t())+self.bias

        return self.sd(out)
```

```
class Transformer(nn.Module):
    def __init__(self, dim=768, depth=12, heads=12,
                        dim_head=64, mlp_dim=768, dropout = 0.,sdepth=0.0):
        super().__init__()
        self.norm = nn.LayerNorm(dim)
        self.layers = nn.ModuleList([])
        dpr = np.linspace(0, sdepth, depth)

        for i in range(depth):
            self.layers.append(Attention(dim, heads, dim_head, dropout,sdepth=dpr[i]))
            self.layers.append(FF(dim, mlp_dim,dropout,sdepth=dpr[i]))


    def forward(self, x):
        for layer in self.layers:
            x = layer(x)+x

        return self.norm(x)
```

```
@MODELS.register_module()
class HaLViT(nn.Module):
    def __init__(self, *, image_size=224, patch_size=16, num_classes=1000, dim=768,
    depth=12, heads=12,dim_head = 64, mlp_dim=768*4,
    pool = 'cls', deit=False ,channels = 3,  dropout = 0., emb_dropout = 0.):
        super().__init__()
        image_height, image_width = pair(image_size)
        patch_height, patch_width = pair(patch_size)
        self.deit = deit
        assert image_height % patch_height == 0 and image_width % patch_width == 0,
        'Image dimensions must be divisible by the patch size.'

        num_patches = (image_height // patch_height) * (image_width // patch_width)
        patch_dim = channels * patch_height * patch_width
        assert pool in {'cls', 'mean'},
        'pool type must be either cls (cls token) or mean (mean pooling)'

        self.to_patch_embedding = nn.Sequential(
            Rearrange('b c (h p1) (w p2) -> b (h w) (p1 p2 c)',
                        p1 = patch_height, p2 = patch_width),
            nn.LayerNorm(patch_dim),
            nn.Linear(patch_dim, dim),
            nn.LayerNorm(dim),
        )


        self.cls_token = nn.Parameter(torch.randn(1, 1, dim))
        if deit:
            self.pos_embedding = nn.Parameter(torch.randn(1, num_patches +2, dim))
            self.dist_token = nn.Parameter(torch.randn(1, 1, dim))
        else:
            self.pos_embedding = nn.Parameter(torch.randn(1, num_patches +1, dim))
        self.dropout = nn.Dropout(emb_dropout)

        self.transformer = Transformer(dim, depth, heads, dim_head, mlp_dim, dropout)

        self.pool = pool


    def forward(self, img):
        x = self.to_patch_embedding(img)
        b, n, _ = x.shape

        cls_tokens = repeat(self.cls_token, '1 n d -> b n d', b = b)

        x = torch.cat((cls_tokens, x), dim=1)
        x += self.pos_embedding[:, :(n + 1)]

        x = self.dropout(x)

        x = self.transformer(x)

        return tuple([None,x[:,0]])
```