Supplementary Material for Binarized Mamba-Transformer for Lightweight Quad Bayer HybridEVS Demosaicing

In the appendix, we first present extensive visualizations of both real and simulated datasets (See Sec. A). Next, we provide a detailed analysis of Bi-Mamba, highlighting its reduction in model complexity (See Sec. B). Finally, we include PyTorch-style pseudo-code for the Mamba-Transformer Block, Binarized Mamba, and Binarized Cross Swin Transformer (See Sec. C).

A. Additoinal Visualization Results

We illustrate additional visualization comparison results of our proposed BMTNet with other State-of-the-Art BNN methods on simulated and real data, as shown in Figure 2 and Figure 1. Our BMTNet demonstrates strong robustness on various scenes and effectively reduces moiré and zip artifacts compared with other BNN methods. The results on real data appear to have a green tint due to the absence of post-processing like color correction.

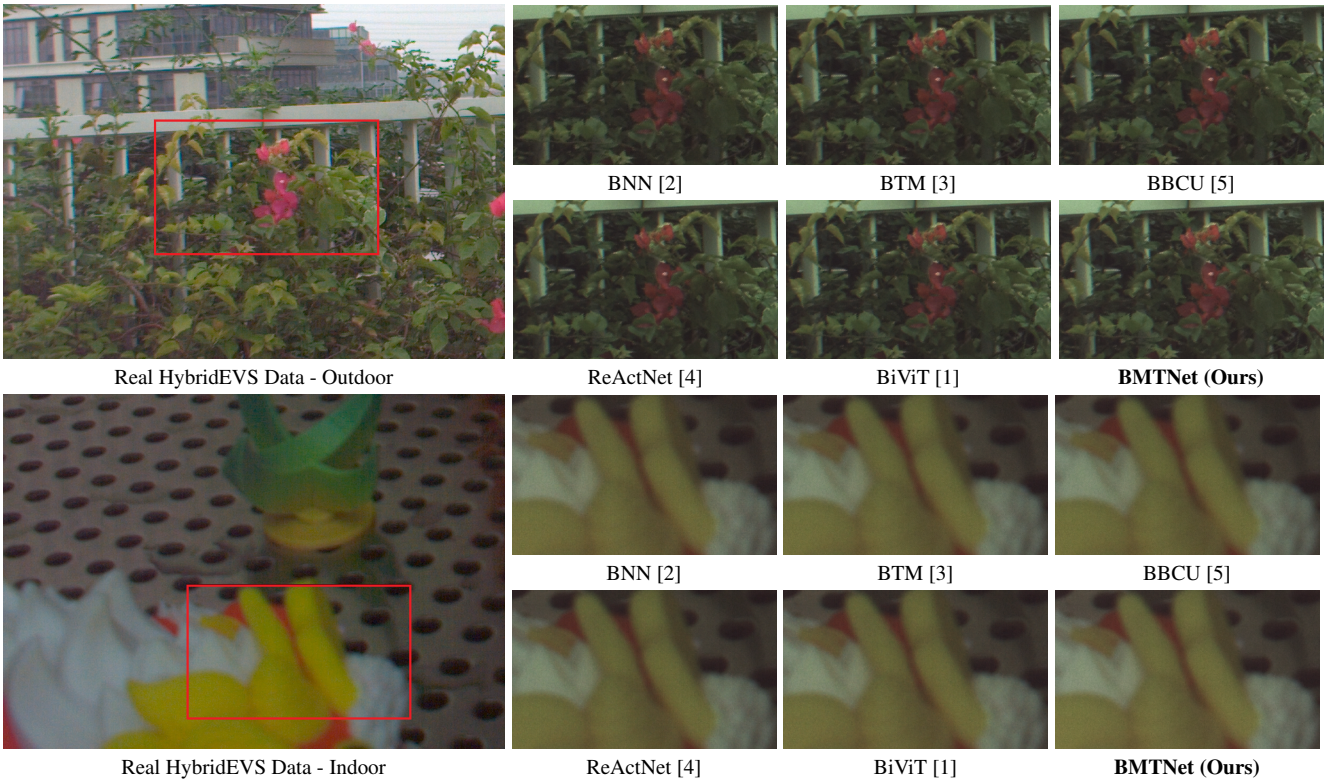

Figure 1. Visualization of the real HybridEVS data across outdoor and indoor scenes. The reference is acquired from a simple ISP, which has an additional color adjustment process.

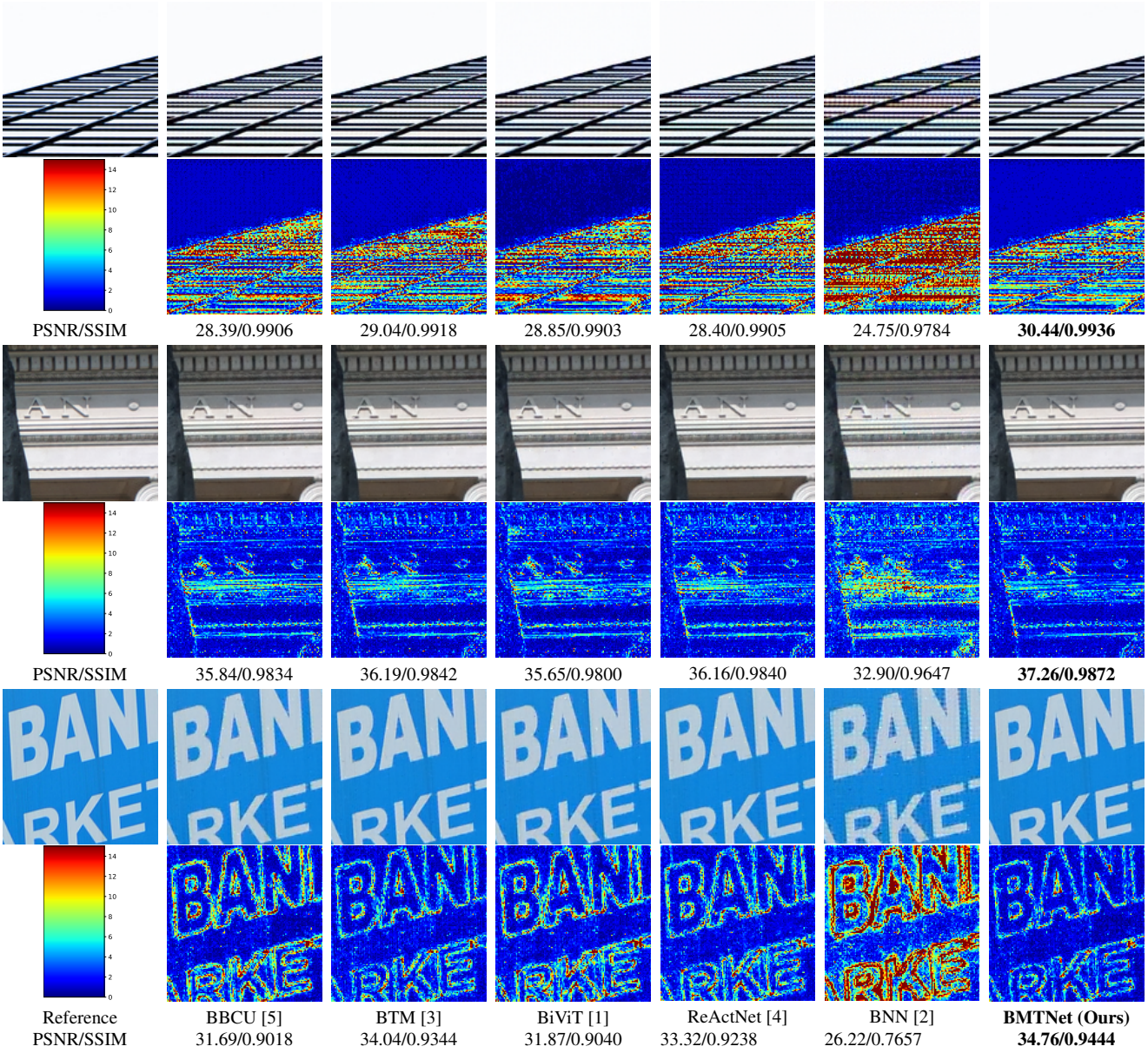

Figure 2. Visual comparison results with difference maps. Our BMTNet surpasses all other BNN methods with less pixel difference.

B. Details of Bi-Mamba

We analyze the model complexity of Mamba before and after binarization, as detailed in Table 1. Specific to the Mamba blocks, as shown in Table 1, the core Selective Scan function accounts for only 2.2% of the total computational load in full-precision Mamba, indicating that most non-critical computations can be binarized. Bi-Mamba significantly compresses the projections, including linear and convolution layers, reducing parameters and computation costs by 97% and 96%, respectively.

Mamba Components	Projections	Selective Scan	Total
	Params (M) / OPs (G)		
Before binarization	5.02/48.27	0/1.09	5.02/49.36
After binarization	**0.16/0.75**	0/1.09	**0.16/1.84**

Table 1. Analysis of Mamba binarization. The proposed binary format significantly reduces parameters and operations.

C. Model details in PyTorch style pseudo-code

In this section, we provide a detailed explanation of the modules we designed, including Cross-Swin State Block, Conv State Block, Quad Bayer Cross Swin Attention and Spatial Position Attention using PyTorch style pseudo-code. **The specific runnable model code is provided in the supplementary material.**

Algorithm 1 Pseudo-code of Binarized Semantic Mamba

```
## Binarized Semantic Mamba (BiS-Mamba)

class biMambaBlock(nn.Module):
    def __init__(
            self,
            hidden_dim: int = 0,
            drop_path: float = 0,
            norm_layer: Callable[..., torch.nn.Module] = partial(nn.LayerNorm, eps=1e-6),
            attn_drop_rate: float = 0,
            d_state: int = 16,
            expand: float = 2.,
            # is_light_sr: bool = False,
            **kwargs,
    ):
        super().__init__()
        self.ln_1 = norm_layer(hidden_dim)
        self.ln_2 = norm_layer(1)
        self.self_attention = biSS2D(d_model=hidden_dim, d_state=d_state,expand=expand,dropout=attn_drop_rate, **kwa
        self.drop_path = DropPath(drop_path)
        self.skip_scale= nn.Parameter(torch.ones(hidden_dim))


    def forward(self, input, embed):
        input = rearrange(input, 'b c h w -> b h w c')
        embed = rearrange(embed, 'b c h w -> b h w c')
        x = self.ln_1(input)
        y = self.ln_2(embed)
        x = input*self.skip_scale + self.drop_path(self.self_attention(x, y))
        x = rearrange(x, 'b h w c -> b c h w')
        return x
```

Algorithm 2 Pseudo-code of Selective Scan in BiS-Mamba

```
## Selective Scan in BiS-Mamba

class biSS2D(nn.Module):
    def __init__(
            self,
            d_model,
            d_state=16,
            d_conv=3,
            expand=2.,
            dt_rank="auto",
            dt_min=0.001,
            dt_max=0.1,
            dt_init="random",
            dt_scale=1.0,
            dt_init_floor=1e-4,
            dropout=0.,
            conv_bias=True,
            bias=False,
            device=None,
            dtype=None,
            **kwargs,
    ):
        factory_kwargs = {"device": device, "dtype": dtype}
```

```
        super().__init__()
        self.d_model = d_model
        self.d_state = d_state
        self.d_conv = d_conv
        self.expand = expand
        self.d_inner = int(self.expand * self.d_model)
        self.dt_rank = math.ceil(self.d_model / 16) if dt_rank == "auto" else dt_rank

        self.in_proj = BinaryLinear_adapscaling(self.d_model, self.d_inner * 2, bias=bias)
        self.conv2d = DABCConv2d(
            in_channels=self.d_inner,
            out_channels=self.d_inner,
            kernel_size=d_conv,
            bias=conv_bias
        )
        self.act = nn.SiLU()


        self.l1 = BinaryLinear_adapscaling(self.d_inner, (self.dt_rank + self.d_state ), bias=False)
        self.l2 = BinaryLinear_adapscaling(self.d_inner, (self.dt_rank + self.d_state ), bias=False)
        self.l3 = BinaryLinear_adapscaling(self.d_inner, (self.dt_rank + self.d_state ), bias=False)
        self.l4 = BinaryLinear_adapscaling(self.d_inner, (self.dt_rank + self.d_state ), bias=False)

        self.l1_b = BinaryLinear_adapscaling(self.d_inner + 1, (self.d_state), bias=False)
        self.l2_b = BinaryLinear_adapscaling(self.d_inner + 1, (self.d_state), bias=False)
        self.l3_b = BinaryLinear_adapscaling(self.d_inner + 1, (self.d_state), bias=False)
        self.l4_b = BinaryLinear_adapscaling(self.d_inner + 1, (self.d_state), bias=False)




        self.ld1, bias1 = self.dt_init(self.dt_rank, self.d_inner, dt_scale, dt_init, dt_min, dt_max, dt_init_floor,
                          **factory_kwargs)
        self.ld2, bias2 = self.dt_init(self.dt_rank, self.d_inner, dt_scale, dt_init, dt_min, dt_max, dt_init_floor,
                          **factory_kwargs)
        self.ld3, bias3 = self.dt_init(self.dt_rank, self.d_inner, dt_scale, dt_init, dt_min, dt_max, dt_init_floor,
                          **factory_kwargs)
        self.ld4, bias4 = self.dt_init(self.dt_rank, self.d_inner, dt_scale, dt_init, dt_min, dt_max, dt_init_floor,
                          **factory_kwargs)
        # self.dt_projs = (self.ld1,self.ld2,self.ld3,self.ld4)
        # self.dt_projs_weight = nn.Parameter(torch.stack([t.weight for t in self.dt_projs], dim=0))  # (K=4, inner,
        self.dt_projs_bias = nn.Parameter(torch.stack([bias1, bias2, bias3, bias4], dim=0))  # (K=4, inner)
        # print(self.ld4.bias.shape)
        # del self.dt_projs

        self.A_logs = self.A_log_init(self.d_state, self.d_inner, copies=4, merge=True)  # (K=4, D, N)
        self.Ds = self.D_init(self.d_inner, copies=4, merge=True)  # (K=4, D, N)

        self.selective_scan = selective_scan_fn

        self.out_norm = nn.LayerNorm(self.d_inner)
        self.out_proj = BinaryLinear_adapscaling(self.d_inner, self.d_model, bias=bias)
        self.dropout = nn.Dropout(dropout) if dropout > 0. else None

    @staticmethod
    def dt_init(dt_rank, d_inner, dt_scale=1.0, dt_init="random", dt_min=0.001, dt_max=0.1, dt_init_floor=1e-4,
                **factory_kwargs):
        dt_proj = BinaryLinear_adapscaling(dt_rank, d_inner, bias=False)

        # Initialize special dt projection to preserve variance at initialization
        dt_init_std = dt_rank ** -0.5 * dt_scale
        if dt_init == "constant":
            nn.init.constant_(dt_proj.weight, dt_init_std)
        elif dt_init == "random":
            nn.init.uniform_(dt_proj.weight, -dt_init_std, dt_init_std)
        else:
            raise NotImplementedError
```

```
        # Initialize dt bias so that F.softplus(dt_bias) is between dt_min and dt_max
        dt = torch.exp(
            torch.rand(d_inner, **factory_kwargs) * (math.log(dt_max) - math.log(dt_min))
            + math.log(dt_min)
        ).clamp(min=dt_init_floor)
        # Inverse of softplus: https://github.com/pytorch/pytorch/issues/72759
        inv_dt = dt + torch.log(-torch.expm1(-dt))
        # with torch.no_grad():
        #     dt_proj.bias.copy_(inv_dt)
        # # Our initialization would set all Linear.bias to zero, need to mark this one as _no_reinit
        # dt_proj.bias._no_reinit = True

        return dt_proj, inv_dt

    @staticmethod
    def A_log_init(d_state, d_inner, copies=1, device=None, merge=True):
        # S4D real initialization
        A = repeat(
            torch.arange(1, d_state + 1, dtype=torch.float32, device=device),
            "n -> d n",
            d=d_inner,
        ).contiguous()
        A_log = torch.log(A)  # Keep A_log in fp32
        if copies > 1:
            A_log = repeat(A_log, "d n -> r d n", r=copies)
            if merge:
                A_log = A_log.flatten(0, 1)
        A_log = nn.Parameter(A_log)
        A_log._no_weight_decay = True
        return A_log

    @staticmethod
    def D_init(d_inner, copies=1, device=None, merge=True):
        # D "skip" parameter
        D = torch.ones(d_inner, device=device)
        if copies > 1:
            D = repeat(D, "n1 -> r n1", r=copies)
            if merge:
                D = D.flatten(0, 1)
        D = nn.Parameter(D)  # Keep in fp32
        D._no_weight_decay = True
        return D

    def forward_core(self, x: torch.Tensor, y: torch.Tensor):
        B, C, H, W = x.shape
        L = H * W
        K = 4
        y = y.view(B,L,-1)
        x_hwwh = torch.stack([x.view(B, -1, L), torch.transpose(x, dim0=2, dim1=3).contiguous().view(B, -1, L)], dim
        xs = torch.cat([x_hwwh, torch.flip(x_hwwh, dims=[-1])], dim=1) # (1, 4, 192, 3136)
        xs2 = xs.view(B, K, -1, L)
        x0 = xs2[:, 0, :, :].view(B, L, -1)
        x1 = xs2[:, 1, :, :].view(B, L, -1)
        x2 = xs2[:, 2, :, :].view(B, L, -1)
        x3 = xs2[:, 3, :, :].view(B, L, -1)
        # print(x0.shape)
        # print(y.shape)
        b0 = self.l1_b(torch.cat((x0,y), dim=-1))
        b1 = self.l2_b(torch.cat((x1,y), dim=-1))
        b2 = self.l3_b(torch.cat((x2,y), dim=-1))
        b3 = self.l4_b(torch.cat((x3,y), dim=-1))
        x0 = self.l1(x0)
        x1 = self.l2(x1)
        x2 = self.l3(x2)
        x3 = self.l4(x3)
```

```
        Bs = rearrange(torch.stack([b0,b1,b2,b3], dim=1),'b k l c -> b k c l')

        x_dbl = rearrange(torch.stack([x0,x1,x2,x3], dim=1),'b k l c -> b k c l')

        dts, Cs = torch.split(x_dbl, [self.dt_rank, self.d_state], dim=2)

        dts2 = dts.view(B, K, -1, L)
        dts2 = [dts[:, i, :, :].contiguous().view(B, L, -1) for i in range(4)]
        dt0 = self.ld1(dts2[0])
        dt1 = self.ld2(dts2[1])
        dt2 = self.ld3(dts2[2])
        dt3 = self.ld4(dts2[3])

        dts = rearrange(torch.stack([dt0,dt1,dt2,dt3], dim=1),'b k l d -> b k d l')
        xs = xs.float().view(B, -1, L)
        dts = dts.contiguous().float().view(B, -1, L) # (b, k * d, l)
        Bs = Bs.float().view(B, K, -1, L)
        Cs = Cs.float().view(B, K, -1, L) # (b, k, d_state, l)
        Ds = self.Ds.float().view(-1)
        As = -torch.exp(self.A_logs.float()).view(-1, self.d_state)
        dt_projs_bias = self.dt_projs_bias.float().view(-1) # (k * d)
        out_y = self.selective_scan(
            xs, dts,
            As, Bs, Cs, Ds, z=None,
            delta_bias=dt_projs_bias,
            delta_softplus=True,
            return_last_state=False,
        ).view(B, K, -1, L)
        assert out_y.dtype == torch.float
        # print(out_y.shape)
        inv_y = torch.flip(out_y[:, 2:4], dims=[-1]).view(B, 2, -1, L)
        wh_y = torch.transpose(out_y[:, 1].view(B, -1, W, H), dim0=2, dim1=3).contiguous().view(B, -1, L)
        invwh_y = torch.transpose(inv_y[:, 1].view(B, -1, W, H), dim0=2, dim1=3).contiguous().view(B, -1, L)

        return out_y[:, 0], inv_y[:, 0], wh_y, invwh_y

    def forward(self, x: torch.Tensor, y: torch.Tensor, **kwargs):
        B, H, W, C = x.shape

        xz = self.in_proj(x)
        x, z = xz.chunk(2, dim=-1)

        x = x.permute(0, 3, 1, 2).contiguous()
        x = self.act(self.conv2d(x))
        y1, y2, y3, y4 = self.forward_core(x, y)
        assert y1.dtype == torch.float32
        y = y1 + y2 + y3 + y4
        y = torch.transpose(y, dim0=1, dim1=2).contiguous().view(B, H, W, -1)
        y = self.out_norm(y)
        y = y * F.silu(z)
        out = self.out_proj(y)
        if self.dropout is not None:
            out = self.dropout(out)
        return out
```

Algorithm 3 Pseudo-code of Binarized Swin Transformer

```
## Binarized Swin Transformer (Bi-SwinT)

class biBlock(nn.Module):
    def __init__(self, input_dim, output_dim, head_dim, window_size, drop_path, type='W', input_resolution=None):
        """ SwinTransformer Block
        """
        super(biBlock, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        assert type in ['W', 'SW']
        self.type = type
        if input_resolution <= window_size:
            self.type = 'W'

        print("Block Initial Type: {}, drop_path_rate:{:.6f}".format(self.type, drop_path))
        self.ln1 = nn.LayerNorm(input_dim)
        self.msa = WMSA(input_dim, input_dim, head_dim, window_size, self.type)
        self.drop_path = DropPath(drop_path) if drop_path > 0. else nn.Identity()
        self.ln2 = nn.LayerNorm(input_dim)
        self.mlp = nn.Sequential(
            BinaryLinear_adapscaling(input_dim, 4 * input_dim),
            nn.GELU(),
            BinaryLinear_adapscaling(4 * input_dim, output_dim),
        )


    def forward(self, x):
        x = x + self.drop_path(self.msa(self.ln1(x)))
        x = x + self.drop_path(self.mlp(self.ln2(x)))
        return x
```

Algorithm 4 Pseudo-code of Self-attention module in Binarized Swin Transformer

```
class WMSA(nn.Module):
    """ Self-attention module in Bi-SwinT
    """

    def __init__(self, input_dim, output_dim, head_dim, window_size, type):
        super(WMSA, self).__init__()
        self.input_dim = input_dim
        self.output_dim = output_dim
        self.head_dim = head_dim
        self.scale = self.head_dim ** -0.5
        self.n_heads = input_dim//head_dim
        self.window_size = window_size
        self.type=type
        self.embedding_layer = BinaryLinear_adapscaling(self.input_dim, 3*self.input_dim, bias=True)
        # self.embedding_layer_qb = BinaryLinear_adapscaling(self.input_dim, self.input_dim, bias=True)
        # TODO recover
        # self.relative_position_params = nn.Parameter(torch.zeros(self.n_heads, 2 * window_size - 1, 2 * window_siz
        self.relative_position_params = nn.Parameter(torch.zeros((2 * window_size - 1)*(2 * window_size -1), self.n_

        self.linear = BinaryLinear_adapscaling(self.input_dim, self.output_dim)

        trunc_normal_(self.relative_position_params, std=.02)
        self.relative_position_params = torch.nn.Parameter(self.relative_position_params.view(2*window_size-1, 2*win

        self.quant_layer = BinaryQuantizer().apply
        self.attn_quant_layer = SoftmaxBinaryQuantizer().apply


    def generate_mask(self, h, w, p, shift):
        """ generating the mask of SW-MSA
```

```
        Args:
            shift: shift parameters in CyclicShift.
        Returns:
            attn_mask: should be (1 1 w p p),
        """
        # supporting sqaure.
        attn_mask = torch.zeros(h, w, p, p, p, p, dtype=torch.bool, device=self.relative_position_params.device)
        if self.type == 'W':
            return attn_mask

        s = p - shift
        attn_mask[-1, :, :s, :, s:, :] = True
        attn_mask[-1, :, s:, :, :s, :] = True
        attn_mask[:, -1, :, :s, :, s:] = True
        attn_mask[:, -1, :, s:, :, :s] = True
        attn_mask = rearrange(attn_mask, 'w1 w2 p1 p2 p3 p4 -> 1 1 (w1 w2) (p1 p2) (p3 p4)')
        return attn_mask

    def forward(self, x):
        """ Forward pass of Window Multi-head Self-attention module.
        Args:
            x: input tensor with shape of [b h w c];
            attn_mask: attention mask, fill -inf where the value is True;
        Returns:
            output: tensor shape [b h w c]
        """
        if self.type!='W':
            x = torch.roll(x, shifts=(-(self.window_size//2), -(self.window_size//2)), dims=(1,2))
            # y = torch.roll(y, shifts=(-(self.window_size//2), -(self.window_size//2)), dims=(1,2))
        x = rearrange(x, 'b (w1 p1) (w2 p2) c -> b w1 w2 p1 p2 c', p1=self.window_size, p2=self.window_size)
        h_windows = x.size(1)
        w_windows = x.size(2)
        # sqaure validation
        # assert h_windows == w_windows

        x = rearrange(x, 'b w1 w2 p1 p2 c -> b (w1 w2) (p1 p2) c', p1=self.window_size, p2=self.window_size)
        qkv = self.embedding_layer(x)
        q, k, v = rearrange(qkv, 'b nw np (threeh c) -> threeh b nw np c', c=self.head_dim).chunk(3, dim=0)
        # q = rearrange(q, 'b nw np (h c) -> h b nw np c', c=self.head_dim)
        # y = rearrange(y, 'b (w1 p1) (w2 p2) c -> b (w1 w2) (p1 p2) c', p1=self.window_size, p2=self.window_size)
        # q = self.embedding_layer_qb(y)
        # q = rearrange(q, 'b nw np (h c) -> h b nw np c', c=self.head_dim)

        binary_q = self.quant_layer(q)
        binary_k = self.quant_layer(k)
        binary_v = self.quant_layer(v)


        sim = torch.einsum('hbwpc,hbwqc->hbwpq', binary_q, binary_k) * self.scale
        # Adding learnable relative embedding
        sim = sim + rearrange(self.relative_embedding(), 'h p q -> h 1 1 p q')


        # print(f"sim: {sim.shape}")
        # print(f"position embed: {rearrange(self.relative_embedding(), 'h p q -> h 1 1 p q').shape}")
        # print(f"positoin params: {self.relative_embedding().shape}")
        # Using Attn Mask to distinguish different subwindows.
        if self.type != 'W':
            attn_mask = self.generate_mask(h_windows, w_windows, self.window_size, shift=self.window_size//2)
            sim = sim.masked_fill_(attn_mask, float("-inf"))

        # probs = nn.functional.softmax(sim, dim=-1)
        probs = self.attn_quant_layer(sim).float().detach() - sim.softmax(-1).detach() + sim.softmax(-1)
        output = torch.einsum('hbwij,hbwjc->hbwic', probs, binary_v)
        output = rearrange(output, 'h b w p c -> b w p (h c)')
        output = self.linear(output)
        output = rearrange(output, 'b (w1 w2) (p1 p2) c -> b (w1 p1) (w2 p2) c', w1=h_windows, p1=self.window_size)
```

```
        if self.type!='W': output = torch.roll(output, shifts=(self.window_size//2, self.window_size//2), dims=(1,2))
        return output

    def relative_embedding(self):
        cord = torch.tensor(np.array([[i, j] for i in range(self.window_size) for j in range(self.window_size)]))
        relation = cord[:, None, :] - cord[None, :, :] + self.window_size -1
        # negative is allowed
        return self.relative_position_params[:, relation[:,:,0].long(), relation[:,:,1].long()]
```

References

[1] Yefei He, Zhenyu Lou, Luoming Zhang, Jing Liu, Weijia Wu, Hong Zhou, and Bohan Zhuang. Bivit: Extremely compressed binary vision transformers. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 5651–5663, 2023.

[2] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks. In *Proceedings of the Advances in Neural Information Processing Systems*, 2016.

[3] Xinrui Jiang, Nannan Wang, Jingwei Xin, Keyu Li, Xi Yang, and Xinbo Gao. Training binary neural network without batch normalization for image super-resolution. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1700–1707, 2021.

[4] Zechun Liu, Zhiqiang Shen, Marios Savvides, and Kwang-Ting Cheng. Reactnet: Towards precise binary neural network with generalized activation functions. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIV 16*, pages 143–159. Springer, 2020.

[5] Bin Xia, Yulun Zhang, Yitong Wang, Yapeng Tian, Wenming Yang, Radu Timofte, and Luc Van Gool. Basic binary convolution unit for binarized image restoration network. *arXiv preprint arXiv:2210.00405*, 2022.