MetaFormer is Actually What You Need for Vision
– *Supplementary Material* –

Weihao Yu[1,2*] Mi Luo[1] Pan Zhou[1] Chenyang Si[1] Yichen Zhou[1,2]
Xinchao Wang[2] Jiashi Feng[1] Shuicheng Yan[1]
[1]Sea AI Lab [2]National University of Singapore
weihaoyu6@gmail.com {luomi,zhoupan,sicy,zhouyc,fengjs,yansc}@sea.com xinchao@nus.edu.sg
Code: https://github.com/sail-sg/poolformer

A. Detailed hyper-parameters on ImageNet-1K

PoolFormer. On ImageNet-1K classification benchmark, we utilize the hyper-parameters shown in Table 1 to train models in our paper. Based on the relation between batch size and learning rate in Table 1, we set the batch size as 4096 and learning rate as 4×10^{-3}. For stochastic depth, following the original paper [3], we linearly increase the probability of dropping a layer from 0.0 for the bottom block to d_r for the top block.

Hybrid Models. We use the hyper-parameters for all models except for the hybrid models with token mixers of pooling and attention. For these hybrid models, we find that they achieve much better performances by setting batch size as 1024, learning rate as 10^{-3}, and normalization as Layer Normalization [1].

B. Training for longer epochs

In our paper, PoolFormer models are trained for the default 300 epochs on ImageNet-1K. For DeiT [6]/ResMLP [5], it is observed that the performance saturates after 400/800 epochs. Thus, we also conduct the experiments of training longer for PoolFormer-S12 and the results are shown in Table 2. We observe that PoolFormer-S12 obtains saturated performance after around 2000 epochs with a top-1 accuracy improvement of 1.8%. However, for fair comparison with other ViT/MLP-like models, we still train PoolFormers for 300 epochs by default.

C. Qualitative results

We use Grad-CAM [4] to visualize the results of different models trained on ImageNet-1K. We find that although ResMLP [5] also activates some irrelevant parts, all models can locate the semantic objects. The activation parts of DeiT [6] and ResMLP [5] in the maps are more scattered, while those of RSB-ResNet [2, 7] and PoolFormer are more gathered.

D. Comparison between Layer Normalization and Modified Layer Normalization

We modify Layer Normalization [1] into Modified Layer Normalization (MNN). It computes the mean and variance along spatial and channel dimensions, compared with only channel dimension in vanilla Layer Normalization. The shape of learnable affine parameters of MLN keeps the same as that of Layer Normalization, *i.e.*, $\mathbb{R}^{\mathbb{C}}$. MLN can be implemented with GroupNorm API in PyTorch by setting the group number as 1. The comparison details are shown in Algorithm 1.

E. Code in PyTorch

We provide the PyTorch-like code in Algorithm 2 associated with the modules used in the PoolFormer block. Algorithm 3 further shows the PoolFormer block built with these modules.

[*]Work done during an internship at Sea AI Lab.

	PoolFormer				
	S12	S24	S36	M36	M48
Peak drop rate of stoch. depth d_r	0.1	0.1	0.2	0.3	0.4
LayerScale initialization ϵ	10^{-5}	10^{-5}	10^{-6}	10^{-6}	10^{-6}
Data augmentation	AutoAugment				
Repeated Augmentation	off				
Input resolution	224				
Epochs	300				
Warmup epochs	5				
Hidden dropout	0				
GeLU dropout	0				
Classification dropout	0				
Random erasing prob	0.25				
EMA decay	0				
Cutmix α	1.0				
Mixup α	0.8				
Cutmix-Mixup switch prob	0.5				
Label smoothing	0.1				
Relation between peak learning rate and batch size	$\text{lr} = \frac{\text{batch_size}}{1024} \times 10^{-3}$				
Batch size used in the paper	4096				
Peak learning rate used in the paper	4×10^{-4}				
Learning rate decay	cosine				
Optimizer	AdamW				
Adam ϵ	1e-8				
Adam (β_1, β_2)	(0.9, 0.999)				
Weight decay	0.05				
Gradient clipping	None				

Table 1. **Hyper-parameters for image classification on ImageNet-1K**

# Epochs	300 (default)	400	500	1000	1500	2000	2500	3000
PoolFormer-S12	77.2	77.5	77.9	78.4	78.6	78.8	78.8	78.8

Table 2. **Performance of PoolFormer trained for different numbers of epochs**.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 1

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 1

[3] Gao Huang, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Q Weinberger. Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer, 2016. 1

[4] Ramprasaath R Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision*, pages 618–626, 2017. 1, 3

[5] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Gautier Izacard, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, et al. Resmlp: Feedforward networks for image classification with data-efficient training. *arXiv preprint arXiv:2105.03404*, 2021. 1, 3

[6] Hugo Touvron, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. Training data-efficient image transformers & distillation through attention. In *International Conference on Machine Learning*, pages 10347–10357. PMLR, 2021. 1, 3

[7] Ross Wightman, Hugo Touvron, and Hervé Jégou. Resnet strikes back: An improved training procedure in timm. *arXiv preprint arXiv:2110.00476*, 2021. 1, 3

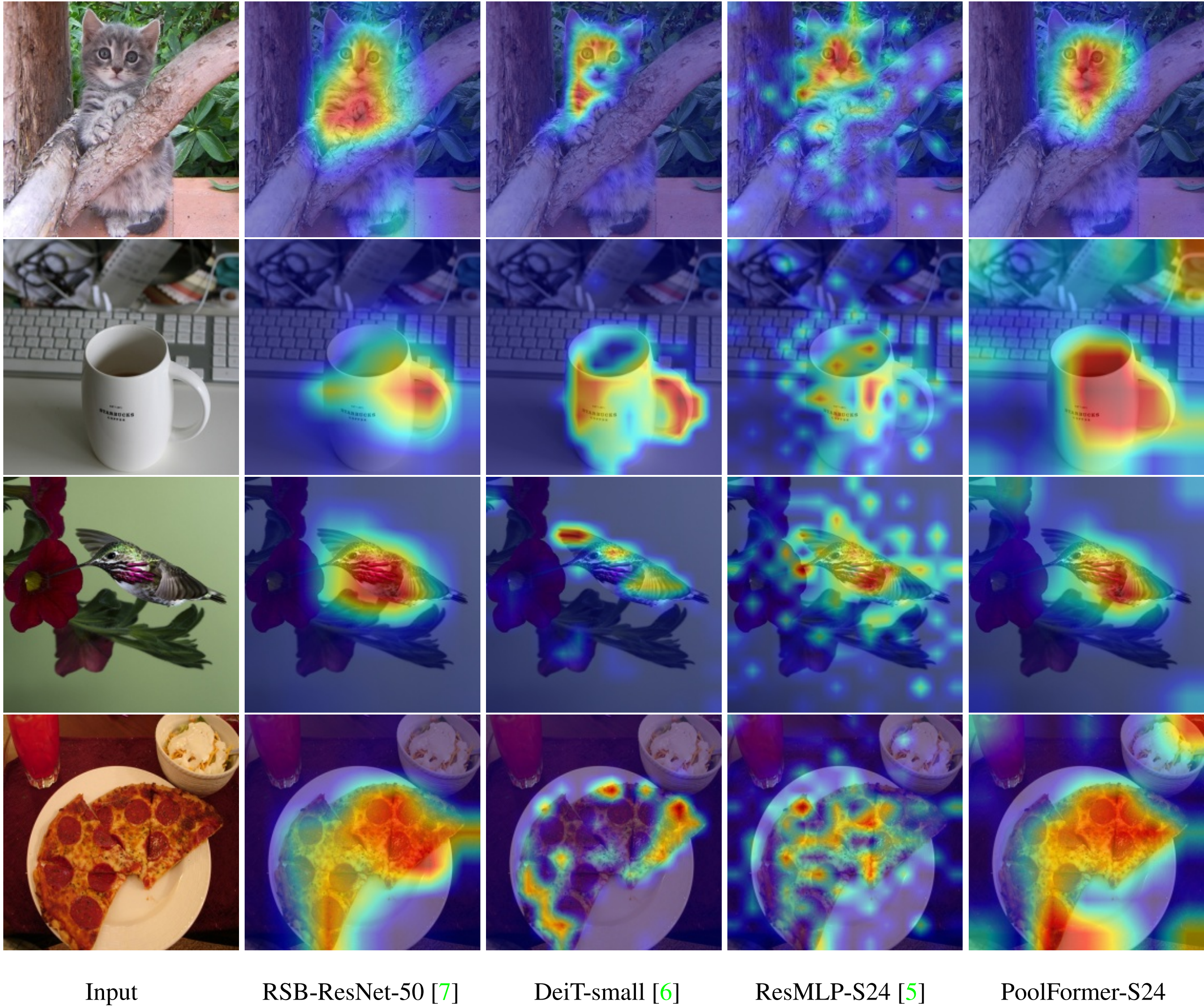

Figure 1. Grad-CAM [4] activation maps of the models trained on ImageNet-1K. The visualized images are from validation set.

Algorithm 1 Comparison between Layer Normalization and Modified Layer Normalization, PyTorch-like Code

```
import torch.nn as nn


class LayerNormChannel(nn.Module):
   """
   Vanilla Layer Normalization normalizes vectors along channel dimension.
   Input: tensor in shape [B, C, H, W].
   """
   def __init__(self, num_channels, eps=1e-05):
      super().__init__()
      # The shape of learnable affine parameters is [num_channels, ].
      self.weight = nn.Parameter(torch.ones(num_channels))
      self.bias = nn.Parameter(torch.zeros(num_channels))
      self.eps = eps

   def forward(self, x):
      u = x.mean(1, keepdim=True) # Compute the means along channel dimension.
      s = (x - u).pow(2).mean(1, keepdim=True) # Compute the variances along channel dimension.
      x = (x - u) / torch.sqrt(s + self.eps)
      x = self.weight.unsqueeze(-1).unsqueeze(-1) * x \
         + self.bias.unsqueeze(-1).unsqueeze(-1)
      return x


class ModifiedLayerNorm(nn.Module):
   """
   Modified Layer Normalization normalizes vectors along channel dimension and spatial dimensions.
   Input: tensor in shape [B, C, H, W]
   """
   def __init__(self, num_channels, eps=1e-05):
      super().__init__()
      # The shape of learnable affine parameters is also [num_channels, ], keeping the same as vanilla Layer
          Normalization.
      self.weight = nn.Parameter(torch.ones(num_channels))
      self.bias = nn.Parameter(torch.zeros(num_channels))
      self.eps = eps

   def forward(self, x):
      u = x.mean([1, 2, 3], keepdim=True) # Compute the mean along channel dimension and spatial dimensions.
      s = (x - u).pow(2).mean([1, 2, 3], keepdim=True) # Compute the variance along channel dimension and
          spatial dimensions.
      x = (x - u) / torch.sqrt(s + self.eps)
      x = self.weight.unsqueeze(-1).unsqueeze(-1) * x \
         + self.bias.unsqueeze(-1).unsqueeze(-1)
      return x


# Modified Layer Normalization can also be implemented using GroupNorm API in PyTorch by setting the group
    number as 1.
class ModifiedLayerNorm(nn.GroupNorm):
   """
   Modified Layer Normalization implemented by Group Normalization with 1 group.
   Input: tensor in shape [B, C, H, W]
   """
   def __init__(self, num_channels, **kwargs):
      super().__init__(1, num_channels, **kwargs)
```

Algorithm 2 Modules for PoolFormer block, PyTorch-like Code

```
import torch.nn as nn


class ModifiedLayerNorm(nn.GroupNorm):
    """
    Modified Layer Normalization implemented by Group Normalization with 1 group.
    Input: tensor in shape [B, C, H, W]
    """
    def __init__(self, num_channels, **kwargs):
        super().__init__(1, num_channels, **kwargs)


class Pooling(nn.Module):
    """
    Implementation of pooling for PoolFormer
    --pool_size: pooling size
    Input: tensor with shape [B, C, H, W]
    """
    def __init__(self, pool_size=3):
        super().__init__()
        self.pool = nn.AvgPool2d(
            pool_size, stride=1, padding=pool_size//2, count_include_pad=False)

    def forward(self, x):
        # Subtraction of the input itself is added
        # since the block already has a residual connection.
        return self.pool(x) - x


class Mlp(nn.Module):
    """
    Implementation of MLP with 1*1 convolutions.
    Input: tensor with shape [B, C, H, W]
    """
    def __init__(self, in_features, hidden_features=None,
                 out_features=None, act_layer=nn.GELU, drop=0.):
        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Conv2d(in_features, hidden_features, 1)
        self.act = act_layer()
        self.fc2 = nn.Conv2d(hidden_features, out_features, 1)
        self.drop = nn.Dropout(drop)
        self.apply(self._init_weights)

    def _init_weights(self, m):
        if isinstance(m, nn.Conv2d):
            trunc_normal_(m.weight, std=.02)
            if m.bias is not None:
                nn.init.constant_(m.bias, 0)

    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x
```

Algorithm 3 PoolFormer block, PyTorch-like Code

```
import torch.nn as nn


class PoolFormerBlock(nn.Module):
   """
   Implementation of one PoolFormer block.
   --dim: embedding dim
   --pool_size: pooling size
   --mlp_ratio: mlp expansion ratio
   --act_layer: activation
   --norm_layer: normalization
   --drop: dropout rate
   --drop path: Stochastic Depth,
      refer to https://arxiv.org/abs/1603.09382
   --use_layer_scale, --layer_scale_init_value: LayerScale,
      refer to https://arxiv.org/abs/2103.17239
   """
   def __init__(self, dim, pool_size=3, mlp_ratio=4.,
             act_layer=nn.GELU, norm_layer=ModifiedLayerNorm,
             drop=0., drop_path=0.,
             use_layer_scale=True, layer_scale_init_value=1e-5):

      super().__init__()

      self.norm1 = norm_layer(dim)
      self.token_mixer = Pooling(pool_size=pool_size)
      self.norm2 = norm_layer(dim)
      mlp_hidden_dim = int(dim * mlp_ratio)
      self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim,
                  act_layer=act_layer, drop=drop)

      # The following two techniques are useful to train deep PoolFormers.
      self.drop_path = DropPath(drop_path) if drop_path > 0. \
         else nn.Identity()
      self.use_layer_scale = use_layer_scale
      if use_layer_scale:
         self.layer_scale_1 = nn.Parameter(
            layer_scale_init_value * torch.ones(dim), requires_grad=True)
         self.layer_scale_2 = nn.Parameter(
            layer_scale_init_value * torch.ones(dim), requires_grad=True)

   def forward(self, x):
      if self.use_layer_scale:
         x = x + self.drop_path(
            self.layer_scale_1.unsqueeze(-1).unsqueeze(-1)
            * self.token_mixer(self.norm1(x)))
         x = x + self.drop_path(
            self.layer_scale_2.unsqueeze(-1).unsqueeze(-1)
            * self.mlp(self.norm2(x)))
      else:
         x = x + self.drop_path(self.token_mixer(self.norm1(x)))
         x = x + self.drop_path(self.mlp(self.norm2(x)))
      return x
```