ResMLP: Feedforward networks for image classification with data-efficient training

Ruba Mutasim

Key Idea:

ResMlp is a simple residual network for image classification that is built entirely on mlps. It falls under this line of research that attempts to revisit literature on convolutional neural network and other simple architectures to to move back form complex Networks,

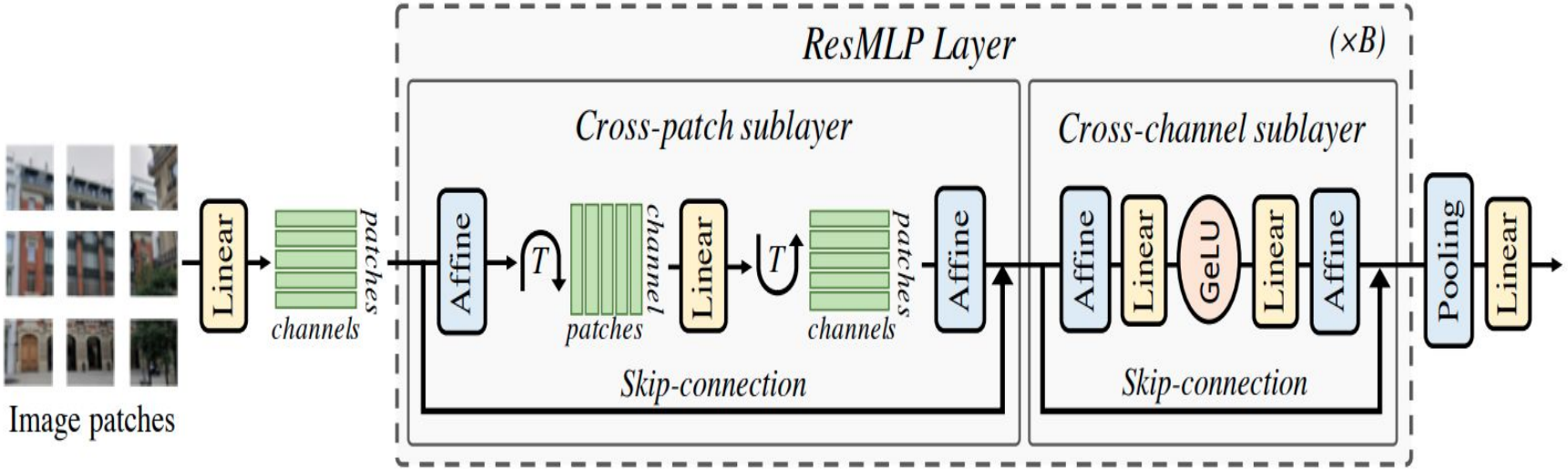
It does so by alternating

I. a linear layer in which image patches interact, independently and identically across channels
II. a two-layer feed-forward network in which channels interact independently per patch

When trained with a modern training strategy using heavy data-augmentation, it attains surprisingly good accuracy/complexity tradeoffs on ImageNet and many other datasets

ResMLP is designed to be simple and encoding little prior about images (namely the translation invariance and local connectivity) : it takes image patches as input, projects them with a linear layer, and sequentially updates their representations with the previous two residual operations

Architecture



Calculations

$$\texttt{Aff}_{\boldsymbol{\alpha},\boldsymbol{\beta}}(\mathbf{x}) = \texttt{Diag}(\boldsymbol{\alpha})\mathbf{x} + \boldsymbol{\beta},$$

$$\begin{aligned}
\mathbf{Z} &= \mathbf{X} + \texttt{Aff}\left(\left(\mathbf{A}\,\texttt{Aff}\left(\mathbf{X}\right)^\top\right)^\top\right), \\
\mathbf{Y} &= \mathbf{Z} + \texttt{Aff}\left(\mathbf{C}\,\texttt{GELU}(\mathbf{B}\,\texttt{Aff}(\mathbf{Z}))\right),
\end{aligned}$$

In Contrast to Vision Transformers

You may notice some similarities between ResMlp and VIT in fact, ResMlp is inspired by the vision transformer, with some simplifications to some parts:

- no self-attention blocks: it is replaced by a linear layer with no non-linearity (stable training)
- no positional embedding: the linear layer implicitly encodes information about patch positions (my hypothesis is by matrix indexing)
- no extra “class” token: Resmlp simply uses average pooling on the patch embeddings
- no normalization based on batch statistics: instead a learnable affine operator (no cost at inf time)

Algorithm

```
# No norm layer
class Affine(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.alpha = nn.Parameter(torch.ones(dim))
        self.beta = nn.Parameter(torch.zeros(dim))
    def forward(self, x):
        return self.alpha * x + self.beta

# MLP on channels
class Mlp(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.fc1 = nn.Linear(dim, 4 * dim)
        self.act = nn.GELU()
        self.fc2 = nn.Linear(4 * dim, dim)
    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.fc2(x)
        return x
```

Algorithm

```
# ResMLP blocks: a linear between patches + a MLP to process them independently
class ResMLP_BLocks(nn.Module):
   def __init__(self, nb_patches ,dim, layerscale_init):
      super().__init__()
      self.affine_1 = Affine(dim)
      self.affine_2 = Affine(dim)
      self.linear_patches = nn.Linear(nb_patches, nb_patches) #Linear layer on patches
      self.mlp_channels = Mlp(dim) #MLP on channels
      self.layerscale_1 = nn.Parameter(layerscale_init * torch.ones((dim))) #LayerScale
      self.layerscale_2 = nn.Parameter(layerscale_init * torch.ones((dim))) # parameters

   def forward(self, x):
      res_1 = self.linear_patches(self.affine_1(x).transpose(1,2)).transpose(1,2)
      x = x + self.layerscale_1 * res_1
      res_2 = self.mlp_channels(self.affine_2(x))
      x = x + self.layerscale_2 * res_2
      return x
```

Algorithm

```
# ResMLP model: Stacking the full network
class ResMLP_models(nn.Module):
   def __init__(self, dim, depth, nb_patches, layerscale_init, num_classes):
      super().__init__()
      self.patch_projector = Patch_projector()
      self.blocks = nn.ModuleList([
         ResMLP_BLocks(nb_patches ,dim, layerscale_init)
         for i in range(depth)])
      self.affine = Affine(dim)
      self.linear_classifier = nn.Linear(dim, num_classes)

   def forward(self, x):
      B, C, H, W = x.shape
      x = self.patch_projector(x)
      for blk in self.blocks:
         x = blk(x)
      x = self.affine(x)
      x = x.mean(dim=1).reshape(B,-1) #average pooling
      return self.linear_classifier(x)
```