Supplementary: Learning specialized activation functions with the Piecewise Linear Unit

1. A plug-and-play implementation for PWLU

Code 1: A plug-and-play implementation in PyTorch

```
import torch

class PWLU(torch.nn.Module):
  def __init__(self, N, T, ...):
    self.N = N
    self.count = 0
    self.T = T
    self.bl = torch.nn.Parameter(torch.Tensor(1).fill_(-3))
    self.br = torch.nn.Parameter(torch.Tensor(1).fill_(3))
    self.kl = torch.nn.Parameter(torch.Tensor(1).fill_(0))
    self.kr = torch.nn.Parameter(torch.Tensor(1).fill_(1))
    self.yp = torch.nn.Parameter(torch.Tensor(N + 1))
    self.register_buffer("running_mean", torch.zeros(1))
    self.register_buffer("running_std", torch.ones(1))
    self.momentum = 0.9
    self.fix = True
    ...

  def collect_stats(x):
    self.running_mean.mul_(self.momentum).add_(1 - self.momentum, x.mean())
    self.running_std.mul_(self.momentum).add_(1 - self.momentum, x.std())

  def reset_boundary():
    self.bl.data.copy_(self.running_mean - self.running_std * 3)
    self.br.data.copy_(self.running_mean + self.running_std * 3)
    dist = (self.br - self.bl) / self.N
    bpoints = [self.bl]
    for i in range(self.N):
      bpoints.append(bpoints[-1] + dist)
    bpoints = torch.Tensor(bpoints)
    self.yp.data.copy_(F.relu(bpoints))
    self.fix = False

  def forward(self, x):
    if self.train and self.count <= self.T:
      self.count += 1
      if self.count < self.T:
        self.collect_stats(x.detach())
      elif self.count == self.T:
        self.reset_boundary()
    return self.compute_pwlu(x)
```

Most of the scalar activation functions such as ReLU and Swish are plug-and-play modules, which is an important reason for their wide acceptance. However, the statistic-based realignment of PWLU breaks the training loop into two phases, making it inconvenient to apply the method. Here we show that PWLU can be implemented as a plug-and-play module as well.

As noted in Section 3.3, our method only requires slight changes to normal training procedures, *which are limited to PWLU itself* and do not affect any other components. So we can hide the phase changing into PWLU module, as shown in Code 1. We track the training iterations in the forward pass and call **collect_stats** or **reset_boundary** accordingly. The main computation **compute_pwlu** can be implemented according to Equation 1, which has been thoroughly explained. Under such implementation, PWLU can be easily applied without any change to the training loop, just like ReLU or Swish.