1 Scale Aware Attention

$\pi_L(F) \cdot F = \sigma(f(\frac{1}{SC} \sum_{S,C})) \cdot F$

where F is a feature map of shape [B, L, S, C], and f is a linear function approximated by a 1×1 convolutional layer.

In PyTorch implementation, we can model the following: $\frac{1}{S} \sum_S$ using **nn.AdaptiveAvgPool2d(1)** where our input is a Tensor of shape [B, C, H, W]. In official implementation, $f(\frac{1}{C} \sum_C)$ is combined into one step by applying a 1x1 convolution to approximate a learnable average over the channel dimension using **nn.Conv2d(in_channels=in_channels, out_channels=1, kernel_size=1, stride=1)**. As this transformation reduces our Tensor to a scalar, we can implement the dot product as an element-wise multiplcation with our initial Tensor.

```
nn.Sequential(
    nn.AdaptiveAvgPool2d(1),
    nn.Conv2d(in_channels=in_channels, out_channels=1, kernel_size=1,
        stride=1),
    nn.ReLU(),
    nn.Hardsigmoid()
)
x * nn.Sequential(x)
```

2 Spatial Aware Attention

$\pi_S(F) \cdot F = \frac{1}{L} \sum_L \sum_K w_{l,j} \cdot F(l; p_k + \Delta p_k; c) \cdot \Delta m_k$

where F is the transformed feature map after applying the scale aware attention.

$\sum_K w_{l,j} \cdot F(l; p_k + \Delta p_k; c) \cdot \Delta m_k$ is a modulated deformable convolution as described in [https://arxiv.org/abs/1811.11168] where Δp_k are learnable offset parameters and Δm_k are learnable modulation parameters, scaled between range [0, 1] to control the contribution of each filter element through learning. Both the offset and modulation (also referred to as the mask in variable naming) parameters are estimated by a separate convolutional layer with $3K$ output channels, where K is the $filtersize * filtersize$.

$\frac{1}{L} \sum_L$ can be intuitively implemented as a torch.mean(x, dim=0), where x is a stacked Tensor of shape [L, B, C, H, W].

One thing to note is that the order of attention transformations in code implementation and the official paper differ. In code, spatial-aware attention is applied first, as the modulated deformable convolutions can also be used for respective up/downscaling of the higher and lower feature maps for Tensor concatenation.

```
# Learnable offset and modulation parameters
OM = nn.Conv2d(in_channels=256, out_channels=K * 3, kernel_size=3,
    stride=1, padding=1)
offset_and_modulations = OM(x)
offset = offset_and_modulations[:, :18, :, :]
modulations = offset_and_modulations[:, 18:, :, :].sigmoid()

# Spatial up/downscaling modulated deformable convolutions
# where stride is 1 for retaining spatial resolution, and stride=2 for
    downscaling of the lower feature map.
DC = nn.DeformConv2d(in_channels=in_channels, out_channels=out_channels,
    kernel_size=3, stride=stride, padding=1, bias=False)

# We can first interpolate the higher feature map to the spatial
    resolution of the mdidle feature map before applying our deformable
    conv, which differs from the order of that in MMDet/Microsoft
low_feature = F.interpolate(low_feature, size=middel_feature, ...)

# deformable convolutions are applied as so
x = DC(x, offset, modulations)

# final Tensor concatenation and averaging over the L-dimension
F = torch.mean(torch.stack([low_feature, mid_feature, high_feature],
    dim=0), dim=0)
```

3 Task Aware Attention

$$\pi_C(F) \cdot (F) = max(\alpha^1(F) \cdot (F_c) + \beta^1(F), \alpha^2(F) \cdot (F_c) + \beta^2(F))$$

where F is a Tensor transformed by scale and spatial aware attentions, and F_c is the feature slice at the c-th channel.

The implementation of task aware attention is fairly straightforward and follows closely with the official paper, with the only noticeable difference being that MMDetection replaces the linear layers with 1x1 convolutions. By applying some additional tensor reshaping, we can use linear layers in the following way:

```
linear1 = nn.Linear(in_features=in_channels,
    out_features=in_channels//reduction)
linear2 = nn.Linear(in_features=in_channels//reduction,
    out_features=in_channels*expansion)
norm = nn.Hardsigmoid()

# applying a hardsigmoid to normalize values to [0, 1], then shifting
```

```
    our range to [-1, 1]
coefficients = norm(linear2(F.relu(linear1(x))))
coefficients = (coefficients - 0.5) * 2
# splitting our tensor of shape [B, 4C] to 4 * [B, C], where each index
    across the the first dimension corresponds to the coefficient to be
    used at the c-th feature slice.
a1, a2, b1, b2 = torch.split(coefficients, in_channels, dim=1)
return torch.max(a1 * x + b1, a2 * x + b2)
```

4 Inputs/Outputs

Inputs: tuple/list of feature maps extracted by the neck (FPN, PAN, etc). Tensors of shape [B, C, H, W]

Outputs: tuple/list of attention-applied feature maps. Tensors of shape [B, C, H, W]. In each iteration of the forward call, we take the higher and lower feature map (or just higher, or lower, depending on the level) and transform it into the 3-dimensional tensor of shape [B, L, S, C] as specified in the paper. However, final outputs are reshaped back into [B, C, H, W] after all transformations, as the spatial-awareness layer involves an averaging over the L dimension. These outputs are passed to the head.