# Appendix

Anonymous ECCV submission

Paper ID 1250

## 1 Detailed Derivations and Proofs for Sec. 3.1 & 3.2

### 1.1 Details for Sec. 3.1

To better present our proposed vMF classifier, we formulate Eq. 2 in submission PDF equivalently as:

$$
\begin{aligned}
p_i^l &= \frac{p_{\mathcal{D}}^{tra}(i) \cdot p(\tilde{\boldsymbol{x}}^l | \kappa_i, \tilde{\boldsymbol{\mu}}_i)}{\sum_{j=1}^{C} p_{\mathcal{D}}^{tra}(j) \cdot p(\tilde{\boldsymbol{x}}^l | \kappa_j, \tilde{\boldsymbol{\mu}}_j)} \\
&= \frac{\exp\{\ \kappa_i \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top + \overbrace{(\frac{d}{2}-1)\cdot \log \kappa_i - \log I_{\frac{d}{2}-1}(\kappa_i)}^{denoted\ as\ b_i} + \overbrace{\log n_i}^{prior}\ \}}{\sum_{j=1}^{C} \exp\{\ \kappa_j \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top + \underbrace{(\frac{d}{2}-1)\cdot \log \kappa_j - \log I_{\frac{d}{2}-1}(\kappa_j)}_{denoted\ as\ b_j} + \underbrace{\log n_j}_{prior}\ \}}.
\end{aligned}
\tag{1}
$$

Based on Eq. 1 in Appendix, we calculate the derivative of $p_i^l$ with respect to $\kappa_i$ as:

$$
\begin{aligned}
\frac{\partial p_i^l}{\partial \kappa_i} &= \frac{\partial p_i^l}{\partial(\kappa_i \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top + b_i)} \cdot (\frac{\partial(\kappa_i \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top)}{\partial \kappa_i} + \frac{\partial b_i}{\partial \kappa_i}) \\
&= (1 - p_i^l) \cdot (\tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top - A_d(\kappa_i)),
\end{aligned}
\tag{2}
$$

where $A_d(\kappa_i) = I_{d/2}(\kappa_i)/I_{d/2-1}(\kappa_i)$. The derivative of $p_i^l$ with respect to $\kappa_j$ is calculated as:

$$
\begin{aligned}
\frac{\partial p_i^l}{\partial \kappa_j} &= \frac{\partial p_i^l}{\partial(\kappa_j \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top + b_j)} \cdot (\frac{\partial(\kappa_j \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top)}{\partial \kappa_j} + \frac{\partial b_j}{\partial \kappa_j}) \\
&= -p_j^l \cdot (\tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top - A_d(\kappa_j)).
\end{aligned}
\tag{3}
$$

The derivatives of $p_i^l$ with respect to $\tilde{\boldsymbol{\mu}}_i$ and $\tilde{\boldsymbol{\mu}}_j$ are formulated as:

$$
\begin{aligned}
\frac{\partial p_i^l}{\partial \tilde{\boldsymbol{\mu}}_i} &= \frac{\partial p_i^l}{\partial(\kappa_i \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top)} \cdot \frac{\partial(\kappa_i \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top)}{\partial \tilde{\boldsymbol{\mu}}_i} = (1 - p_i^l) \cdot \kappa_i \cdot \tilde{\boldsymbol{x}}^l \\
\frac{\partial p_i^l}{\partial \tilde{\boldsymbol{\mu}}_j} &= \frac{\partial p_i^l}{\partial(\kappa_j \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top)} \cdot \frac{\partial(\kappa_j \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top)}{\partial \tilde{\boldsymbol{\mu}}_j} = -p_j^l \cdot \kappa_j \cdot \tilde{\boldsymbol{x}}^l.
\end{aligned}
\tag{4}
$$

**Implement Details**. Both forward and backward operations with respect to $b_i$ are **not supported** by **Pytorch** [3] framework. In addition, the floating point precision for $I_v(\kappa)$ with the large $v$ and small $\kappa$ (e.g., $v = 511$ and $\kappa = 16$) exceeds *float*64 which is the maximum floating point precision of **CUDA**. While the floating point precision for $\log I_v(\kappa)$ is in the normal interval.

To implement our method, we first calculate $b_i$ and its derivative by **mpmath** [1] library which allows the floating pointing operation with arbitrary precision. Then, we convert them to the data type of **Pytorch**. Here is our core code for the above steps:

```python
import mpmath as mp
import numpy as np
import torch
Iv = np.frompyfunc(mp.besseli , 2, 1) # Bessel Function I_v()
log = np.frompyfunc(mp.log , 1, 1) #  Logarithmic Function
# Forward and backward functions for b_i
class Function_Bias(torch.autograd.Function):
    @staticmethod
    def forward(self , d, kappa):
        self.k = kappa.data.cpu().numpy()
        self.v = d / 2 - 1
        bias = self.v * log(self.k) - log(Iv(self.v, self.k))
        bias = torch.Tensor([float(bias)]).type_as(kappa)
        self.save_for_backward(kappa)
        return bias
    @staticmethod
    def backward(self , grad_output):
        kappa = self.saved_tensors[-1]
        Adk = Iv(self.v+1, self.k) / Iv(self.v, self.k)
        Adk = torch.Tensor([float(Adk)]).type_as(kappa)
        return None, - grad_output * Adk
```

See core code in supplement material for more implement details.

## 1.2 Details for Sec. 3.2

For simplification, we abbreviate $o_\Lambda(\kappa_i, \kappa_j, \tilde{\boldsymbol{\mu}}_i, \tilde{\boldsymbol{\mu}}_j)$ as $o_\Lambda$. To better present the distribution overlap coefficient, we formulate Eq. 6 in submission PDF equivalently as:

$$\begin{aligned} KL_{ij} &= \ln \frac{C_d(\kappa_i)}{C_d(\kappa_j)} + A_d(\kappa_i) \cdot (\kappa_i - \kappa_j \cdot \tilde{\boldsymbol{\mu}}_i \tilde{\boldsymbol{\mu}}_j^\top) \\ &= b_i - b_j + A_d(\kappa_i) \cdot (\kappa_i - \kappa_j \cdot \tilde{\boldsymbol{\mu}}_i \tilde{\boldsymbol{\mu}}_j^\top). \end{aligned} \tag{5}$$

The derivatives of $o_\Lambda$ with respect to $\kappa_i$ and $\kappa_j$ are formulated as:

$$\begin{aligned} \frac{\partial o_\Lambda}{\partial \kappa_i} &= \frac{\partial o_\Lambda}{\partial KL_{ij}} \cdot \frac{\partial KL_{ij}}{\partial \kappa_i} = o_\Lambda^2 \cdot \frac{\partial A_d(\kappa_i)}{\partial \kappa_i} \cdot (-\kappa_i + \kappa_j \cdot \tilde{\boldsymbol{\mu}}_i \tilde{\boldsymbol{\mu}}_j^\top) \\ \frac{\partial o_\Lambda}{\partial \kappa_j} &= \frac{\partial o_\Lambda}{\partial KL_{ij}} \cdot \frac{\partial KL_{ij}}{\partial \kappa_j} = o_\Lambda^2 \cdot (-A_d(\kappa_j) + A_d(\kappa_i) \cdot \tilde{\boldsymbol{\mu}}_i \tilde{\boldsymbol{\mu}}_j^\top), \end{aligned} \tag{6}$$

where the derivative of $A_d(\kappa_i)$ with respect to $\kappa_i$ is defined as:

$$\frac{\partial A_d(\kappa_i)}{\partial \kappa_i} = 1 - \frac{d-1}{\kappa_i} \cdot A_d(\kappa_i) - A_d^2(\kappa_i). \tag{7}$$

The derivatives with respect to $\tilde{\boldsymbol{\mu}}_i$ and $\tilde{\boldsymbol{\mu}}_j$ are easily derived, following the above operations. The results are demonstrated in Tab. 1 of the submission PDF.

**Implement Details**. Facing the same case as $b_i$, we need to define the forward and backward functions of $A_d(\kappa_i)$ manually. The core code is demonstrated as:

```python
import mpmath as mp
import numpy as np
import torch
Iv = np.frompyfunc(mp.besseli, 2, 1) # Bessel Function I_v()
# Forward and backward functions for A_d(\kappa)
class Function_Adk(torch.autograd.Function):
    @staticmethod
    def forward(self, d, kappa):
        k = kappa.data.cpu().numpy()
        self.d, v = d, d / 2 - 1
        Adk = Iv(v+1, k) / Iv(v, k)
        Adk = torch.Tensor([float(Adk)]).type_as(kappa)
        self.save_for_backward(kappa, Adk)
        return Adk
    @staticmethod
    def backward(self, grad_output):
        kappa, Adk = self.saved_tensors
        grad_Adk = 1 - (self.d - 1) / kappa * Adk - Adk ** 2
        return None, grad_output * grad_Adk
```

## 2 Relation with Other classifiers

### 2.1 Balanced Cosine Classifier

Setting $\kappa_i = const\ \sigma, \forall i \in [1, C]$, Eq. 2 in submission PDF can be re-write as:

$$\begin{aligned} p_i^l &= \frac{p_{\mathcal{D}}^{tra}(i) \cdot p(\tilde{\boldsymbol{x}}^l | \sigma, \tilde{\boldsymbol{\mu}}_i)}{\sum_{j=1}^{C} p_{\mathcal{D}}^{tra}(j) \cdot p(\tilde{\boldsymbol{x}}^l | \sigma, \tilde{\boldsymbol{\mu}}_j)} \\ &= \frac{n_i \cdot \exp\{\sigma \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_i^\top\}}{\sum_{j=1}^{C} n_j \cdot \exp\{\sigma \cdot \tilde{\boldsymbol{x}}^l \tilde{\boldsymbol{\mu}}_j^\top\}}. \end{aligned} \tag{8}$$

Consequently, the balanced cosine classifier can be considered a special case of our vMF classifier.

### 2.2 Convert Other Classifiers into Ours

In this paper, we take three classifiers into account, including linear, $\tau$-norm, and causal classifiers, following the default setting (i.e., ignoring the bias terms). To measure the distribution overlap coefficient of them above, we develop a conversion method to convert them in a vMF classifier way.

For the linear classifier, given a feature vector $\boldsymbol{x} \in \mathbb{R}^{1\times d}$ and classifier weights $\boldsymbol{W}^{lin} = \{\boldsymbol{w}_1^{lin}, ..., \boldsymbol{w}_i^{lin}, ..., \boldsymbol{w}_C^{lin}\} \in \mathbb{R}^{C\times d}$, the score for class $i$ can be defined as:

$$s_i^{lin} = \boldsymbol{w}_i^{lin}\boldsymbol{x}^\top = \underbrace{\|\boldsymbol{w}_i^{lin}\|_2}_{compactness} \cdot \overbrace{\frac{\boldsymbol{w}_i^{lin}}{\|\boldsymbol{w}_i^{lin}\|_2}}^{orientation} \boldsymbol{x}^\top. \tag{9}$$

For the $\tau$-norm classifier, given a feature vector $\boldsymbol{x} \in \mathbb{R}^{1\times d}$ and classifier weights $\boldsymbol{W}^{\tau} = \{\boldsymbol{w}_1^{\tau}, ..., \boldsymbol{w}_i^{\tau}, ..., \boldsymbol{w}_C^{\tau}\} \in \mathbb{R}^{C\times d}$, the score for class $i$ can be defined as:

$$s_i^{\tau} = \frac{\boldsymbol{w}_i^{\tau}}{\|\boldsymbol{w}_i^{\tau}\|_2^{\tau}}\tilde{\boldsymbol{x}}^\top = \underbrace{\|\boldsymbol{w}_i^{\tau}\|_2^{1-\tau}}_{compactness} \cdot \overbrace{\frac{\boldsymbol{w}_i^{\tau}}{\|\boldsymbol{w}_i^{\tau}\|_2}}^{orientation} \tilde{\boldsymbol{x}}^\top. \tag{10}$$

For the causal classifier, given a feature vector $\boldsymbol{x} \in \mathbb{R}^{1\times d}$ and classifier weights $\boldsymbol{W}^{cau} = \{\boldsymbol{w}_1^{cau}, ..., \boldsymbol{w}_i^{cau}, ..., \boldsymbol{w}_C^{cau}\} \in \mathbb{R}^{C\times d}$, the score for class $i$ can be defined as:

$$s_i^{cau} = \frac{\boldsymbol{w}_i^{cau}}{\|\boldsymbol{w}_i^{cau}\|_2 + \gamma}\tilde{\boldsymbol{x}}^\top = \underbrace{\frac{\|\boldsymbol{w}_i^{cau}\|_2}{\|\boldsymbol{w}_i^{cau}\|_2 + \gamma}}_{compactness} \cdot \overbrace{\frac{\boldsymbol{w}_i^{cau}}{\|\boldsymbol{w}_i^{cau}\|_2}}^{orientation} \tilde{\boldsymbol{x}}^\top. \tag{11}$$

In our experiment, the optimal setting $\tau$ of $\tau$-norm classifier [2] is equal to 0.7. $\gamma$ of the causal classifier [5] is set as $1/16$, following the official codes. For the causal classifier, we do not apply the causal post-processing algorithm proposed by them. In addition, our ablation study on post-training calibration algorithm with different classifiers is shown in Tab. 4 of submission PDF.

## 3 Ablation Study on Initializing $\mathcal{K}$

## 4 Long-tailed Image Classification Task on Places-LT

**Statement**: we are so sorry for ignoring **Submission Guidelines for supplement material** that "It may not include results on additional datasets, results obtained with an improved version of the method (e.g., following additional parameter tuning or training), or an updated or corrected version of the submission PDF". Therefore, we cannot provide the experimental results of ablation study on initializing $\mathcal{K}$ and the experimental results of image classification task on

Places-LT mentioned in submission PDF. **We will add the above results in the future version of our paper**.

In this section, we briefly describe how to initialize $\kappa$ for our vMF classifier on ImageNet-LT. The first initialization way is to initialize all $\kappa$ for 1000 classes as 16 (is proven to be the optimal setting on ImageNet-LT for the related classifier [4]). The second way is to initialize all $\kappa$ for 1000 classes by randomly sampling 1000 values from a Gaussian distribution with mean 16 and standard deviation 1. The above two ways get the approximate results.

In addition, we also conduct experiments on initializing $\kappa$ with different values (8, 16, 24, 32), respectively. The optimal value is 16.

For the initialization of $\kappa$ on ADE20K and LVIS-v1.0, we keep the default setting on ImageNet-LT (i.e., initialize $\kappa = 16$) without tuning, limited by computation resource.

## References

1. Johansson, F., et al.: mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18) (December 2013)
2. Kang, B., Xie, S., Rohrbach, M., Yan, Z., Gordo, A., Feng, J., Kalantidis, Y.: Decoupling representation and classifier for long-tailed recognition (2019)
3. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A.: Automatic differentiation in pytorch (2017)
4. Ren, J., Yu, C., Ma, X., Zhao, H., Yi, S., et al.: Balanced meta-softmax for long-tailed visual recognition. Advances in Neural Information Processing Systems **33**, 4175–4186 (2020)
5. Tang, K., Huang, J., Zhang, H.: Long-tailed classification by keeping the good and removing the bad momentum causal effect. In: NeurIPS (2020)