Generalized Data Weighting via Class-level Gradient Manipulation

Can Chen¹, Shuhao Zheng¹, Xi Chen¹, Erqun Dong¹, Xue Liu¹, Hao Liu², Dejing Dou³

¹McGill University, ²The Hong Kong University of Science and Technology, ³Baidu Research
{can.chen, shuhao.zheng, erqun.dong}@mail.mcgill.ca
xi.chen11@mcgill.ca, xueliu@cs.mcgill.ca, liuh@ust.hk, doudejing@baidu.com

Abstract

Label noise and class imbalance are two major issues coexisting in real-world datasets. To alleviate the two issues, state-of-the-art methods reweight each instance by leveraging a small amount of clean and unbiased data. Yet, these methods overlook class-level information within each instance, which can be further utilized to improve performance. To this end, in this paper, we propose Generalized Data Weighting (GDW) to simultaneously mitigate label noise and class imbalance by manipulating gradients at the class level. To be specific, GDW unrolls the loss gradient to class-level gradients by the chain rule and reweights the flow of each gradient separately. In this way, GDW achieves remarkable performance improvement on both issues. Aside from the performance gain, GDW efficiently obtains class-level weights without introducing any extra computational cost compared with instance weighting methods. Specifically, GDW performs a gradient descent step on class-level weights, which only relies on intermediate gradients. Extensive experiments in various settings verify the effectiveness of GDW. For example, GDW outperforms state-of-the-art methods by 2.56% under the 60% uniform noise setting in CIFAR10. Our code is available at https://github.com/GGchen1997/GDW-NIPS2021.

1 Introduction

Real-world classification datasets often suffer from two issues, i.e., label noise [1] and class imbalance [2]. On the one hand, label noise often results from the limitation of data generation, e.g., sensor errors [3] and mislabeling from crowdsourcing workers [4]. Label noise misleads the training process of DNNs and degrades the model performance in various aspects [5, 6, 7]. On the other hand, imbalanced datasets are either naturally long-tailed [8, 9] or biased from the real-world distribution due to imperfect data collection [10, 11]. Training with imbalanced datasets usually results in poor classification performance on weakly represented classes [12, 13, 14]. Even worse, these two issues often coexist in real-world datasets [15].

To prevent the model from memorizing noisy information, many important works have been proposed, including label smoothing [16], noise adaptation [17], importance weighting [18], GLC [19], and Co-teach [20]. Meanwhile, [12, 13, 14, 21] propose effective methods to tackle class imbalance. However, these methods inevitably introduce hyper-parameters (e.g., the weighting factor in [13] and the focusing parameter in [21]), compounding real-world deployment.

Inspired by recent advances in meta-learning, some works [22, 23, 24, 25] propose to solve both issues by leveraging a clean and unbiased meta set. These methods treat instance weights as hyper-

*Equal contribution; Names listed in alphabetical order.
parameters and dynamically update these weights to circumvent hyper-parameter tuning. Specifically, MWNet [23] adopts an MLP with the instance loss as input and the instance weight as output. Due to the MLP, MWNet has better scalability on large datasets compared with INSW [24] which assigns each instance with a learnable weight. Although these methods can handle label noise and class imbalance to some extent, they cannot fully utilize class-level information within each instance, resulting in the potential loss of useful information. For example, in a three-class classification task, every instance has three logits. As shown in Figure 1, every logit corresponds to a class-level gradient flow which stems from the loss function and back-propagates. These gradient flows represent three kinds of information: "not cat", "dog", and "not bird". Instance weighting methods [23, 22] alleviate label noise by downweighting all the gradient flows of the instance, which discards three kinds of information simultaneously. Yet, downweighting the "not bird" gradient flow is a waste of information. Similarly, in class imbalance scenarios, different gradient flows represent different class-level information. Therefore, it is necessary to reweight instances at the class level for better information usage.

To this end, we propose Generalized Data Weighting (GDW) to tackle label noise and class imbalance by class-level gradient manipulation. Firstly, we introduce class-level weights to represent the importance of different gradient flows and manipulate the gradient flows with these class-level weights. Secondly, we impose a zero-mean constraint on class-level weights for stable training. Thirdly, to efficiently obtain class-level weights, we develop a two-stage weight generation scheme embedded in bi-level optimization. As a side note, the instance weighting methods [22, 23, 24, 25] can be considered special cases of GDW when class-level weights within any instance are the same. In this way, GDW achieves impressive performance improvement in various settings.

To sum up, our contribution is two-fold:

1. For better information utilization, we propose GDW, a generalized data weighting method, which better handles label noise and class imbalance. To the best of our knowledge, we are the first to propose single-label class-level weighting on gradient flows.

2. To obtain class-level weights efficiently, we design a two-stage scheme embedded in a bi-level optimization framework, which does not introduce any extra computational cost. To be specific, during the back-propagation we store intermediate gradients, with which we update class-level weights via a gradient descent step.

2 Related Works

2.1 Non-Meta-Learning Methods for Label Noise

Label noise is a common problem in classification tasks [6, 7, 5]. To avoid overfitting to label noise, [16] propose label smoothing to regularize the model. [17, 26] form different models to indicate the relation between noisy instances and clean instances. [18] estimate an importance weight for each instance to represent its value to the model. [20] train two models simultaneously and let them teach each other in every mini-batch. However, without a clean dataset, these methods cannot handle severe noise [22, 19] correct the prediction of the model by estimating the label corruption matrix via a clean validation set, but this matrix is the same across all instances. Instead, our method generates dynamic class-level weights for every instance to improve training.

2.2 Non-Meta-Learning Methods for Class Imbalance

Many important works have been proposed to handle class imbalance [27, 28, 29, 12, 30, 13, 31, 21, 14, 28, 29] propose to over-sample the minority class and under-sample the majority class. [27, 30]
Table 1: Related works comparison. "Noise" and "Imbalance" denote whether the method can solve label noise and class imbalance. "Class-level" denotes whether the method utilizes class-level information in each instance, and "Scalability" denotes whether the method can scale to large datasets.

<table>
<thead>
<tr>
<th>Method</th>
<th>Noise</th>
<th>Imbalance</th>
<th>Class-level</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Focal [21]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Balanced [13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Co-teaching [20]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GLC [19]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>L2RW [22]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>INSW [24]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MWNet [23]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Soft-label [42]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Gen-label [40]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>GDW</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

learn a class-dependent cost matrix to obtain robust representations for both majority and minority classes. [12, 13, 21, 14] design a reweighting scheme to rebalance the loss for each class. These methods are quite effective, whereas they need to manually choose loss functions or hyper-parameters.

[32, 33] manipulate the feature space to handle class imbalance while introducing extra model parameters. [31] decouple representation learning and classifier learning on long-tailed datasets, but with extra hyper-parameter tuning. In contrast, meta-learning methods view instance weights as hyper-parameters and dynamically update them via a meta set to avoid hyper-parameter tuning.

2.3 Meta-Learning Methods

With recent development in meta-learning [34, 35, 36], many important methods have been proposed to handle label noise and class imbalance via a meta set [37, 38, 22, 23, 24, 25, 40]. [38] propose MentorNet to provide a data-driven curriculum for the base network to focus on correct instances. To distill effective supervision, [41] estimate pseudo labels for noisy instances with a meta set. To provide dynamic regularization, [42, 43] treat labels as learnable parameters and adapt them to the model’s state. Although these methods can tackle label noise, they introduce huge amounts of learnable parameters and thus cannot scale to large datasets. To alleviate class imbalance, [37] describe a method to learn from long-tailed datasets. Specifically, [37] propose to encode meta-knowledge into a meta-network and model the tail classes by transfer learning.

Furthermore, many meta-learning methods propose to mitigate the two issues by reweighting every instance [22, 43, 23, 24, 25]. [43] equip each instance and each class with a learnable parameter to govern their importance. By leveraging a meta set, [22, 23, 24, 25] learn instance weights and model parameters via bi-level optimization to tackle label noise and class imbalance. [22] assign weights to training instances only based on their gradient directions. Furthermore, [24] combine reinforce learning and meta-learning, and treats instance weights as rewards for optimization. However, since each instance is directly assigned with a learnable weight, INSW can not scale to large datasets. Meanwhile, [23, 25] adopt a weighting network to output weights for instances and use bi-level optimization to jointly update the weighting network parameters and model parameters. Although these methods handle label noise and class imbalance by reweighting instances, a scalar weight for every instance cannot capture class-level information, as shown in Figure 1. Therefore, we introduce class-level weights for different gradient flows and adjust them to better utilize class-level information.

We show the differences between GDW and other related methods in Table I.

3 Method

3.1 Notations

In most classification tasks, there is a training set $D_{train} = \{(x_i, y_i)\}_{i=1}^N$ and we assume there is also a clean unbiased meta set $D_{meta} = \{(x_i^v, y_i^v)\}_{i=1}^M$. We aim to alleviate label noise and class imbalance in $D_{train}$ with $D_{meta}$. The model parameters are denoted as $\theta$, and the number of classes is denoted as $C$.

3.2 Class-level Weighting by Gradient Manipulation

To utilize class-level information, we learn a class-level weight for every gradient flow instead of a scalar weight for all $C$ gradient flows in [23]. Denote $L$ as the loss of any instance. Applying the
chain rule, we unroll the gradient of $\mathcal{L}$ w.r.t. $\theta$ as
\begin{equation}
\nabla_{\theta} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \theta} = \frac{\partial \mathcal{L}}{\partial \theta} \frac{\partial \theta}{\partial \theta} = D_1 D_2,
\end{equation}
where $I \in \mathbb{R}^C$ represents the predicted logit vector of the instance. We introduce class-level weights $\omega \in \mathbb{R}^C$ and denote the $j^{th}$ component of $\omega$ as $\omega_j$. To indicate the importance of every gradient flow, we perform an element-wise product $f_{\omega}(\cdot)$ on $D_1$ with $\omega$. After this manipulation, the gradient becomes
\begin{equation}
f_{\omega} (\nabla_{\theta} \mathcal{L}) \doteq (\omega \otimes \frac{\partial \mathcal{L}}{\partial \theta}) \frac{\partial \theta}{\partial \theta} = (\omega \otimes D_1) D_2 \equiv D'_1 D_2,
\end{equation}
where $\otimes$ denotes the element-wise product of two vectors. Note that $\omega_j$ represents the importance of the $j^{th}$ gradient flow. Obviously, instance weighting is a special case of GDW when elements of $\omega$ are the same. Most classification tasks \cite{44,45,46} adopt the Softmax-CrossEntropy loss. In this case, we have $D_1 = p - y$, where $p \in \mathbb{R}^C$ denotes the probability vector output by softmax and $y \in \mathbb{R}^C$ denotes the one-hot label of the instance (see Appendix A for details).

As shown in Figure 1 for a noisy instance (e.g., cat mislabeled as "dog"), instance weighting methods assign a low scalar weight to all gradient flows of the instance. Instead, GDW assigns class-level weights to different gradient flows by leveraging the meta set. In other words, GDW tries to downweight the gradient flows for "dog" and "not cat", and upweight the gradient flow for "not bird". Similarly, in imbalance settings, different gradient flows have different class-level information. Thus GDW can also better handle class imbalance by adjusting the importance of different gradient flows.

### 3.3 Zero-mean Constraint on Class-level Weights

To retain the Softmax-CrossEntropy loss structure, i.e. the $p - y$ form, after the manipulation, we impose a zero-mean constraint on $D'_1$. That is, we analyze the $j^{th}$ element of $D'_1$ (see Appendix B.1 for details):
\begin{equation}
\omega_j (p_j - y_j) = \omega_t \left(p'_j - y_j\right) + \left(\sum_k \omega_k p_k - \omega_t\right) p'_j,
\end{equation}
where $p'_j \doteq \frac{\omega_j p_j}{\sum_k \omega_k p_k}$ is the weighted probability, and $\omega_t$ denotes the class-level weight at the target (label) position. We observe that the first term in Eq. \cite{3} satisfies the structure of the gradient of the Softmax-CrossEntropy loss, and thus propose to eliminate the second term which messes the structure. Specifically, we let
\begin{equation}
\sum_k \omega_k p_k - \omega_t = 0 \Rightarrow \omega_t = \frac{\sum_j \omega_j p_j}{1 - p_t},
\end{equation}
where $p_t$ is the probability of the target class. Note that $\sum_j \omega_j y_j = \omega_t$, and thus we have
\begin{equation}
\sum_j \omega_j (p_j - y_j) = 0.
\end{equation}
This restricts the mean of $D'_1$ to be zero. Therefore, we name this constraint as the zero-mean constraint. With this, we have
\begin{equation}
D'_1 = \omega_t (p' - y).
\end{equation}

Eq. \cite{3} indicates that $\omega$ adjust the gradients in two levels, i.e., instance level and class level. Namely, the scalar $\omega_t$ acts as the instance-level weight in previous instance weighting methods \cite{22,23,24,25}, and the $\omega_j$'s are the class-level weights manipulating gradient flows by adjusting the probability from $p$ to $p'$.

### 3.4 Efficient Two-stage Weight Generation Embedded in Bi-level Optimization

In this subsection, we first illustrate the three-step bi-level optimization framework in \cite{23}. Furthermore, we embed a two-stage scheme in the bi-level optimization framework to efficiently obtain class-level weights, with which we manipulate gradient flows and optimize model parameters.

4
We keep the notations of \( \theta \) where \( n \) is the mini-batch size, \( \omega \) is the first-stage (instance-level) weight and \( \omega' \) is the second-stage (class-level) weight.

**Three-step Bi-level Optimization.** Generally, the goal of classification tasks is to obtain the optimal model parameters \( \theta^* \) by minimizing the average loss on \( D_{\text{train}} \), denoted as \( \frac{1}{N} \sum_{i=1}^{N} l_{\text{train}}(x_i, y_i; \theta) \). As an instance weighting method, [23] adopt a three-layer MLP parameterized by \( \phi \) as the weighting network and take the loss of the \( i^{th} \) instance as input and output a scalar weight \( \omega_i \). Then \( \theta^* \) is optimized by minimizing the instance-level weighted training loss:

\[
\theta^*(\phi) = \arg\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} \omega_i(\phi) l_{\text{train}}(x_i, y_i; \theta).
\] (7)

To obtain the optimal \( \omega_i \), they propose to use a meta set as meta-knowledge and minimize the meta-loss to obtain \( \phi^* \):

\[
\phi^* = \arg\min_{\phi} \frac{1}{M} \sum_{i=1}^{M} l_{val}(x_i^v, y_i^v; \theta^*(\phi)).
\] (8)

Since the optimization for \( \theta^*(\phi) \) and \( \phi^* \) is nested, they adopt an online strategy to update \( \theta \) and \( \phi \) with a three-step optimization loop for efficiency. Denote the two sets of parameters at the \( r^{th} \) loop as \( \theta_r \) and \( \phi_r \) respectively, and then the three-step loop is formulated as:

**Step 1** Update \( \theta_{r-1} \) to \( \hat{\theta}_r(\phi) \) via an SGD step on a mini-batch training set by Eq. [7].

**Step 2** With \( \hat{\theta}_r(\phi) \), update \( \phi_{r-1} \) to \( \phi_r \) via an SGD step on a mini-batch meta set by Eq. [8].

**Step 3** With \( \phi_r \), update \( \theta_{r-1} \) to \( \theta_r \) via an SGD step on the same mini-batch training set by Eq. [7].

Instance weights in Step 3 are better than those in Step 1, and thus are used to update \( \theta_{r-1} \).

**Two-stage Weight Generation.** To guarantee scalability, we apply the same weighting network in [23] to obtain weights. To efficiently train \( \phi \) and \( \theta \), we also adopt the three-step bi-level optimization framework. Moreover, we propose an efficient two-stage scheme embedded in Step 1-3 to generate class-level weights. This process does not introduce any extra computational cost compared to [23]. We keep the notations of \( \theta_r \) and \( \phi_r \) unchanged.

The first stage is embedded in Step 1. Explicitly, we obtain the first-stage class-level weights \( \omega_i = \omega_i 1 \), by cloning the output of the weighting network for \( C \) times. Then we leverage the cloned weights \( \omega_i \) to manipulate gradients and update \( \theta \) with a mini-batch of training instances:

\[
\hat{\theta}_r(\phi_{r-1}) \leftarrow \theta_{r-1} - \eta_{\theta} \frac{1}{n} \sum_{i=1}^{n} f_{\omega_i(\phi_{r-1})} \left( \nabla_{\theta} l_{\text{train}}(x_i, y_i; \theta_{r-1}) \right),
\] (9)

where \( n \) is the mini-batch size, \( \eta_{\theta} \) is the learning rate of \( \theta \), and \( f_{\omega_i(\phi_{r-1})}(\cdot) \) is the gradient manipulation operation defined in Eq. [2].

The second stage is embedded in Step 2 and Step 3. Specifically in Step 2, GDW optimizes \( \phi \) with a mini-batch meta set:

\[
\phi_r \leftarrow \phi_{r-1} - \eta_{\phi} \frac{1}{m} \sum_{i=1}^{m} \nabla_{\phi_r, \hat{\theta}_r(\phi_{r-1})} l_{\text{meta}}(x_i^v, y_i^v; \hat{\theta}_r(\phi_{r-1})),
\] (10)

where \( m \) is the mini-batch size and \( \eta_{\phi} \) is the learning rate of \( \phi \). During the back-propagation in updating \( \phi_r \), GDW generates the second-stage weights using the intermediate gradients \( g_r \) on \( \omega_i \).
We conduct extensive experiments on classification tasks to examine the performance of GDW. We mainly compare GDW with meta-learning methods: 1) L2RW \cite{22}, which assigns weights to instances based on gradient directions; 2) INSW \cite{24}, which derives instance weights adaptively from the meta set; 3) MWNet \cite{23}; 4) Soft-label \cite{42}, which learns a label-smoothing parameter for every instance; 5) Gen-label \cite{40}, which generates a meta-soft-label for every instance. We also compare GDW with some traditional methods: 6) BaseModel, which trains ResNet-32 on the noisy training set; 7) Fine-tuning, which uses the meta set to fine-tune the trained model in BaseModel; 8) Co-teaching \cite{20}; 9) GLC \cite{19}.

\begin{algorithm}[h]
\begin{algorithmic}
   \State \textbf{Input:} Training set: $D_{train}$, Meta set: $D_{meta}$, batch size $n$, m, \# of iterations $T$
   \State Initial model parameters: $\theta_0$, initial weighting network parameters: $\phi_0$
   \State \textbf{Output:} Trained model: $\theta_T$
   \For{$\tau \leftarrow 1$ \textbf{to} $T$}
   \State $\{x_i, y_i\}_{i=1}^m \leftarrow \text{SampleFrom}(D_{train})$
   \State $\{x_i, y_i\}_{i=1}^m \leftarrow \text{SampleFrom}(D_{meta})$
   \State Generate $\omega_i$ via the weighting network parameterized by $\phi_{r-1}$
   \State Manipulate gradients via Eq. \eqref{eq:manipulate1} and update $\theta_r$ by Eq. \eqref{eq:theta_update1}
   \State Update $\phi_r$ by Eq. \eqref{eq:phi_update1}:
   \State Update $\omega_r$ to $\omega'_r$ by Eq. \eqref{eq:omega_update1} and constrain $\omega'_r$ by Eq. \eqref{eq:clip1}
   \State Manipulate gradients with $\omega'_r$ via Eq. \eqref{eq:manipulate1} and update $\theta_r$ by Eq. \eqref{eq:theta_update2}
   \EndFor
\end{algorithmic}
\caption{Generalized Data Weighting via Class-Level Gradients Manipulation}
\end{algorithm}

4 Experiments

We conduct extensive experiments on classification tasks to examine the performance of GDW. We compare GDW with other methods in the label noise setting and class imbalance setting in Section 4.1 and Section 4.2, respectively. Next, we perform experiments on the real-world dataset Clothing1M \cite{4} in Section 4.3. We conduct further experiments to verify the performance of GDW in the mixed setting, i.e. the coexistence of label noise and class imbalance (see Appendix F for details).

4.1 Label Noise Setting

Setup. Following \cite{23}, we study two settings of label noise: a) Uniform noise: every instance’s label uniformly flips to other class labels with probability $p$; b) Flip noise: each class randomly flips to another class with probability $p$. Note that the probability $p$ represents the noise ratio. We randomly select 100 clean images per class from CIFAR10 \cite{47} as the meta set (1000 images in total). Similarly, we select a total of 1000 images from CIFAR100 as its meta set. We use ResNet-32 \cite{48} as the classifier model.

Comparison methods. We mainly compare GDW with meta-learning methods: 1) L2RW \cite{22}, which assigns weights to instances based on gradient directions; 2) INSW \cite{24}, which derives instance weights adaptively from the meta set; 3) MWNet \cite{23}; 4) Soft-label \cite{42}, which learns a label-smoothing parameter for every instance; 5) Gen-label \cite{40}, which generates a meta-soft-label for every instance. We also compare GDW with some traditional methods: 6) BaseModel, which trains ResNet-32 on the noisy training set; 7) Fine-tuning, which uses the meta set to fine-tune the trained model in BaseModel; 8) Co-teaching \cite{20}; 9) GLC \cite{19}.

Note that $g_i = \frac{1}{m} \sum_{i=1}^{m} \nabla \omega_i l_{meta}(x_i^c, y_i^c; \theta_r(\phi_{r-1}))$, and we have

$$\omega'_i = \omega_i - \text{clip}(\eta_\omega \frac{g_i}{\|g\|_1}, -c, c),$$

where $\|g\|_1$ denotes the $\ell^1$ norm of class-level gradients within a minibatch and $c = 0.2$ represents the clip parameter. Then we impose the zero-mean constraint proposed in Eq. \eqref{eq:zero_mean} on $\omega'_i$, which is later used in \textbf{Step 3} to update $\theta_{r-1}$. Note that the two-stage weight generation scheme does not introduce any extra computational cost compared to MWNet because this generation process only utilizes the intermediate gradients during the back-propagation. In \textbf{Step 3}, we use $\omega'_i$ to manipulate gradients and update the model parameters $\theta_{r-1}$:

$$\theta_r \leftarrow \theta_{r-1} - \eta_{\theta} \frac{1}{n} \sum_{i=1}^{n} f(\omega'_i (\nabla \theta l_{train}(x_i, y_i; \theta_{r-1}))).$$

The only difference between \textbf{Step 1} and \textbf{Step 3} is that we use $\omega'_i$ instead of the cloned output of the weighting network $\omega_i$ to optimize $\theta$. Since we only introduce $\phi$ as extra learnable parameters, GDW can scale to large datasets. We summarize GDW in Algorithm \ref{alg:gdw} Moreover, we visualize the two-stage weight generation process in Figure \ref{fig:two-stage} for better demonstration.
Training. Most of our training settings follow [23] and we use the cosine learning rate decay schedule [49] for a total of 80 epochs for all methods. See Appendix C for details.

Analysis. For all experiments, we report the mean and standard deviation over 5 runs in Table 2 and Table 3 where the best results are in bold and the second-best results are marked by underlines. First, we can observe that GDW outperforms nearly all the competing methods in all noise settings except for the 40% flip noise setting. Under this setting, GLC estimates the label corruption matrix well and thus performs the best, whereas the flip noise assumption scarcely holds in real-world scenarios. Note that GLC also performs much better than MWNet under the 40% flip noise setting as reported in [23]. Besides, under all noise settings, GDW has a consistent performance gain compared with MWNet, which aligns with our motivation in Figure 1. Furthermore, as the ratio increases from 40% to 60% in the uniform noise setting, the gap between GDW and MWNet increases from 1.68% to 2.97% in CIFAR10 and 1.45% to 3.12% in CIFAR100. Even under 60% uniform noise, GDW still has low test errors in both datasets and achieves more than 3% gain in CIFAR10 and 6% gain in CIFAR100 compared with the second-best method. Last but not least, GDW outperforms Soft-label and Gen-label in all settings. One possible reason is that manipulating gradient flows is a more direct way to capture class-level information than learning labels.

In Figure 3, we show the distribution of class-level target weight (ω_t) on clean and noisy instances in one epoch under the CIFAR10 40% uniform noise setting. We observe that ω_t of most clean instances are larger than that of most noisy instances, which indicates that ω_t can distinguish between clean instances and noisy instances. This is consistent with Eq. 3 that ω_t serves as the instance weight.

To better understand the changing trend of non-target class-level weights, we visualize the ratio of increased weights in one epoch in Figure 5 under the CIFAR10 40% uniform noise setting. Specifically, there are three categories: non-target weights on clean instances (ω_n^c), true target weights on noisy instances (ω^n_t) and non-target (excluding true targets) weights on noisy instances (ω^n_t). Formally, "target weight" means the class-level weight on the label position, "true-target weight" means the class-level weight on the true label position, which are only applicable for noisy instances, "non-target weight" means the class-level weight except the label position and the true label position. For example, as shown in Figure 1, where a cat is mislabeled as "dog", the corresponding meanings of the notations are as follows: 1) ω^n_t means ω_{dog} ("dog" is the target); 2) ω^n_t means ω_{cat} ("cat" is the true target); 3) ω_n^t means ω_{bird} ("bird" is one of the non-targets). For a correctly labeled...
We have also tried training without the zero-mean constraint. If negative, \( \omega_{tt} \) contributes to the decrease of the true target logit \( l_{tt} \) after a gradient descent step. If negative, \( \sum_k \omega_k p_k - \omega_t \) contributes to the increase of the non-target logit \( l_{nt} \). Therefore, without the zero-mean constraint, the second term in Eq. (13) may hurt the performance of the model regardless of the sign of \( \sum_k \omega_k p_k - \omega_t \). Similarly, training without the constraint results in poor performance in other settings. Hence we omit those results in the following subsections.

### 4.2 Class Imbalance Setting

**Setup and comparison methods.** The imbalance factor \( \mu \in (0, 1) \) of a dataset is defined as the number of instances in the smallest class divided by that of the largest \( \mu \). Long-Tailed CIFAR \( \mu \) are created by reducing the number of training instances per class according to an exponential function \( n = n_i \mu^{i/(C-1)} \), where \( i \) is the class index (0-indexed) and \( n_i \) is the original number of

![distribution of target weight](image)

**Figure 3:** Class-level target weight (\( \omega_t \)) distribution on CIFAR10 under 40% uniform noise. \( \omega_t \) of most clean instances are larger than that of most noisy instances, which means \( \omega_t \) can differentiate between clean and noisy instances.

![gradient flow info](image)

**Figure 4:** The change of class-level weights in an iteration for a noisy instance (cat mislabeled as "dog"). MWNet downweights all gradient flows. In contrast, GDW upweights the "not bird" gradient flow for better information use.

*cat*, the corresponding meanings are: 1) \( \omega_{nt}^c = \omega_{nt}^n \) means \( \omega_{nt} \) ("cat" is both the target and the true target); 2) \( \omega_{nt}^c \) means \( \omega_{nt} \) ("dog" and "bird" are both non-targets).

Note that in Figure 5, \( \omega_{nt}^c \) represents the importance of the "not cat" gradient flow and \( \omega_{nt}^n \) represents the importance of the "not bird" gradient flow. If the cat image in Figure 5 is correctly labeled as "cat", then the two non-target weights \( \omega_{nt}^c \) and \( \omega_{nt}^n \) are used to represent the importance of the "not dog" and the "not bird" gradient flows, respectively. In one epoch, we calculate the ratios of the number of increased \( \omega_{nt}^c \), \( \omega_{nt}^n \) and \( \omega_{nt}^{nt} \) to the number of all corresponding weights. \( \omega_{nt}^c \) and \( \omega_{nt}^n \) are expected to increase since their gradient flows contain valuable information, whereas \( \omega_{nt}^{nt} \) is expected to decrease because the "not cat" gradient flow contains harmful information. Figure 5 aligns perfectly with our expectation. Note that the lines of \( \omega_{nt}^c \) and \( \omega_{nt}^n \) nearly coincide with each other and fluctuate around 65%. This means non-target weights on clean instances and noisy instances share the same changing pattern, i.e., around 65% of \( \omega_{nt}^c \) and \( \omega_{nt}^n \) increase. Besides, less than 20% of \( \omega_{nt}^{nt} \) increase and thus more than 80% decrease, which means the gradient flows of \( \omega_{nt}^{nt} \) contain much harmful information.

In Figure 4, we show the change of class-level weights in an iteration for a noisy instance, i.e., a cat image mislabeled as "dog". The gradient flows of "not cat" and "dog" contain harmful information and thus are downweighted by GDW. In addition, GDW upweights the valuable "not bird" gradient flow from 0.45 to 0.63. By contrast, unable to capture class-level information, MWNet downweights all gradient flows from 0.45 to 0.43, which leads to information loss on the "not bird" gradient flow.

**Training without the zero-mean constraint.** We have also tried training without the zero-mean constraint in Section 3.3 and got poor results (see Appendix B.2 for details). Denote the true target as \( tt \) and one of the non-target labels as \( nt \). Note that the gradient can be unrolled as (see Appendix B.2 for details):

\[
\begin{align*}
\mathcal{L} & = \frac{1}{N} \sum_{i=1}^{N} \left( \sum_{j=1}^{C} \omega_j \left( p_j - y_j \right) \frac{\partial l}{\partial \theta} + \sum_{k=1}^{C} \omega_k p_k - \omega_t \right) \sum_{j=1}^{C} p_j \frac{\partial l_j}{\partial \theta}, \quad (13)
\end{align*}
\]

If \( \sum_k \omega_k p_k - \omega_t \) is positive and the learning rate is small enough, \( (\sum_k \omega_k p_k - \omega_t) \frac{\partial l_{tt}}{\partial \theta} \) contributes to the decrease of the true target logit \( l_{tt} \) after a gradient descent step. If negative, \( (\sum_k \omega_k p_k - \omega_t) \frac{\partial l_{nt}}{\partial \theta} \) contributes to the increase of the non-target logit \( l_{nt} \). Therefore, without the zero-mean constraint, the second term in Eq. (13) may hurt the performance of the model regardless of the sign of \( \sum_k \omega_k p_k - \omega_t \). Similarly, training without the constraint results in poor performance in other settings. Hence we omit those results in the following subsections.
We have conducted further experiments on imbalanced settings to verify the effectiveness of GDW. As shown in Table 4, GDW performs best in nearly all settings and exceeds MWNet Analysis. We also observed that models trained in the imbalanced setting tend to worsen the model learned from the imbalanced training set. These results align with the experimental results in [24] which also deals with class imbalance.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>CIFAR10</th>
<th>CIFAR100</th>
</tr>
</thead>
<tbody>
<tr>
<td>BaseModel</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>Fine-tuning</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>Focal</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>Balanced</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>L2RW</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>INSW</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>MWNet</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
<tr>
<td>GDW</td>
<td>$\mu = 1$</td>
<td>$\mu = 0.1$</td>
</tr>
</tbody>
</table>

As shown in Table 4, GDW performs best in nearly all settings and exceeds MWNet by 8.6% when the imbalance ratio $\mu$ is 0.01 in CIFAR10. Besides, INSW achieves competitive performance at the cost of introducing a huge amount of learnable parameters (equal to the training dataset size $N$). Furthermore, we find that BaseModel achieves competitive performance, but fine-tuning on the meta set hurts the model’s performance. We have tried different learning rates from $10^{-7}$ to $10^{-1}$ for fine-tuning, but the results are similar. One explanation is that the balanced meta set worsens the model learned from the imbalanced training set. These results align with the experimental results in [24] which also deals with class imbalance.

Denote the smallest class as $C9$ and the second smallest class as $C8$ in Long-Tailed CIFAR10 with $\mu = 0.1$. Recall that $\omega_j$ denotes the $j^{th}$ class-level weight. For all $C9$ instances in an epoch, we calculate the ratio of the number of increased $\omega_8$ to the number of all $\omega_8$, and then visualize the ratio trend in Figure 6. Since $C9$ is the smallest class, instance weighting methods upweight both $\omega_8$ and $\omega_9$ on a $C9$ instance. Yet in Figure 6 less than 10% of $\omega_8$ increase and thus more than 90% decrease. This can be explained as follows. There are two kinds of information in the long-tailed dataset regarded to $C8$: "C8" and "not C8". Since $C8$ belongs to the minority class, the dataset is biased towards the "not C8" information. Because $\omega_8$ represents the importance of "not C8", a smaller $\omega_8$ weakens the "not C8" information. As a result, decreased $\omega_8$ achieves a balance between two kinds of information: "C8" and "not C8", thus better handling class imbalance at the class level. We have conducted further experiments on imbalanced settings to verify the effectiveness of GDW and see Appendix D for details.
Table 5: Test accuracy on Clothing1M.

<table>
<thead>
<tr>
<th>Method</th>
<th>BaseModel</th>
<th>Fine-tuning</th>
<th>Co-teaching</th>
<th>GLC</th>
<th>L2RW</th>
<th>INSW</th>
<th>MWNet</th>
<th>Soft-label</th>
<th>Gen-label</th>
<th>GDW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy(%)</td>
<td>65.02</td>
<td>67.68</td>
<td>68.13</td>
<td>68.60</td>
<td>68.80</td>
<td>68.25</td>
<td>68.46</td>
<td>68.69</td>
<td>67.64</td>
<td>69.39</td>
</tr>
</tbody>
</table>

4.3 Real-world Setting

Setup and training. The Clothing1M dataset contains one million images from fourteen classes collected from the web [4]. Labels are constructed from surrounding texts of images and thus contain some errors. We use the ResNet-18 model pre-trained on ImageNet [50] as the classifier. The comparison methods are the same as those in the label noise setting since the main issue of Clothing1M is label noise [4]. All methods are trained for 5 epochs via SGD with a 0.9 momentum, a $10^{-3}$ initial learning rate, a $10^{-3}$ weight decay, and a 128 batchsize. See Appendix E for details.

Analysis. As shown in Table 5, GDW achieves the best performance among all the comparison methods and outperforms MWNet by 0.93%. In contrast to unsatisfying results in previous settings, L2RW performs quite well in this setting. One possible explanation is that, compared with INSW and MWNet which update weights iteratively, L2RW obtains instance weights only based on current gradients. As a result, L2RW can more quickly adapt to the model’s state, but meanwhile suffers from unstable weights [23]. In previous settings, we train models from scratch, which need stable weights to stabilize training. Therefore, INSW and MWNet generally achieve better performance than L2RW. Whereas in this setting, we use the pre-trained ResNet-18 model which is already stable enough. Thus L2RW performs better than INSW and MWNet.

5 Conclusion

Many instance weighting methods have recently been proposed to tackle label noise and class imbalance, but they cannot capture class-level information. For better information use when handling the two issues, we propose GDW to generalize data weighting from instance level to class level by reweighting gradient flows. Besides, to efficiently obtain class-level weights, we design a two-stage weight generation scheme which is embedded in a three-step bi-level optimization framework and leverages intermediate gradients to update class-level weights via a gradient descent step. In this way, GDW achieves remarkable performance improvement in various settings.

The limitations of GDW are two-fold. Firstly, the gradient manipulation is only applicable to single-label classification tasks. When applied to multi-label tasks, the formulation of gradient manipulation need some modifications. Secondly, GDW does not outperform comparison methods by a large margin in class imbalance settings despite the potential effectiveness analyzed in Section 4.2. One possible explanation is that better information utilization may not result in performance gain which also depends on various other factors.

6 Acknowledgement

We thank Prof. Hao Zhang from Tsinghua University for helpful suggestions. This research was supported in part by the MSR-Mila collaboration funding. Besides, this research was empowered in part by the computational support provided by Compute Canada (www.computecanada.ca).

References


