Extreme Classification

Akaash Patel
Keble College
University of Oxford

A thesis submitted for the degree of
MMath in Mathematics and Statistics

Trinity Term 2022

Abstract

Real-world datasets often have a small number of *dominant* classes accounting for a large proportion of the training set, with a tail of relatively *rare* ones. Naïve training methods are prone to an unwanted bias towards dominant classes.

We consider two types of techniques to handle this. Firstly, we modify the loss used for training to obtain better results on rare classes. Re-weighting it by class is a natural solution, though not always effective for deep neural networks. We find that 'margin adjustment', which aims to shift the decision boundaries, and the second method of 'logit adjustment' give better empirical results. Re-weighting only near the end of training gives further improvement on the latter two techniques.

Along the way we provide a brief insight into alternative techniques that could better handle severe imbalance and the 'open-world' setting, where unseen classes appear in the test set, and a generalisation of our method for arbitrary, known test class distributions.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Background 1

1.3 Outline 3

2 Cost-sensitive learning **4**

2.1 Statistical grounding 4

2.2 Weighted loss functions 6

2.3 Margin adjustment 8

2.4 Effectiveness of weighting 9

2.5 Extension 10

3 Logit adjustment **12**

3.1 Statistical grounding 12

3.2 Choices of updates 14

3.3 Combination with loss weighting 15

3.4 Extension 17

4 Experimental Results **19**

4.1 Synthetic Data - Binary 19

4.2 Synthetic Data - Multiclass 23

4.3 CIFAR-10 25

5 Conclusion **31**

A Code **32**

A.1 Synthetic Data 32

A.2 CIFAR-10 43

Bibliography **46**

Chapter 1

Introduction

1.1 Motivation

In classification, often training datasets exhibit a long-tailed class distribution; a small proportion of the classes cover the majority of training data. Meanwhile, the remaining classes each have few samples, making prediction on them difficult.

Let's consider a real-world example of an image classification problem based on a dataset exhibiting this behaviour. The iNaturalist Challenge 2018 dataset [11] consists of over 450000 training images of over 8000 different species of organisms. Some species had thousands of images, while others had just one or two. This reflects the fact that some species are more abundant than others or easier to photograph (i.e. to sample, from a statistical perspective).

The trained model is typically biased towards these dominant classes, leading to poor recall on the rarer classes. The intuition behind this bias is as follows: a classifier that only predicts the most dominant classes for all training examples could achieve relatively high training accuracy. This results in a large number of false positives for such classes, so we look for methods to prevent this.

1.2 Background

Here we present some definitions and methodologies that are standard in machine and deep learning, and will be used throughout the main body of this dissertation. We define what a classifier is, formulate how we learn one using loss functions and give some properties of neural networks.

Definition 1 (Classifier)**.** Let $\mathcal{X}$ be an input space and $\mathcal{Y}$ a finite output space. A *classifier* is a function $h : \mathcal{X} \to \mathcal{Y}$.

In a general setting $\mathcal{X}$ can be any kind of input space; for example $\mathbb{R}^p$ for some p or the space of images (as in iNaturalist). However it is simpler to consider $\mathcal{X} = \mathbb{R}^p$ and $\mathcal{Y} = \{1, ..., K\}$ for some $K \geq 3$, or $\mathcal{Y} = \{-1, +1\}$ for binary classification. Other types of input can be converted into a vector, say by 'flattening' a matrix of pixel values. Classifiers can be expressed in the form:

$$h(x) = \arg\max_{i \in \mathcal{Y}} f_i(x)$$

where $f_i(x) : \mathcal{X} \to \mathbb{R}$ are the *discriminant functions*. $f_i(x)$ quantifies the 'confidence' that an input x belongs to class i, so we pick the class with the highest associated confidence. For a given h, there is a non-unique $f(x) := (f_i(x))_{i \in \mathcal{Y}}$. Since classifiers as defined above are learned through their discriminant functions, in this dissertation, we mean f when referring to the classifier. The set of points where two discriminant functions are equal is called the *decision boundary* for the corresponding classes. When $|\mathcal{Y}| = 2$, we can combine the discriminant functions, and consider $f(x) = f_{+1}(x) - f_{-1}(x)$, to attain:

$$h(x) = \begin{cases} +1 & f(x) \geq 0 \\ -1 & f(x) < 0 \end{cases}.$$

Suppose we have drawn a training dataset (of size n), $\mathcal{D} = \{(x_i, y_i) \in \mathcal{X} \times \mathcal{Y} : i \in \{1, ..., n\}\}$, from a fixed joint distribution, with samples drawn independently. We aim to find a classifier f that 'best' predicts, given $x \in \mathcal{X}$, a class $h(x) \in \mathcal{Y}$. Hence we want to assess how f performs on a single instance (x, y).

Definition 2 (Loss function)**.** A *loss function*, or the loss, is a function $l : \mathcal{Y} \times \mathbb{R}^K \to \mathbb{R}_{\geq 0}$, that puts a cost on a classifier's output, given the true class.

We consider $l(y, f(x))$, which is greater the more confidently f predicts *false* classes (ones that are not the *true* class y). Since our samples (X, Y) are drawn from a random distribution, we consider the following:

Definition 3 (Risk and Bayes classifier)**.** Given a loss function l, the *risk*, $R_l(f)$, of a classifier f is the expected value of the loss, $R_l(f) = \mathbb{E}(l(Y, f(X))$ over the joint distribution of (X, Y). The *Bayes classifier*, f^*, is one that minimises this risk.

Therefore we would like to learn the Bayes classifier; however we cannot do this exactly without knowing the true distribution of (X, Y). In practice we minimise an unbiased estimate for it, the *empirical risk*:

$$\hat{R}_l(f) = \frac{1}{n} \sum_{i=1}^{n} l(y_i, f(x_i)). \tag{1}$$

This is known as *empirical risk minimisation* (ERM). In selection of a machine learning algorithm, we implicitly search for a minimiser over a class of functions f.

This class could be simple, such as affine classifiers, where $f(x) = Wx + b$, for some weight matrix $W \in \mathbb{R}^{K \times p}$, and bias $b \in \mathbb{R}^K$. Or it could be more complex, such as a deep neural network with a higher dimensional parameter space.

In any case, the class of functions is parameterised by $\theta \in \Theta$, where Θ is the parameter space. The class of affine classifiers has $\theta = (W, b) \in \Theta = \mathbb{R}^{K \times p} \times \mathbb{R}^K$ and $f^\theta = f^{(W,b)} : x \mapsto Wx + b$. For neural networks, evaluations of the discriminant functions are also called the *logits* (outputs) — before softmax is implemented, if at all.

Since it is enough to learn the best θ, we formulate ERM, without any regularisation, as:

$$\min_{\theta \in \Theta} \frac{1}{n} \left(\sum_{i=1}^{n} l(y_i, f^\theta(x_i)) \right).$$

The most common choice of loss function for classification neural networks is the *cross-entropy loss*:

$$l(y, f(x)) = -\log \left(\frac{\exp(f_y(x))}{\sum_{j=1}^{K} \exp(f_j(x))} \right) = \log \left(1 + \sum_{j \neq y} \exp\left(f_j(x) - f_y(x)\right) \right). \quad (2)$$

We take l to be the cross-entropy loss unless stated otherwise.

1.3 Outline

As of this dissertation, there has been much progress in addressing class imbalance and long-tail learning. Zhang et al. 2021 [14] gives a survey of this, and categorises existing methods as 'class-rebalancing', 'information augmentation' or 'module improvement'. The most intuitive of these is class-rebalancing, focussing on balancing the contribution of each class to our learned classifier.

Class-rebalancing methods are further sub-categorised as 'cost-sensitive learning', 'logit adjustment' and 'resampling'. We investigate the statistical grounding of the first two, discuss examples of them, provide some empirical results reflecting the intuition and theory behind them, and demonstrate their effectiveness.

Chapter 2

Cost-sensitive learning

To account for varying frequencies of each class $y \in \mathcal{Y}$ in our training set, we can adjust our loss function l. We upweight the contribution to the overall empirical risk from training examples belonging to rare classes relative to ones from dominant classes, in an effort to improve accuracy on them.

The methods outlined in this chapter consider the cross-entropy loss, and modifications of it in two senses: weighting — rescaling $l(y, f(x))$ by factors α_y depending only on the true class, and margin adjustment — shifting the decision boundaries closer towards dominant classes.

2.1 Statistical grounding

Recall that the risk is the expected loss, where our example (X, Y) is drawn from an unknown distribution. Classically we assume that our training and test sets are both drawn from this same distribution. Given a long-tailed training set, we might conclude this assumption to be incorrect — in particular the marginal distributions of the classes do not coincide. However we still assume that the conditional distribution or density, $\mathbb{P}(X|Y)$, are the same for both dataset sources. We also assume that our test set is *balanced*, meaning that the class $Y \sim \text{Unif}(\mathcal{Y})$ marginally.

Let $\mathbb{P}_{train}$ and $\mathbb{P}_{test}$ be these true distributions from which our datasets are drawn. Our 'test risk' now becomes: $R_{l,test}(f) = \mathbb{E}_{test}(l(Y, f(X))$, with expectation taken under $\mathbb{P}_{test}$ (similarly we denote an expectation under $\mathbb{P}_{train}$ with $\mathbb{E}_{train}$). For a single observation (X, Y) we have that:

$$\mathbb{P}_{train}(X, Y) = \mathbb{P}_{train}(Y)\mathbb{P}(X|Y) \tag{3}$$

and

$$\mathbb{P}_{test}(X, Y) = \mathbb{P}_{test}(Y)\mathbb{P}(X|Y) = \frac{1}{K}\mathbb{P}(X|Y). \tag{4}$$

We can write $R_{l,test}(f)$ as (weighting samples by the likelihood ratio as in importance sampling):

$$\begin{aligned}\mathbb{E}_{test}(l(Y, f(X)) &= \mathbb{E}_{train}\left(l(Y, f(X))\frac{\mathbb{P}_{test}(X, Y)}{\mathbb{P}_{train}(X, Y)}\right) \\ &= \mathbb{E}_{train}\left(l(Y, f(X))\frac{1/K}{\mathbb{P}_{train}(Y)}\right) \\ &\propto \mathbb{E}_{train}\left(\frac{l(Y, f(X))}{\mathbb{P}_{train}(Y)}\right).\end{aligned}$$

We therefore, to minimise the test risk, perform ERM over the training dataset, instead with a weighted loss, $\tilde{l}$, where

$$\tilde{l}(y, f(x)) = \alpha_y l(y, f(x)) \tag{5}$$

and α_y is an estimator of $\mathbb{P}_{train}(y)^{-1}$ up to proportionality. Equivalently, we can minimise a *weighted empirical risk* with loss l:

$$\hat{R}_{\tilde{l}}(f) = \frac{1}{n}\sum_{i=1}^{n} \alpha_{y_i} l(y_i, f(x_i)) \tag{6}$$

Meanwhile margin adjustment seeks to change the minimum distance of points of a given class to their corresponding decision boundaries. We counteract the bias against rare classes by increasing this distance for them, which has the effect of expanding the region that the classifier assigns to each of those classes.

First let's consider a binary classification problem, which is imbalanced, with $+1$ as the dominant class, in which we learn an affine classifier. Therefore the decision boundary is given by $f_{+1}(x) - f_{-1}(x) = f(x) = 0$, which is affine in x. To shift it towards the dominant $+1$ class, thus expanding the region of classification for -1, we can set the boundary to instead be $f = \Delta$ from some $\Delta > 0$. Letting $\tilde{f} = f - \Delta$, our decision boundary becomes $\tilde{f} = 0$, so learning $\tilde{f}$ as usual gives us the shifted decision boundary. The cross-entropy loss for $\tilde{f}$ is then:

$$l(y, \tilde{f}(x)) = \log(1 + \exp(-y\tilde{f}(x))) = \log(1 + \exp(-y(f(x) + \Delta))) =: l_\Delta(y, f(x)).$$

As expected, increasing Δ increases $l_\Delta(-1, f)$ relative to $l(-1, f)$, and there is a relative decrease when the true label is $+1$ instead. Hence, for large Δ, we have the desired upweighting of the loss for the rarer class. In a multiclass setting, we now consider a separate margin for each pair of classes, and analogously our new loss is:

$$l_\Delta(y, f(x)) = \log\left(1 + \sum_{j \neq y} \exp\left(f_j(x) - f_y(x) + \Delta_{yj}\right)\right) \tag{7}$$

where Δ_{jk} is a 'margin shift' from class j towards class k, for each $j \neq k \in \mathcal{Y}$. Considering just a single observation (x, y), a large Δ_{yj} would 'force' $f_j - f_y$ down, thus pushing the decision boundary away from class y towards j. For rare y and/or frequent j we would like a large Δ_{yj}.

2.2 Weighted loss functions

Losses of the form in (5) can be parameterised by choice of $\alpha = (\alpha_i)_{i \in \mathcal{Y}}$. A uniform training class distribution gives $\alpha_i = K \propto 1$ for each $i \in \mathcal{Y}$, meaning our modified loss is proportional to the unweighted one, and we learn the same classifier.

An intuitive choice is to take $\alpha_y = \pi_y^{-1}$, where π_y is the empirical estimator for $\mathbb{P}_{train}(y)$. This results in the following *class-weighted* cross-entropy loss:

$$\tilde{l}(y, f(x)) = \frac{l(y, f(x))}{\pi_y} \tag{8}$$

For an alternative weighting, we introduce the notion of the *effective number* of samples from a class, as in Cui et al. 2019 [4]. We use these to find an alternative approximation for the training class distribution, by reconsidering the data sampling process.

First let's consider S_y, the set of samples from a class y. Then take a pair of these samples. Rather than think of them as completely separate instances, we view them as either, effectively, 'distinct' or 'duplicates' of one another. We can also think of this as partitioning S_y, where two samples are in the same part if and only if they are duplicates of eachother.

Now suppose we draw a new sample belonging to class y. Then it is either effectively new, or a duplicate of another one in S_y. If we add it to S_y, in the first case, the effective number of samples, equivalently the number of parts, in S_y increases by one, and in the second, it remains unchanged.

For some motivation to do this, consider the iNaturalist dataset. We could have two or more distinct images, that are of the same exact animal. In a sense, we have duplicates of this sample, so we could consider these images effectively the same.

While the true size of S_y is simply n_y (the number of examples in our dataset in class y), we now detail how to find the expected number of effective samples. We also assume that for each class, there is the same finite number, N, of possible parts.

Proposition 4. *Let $\beta = (N-1)/N$. Then the expected number of effective samples belonging to a class y, with class frequency n_y, is:*

$$E_{n_y} = \frac{1-\beta^{n_y}}{1-\beta}$$

Proof. Our proof follows the same line of reasoning as in [4]. We proceed by induction on n_y. When $n_y = 1$, there can only be one effective sample, so $E_1 = 1$ and the claim holds.

Now suppose the claim holds for some $k-1$, so that $E_{k-1} = (1-\beta^{k-1})/(1-\beta)$. To consider E_k, suppose we have $k-1$ true samples in S_y, and we draw a kth. Now condition on the effective number of samples in the first $k-1$. Given that this is e_{k-1}, the new sample belongs to a pre-seen part with probability $p = e_{k-1}/N$. Otherwise it forms an additional part.

$$\mathbb{E}(e_k|e_{k-1}) = pe_{k-1} + (1-p)(e_{k-1}+1) = 1 + \frac{N-1}{N}e_{k-1}$$

Taking total expectations and using our inductive assumption, we get:

$$E_k = \mathbb{E}(\mathbb{E}(e_k|e_{k-1})) = 1 + \beta E_{k-1} = 1 + \beta\left(\frac{1-\beta^{k-1}}{1-\beta}\right) = \frac{1-\beta^k}{1-\beta}$$

We conclude that the claim holds for all n_y. □

Since a given class y has n_y samples in the training set, we estimate its number of effective samples with E_{n_y}, and weight our losses accordingly: $\alpha_y = E_{n_y}^{-1}$ and

$$l_\beta(y, f(x)) = \frac{1-\beta}{1-\beta^{n_y}} l(y, f(x)). \tag{9}$$

We can treat β as a hyperparameter to be tuned; for large datasets, N is typically large, so we need β to be very close to 1. Using $\beta = 0$ returns the unweighted loss (2), while letting $\beta \to 1$, yields $\alpha_y \to n_y^{-1}$ by L'Hôpital's rule, which is the class-weighted loss (8) up to proportionality. Hence $\beta = 1$ in reference to a loss function means the class-weighted loss from now on. We can therefore view (9) as an interpolation between the unweighted and class-weighted losses, or E_{n_y} as a smoothing of the empirical class frequency n_y.

Figure 1 shows this interpolation between $\beta = 0$ and $\beta = 1$, where decreasing β smooths the observed class probabilities towards uniform ones. Later we see this interpolation as a translation in the learned decision boundary of the classifier under ERM with l_β.

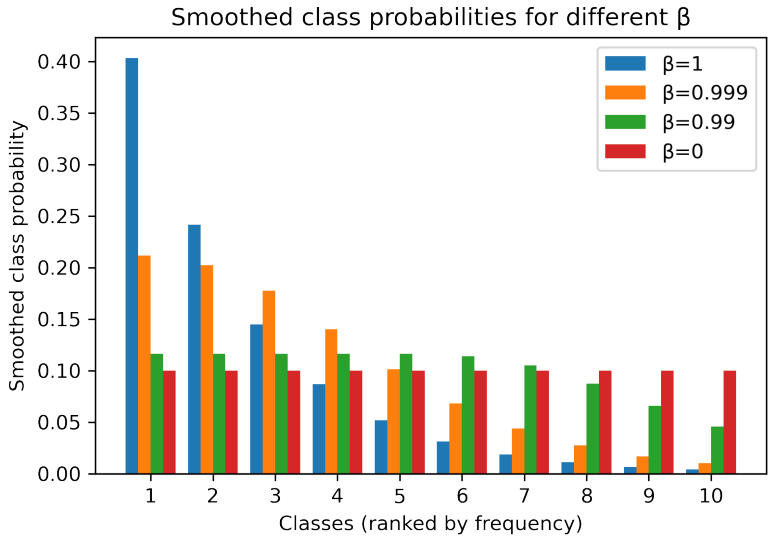

Figure 1: Normalised E_{n_y} for different β for a long-tailed version of CIFAR-10 [8]
.

2.3 Margin adjustment

Now we consider different choices for Δ in (7) that encourage classification on the rare classes. One such choice, in Cao et al. 2019 [2], is:

$$\Delta_{yj} = C\pi_y^{-1/4} \tag{10}$$

where C is a tuneable hyperparameter. This gives rise to the 'LDAM' ('label-distribution-aware margin') loss, as named in the corresponding paper. Here the margin shift from the true 'positive' class y is uniform towards each of the other 'negative' classes j. We could instead make a choice of shift independent of the true class. Each time j is the negative class, we now have a fixed margin shift towards it:

$$\Delta_{yj} = F(j), \tag{11}$$

where $F(j) \leq 0$ for all $j \in \mathcal{Y}$ and $F(j)$ increases with π_j. To see why, we note that in the empirical risk with this Δ, the rare classes' logits appear as the negatives f_j for a large number of observations. Without margin adjustment, they are learned to be smaller, which has the effect of lowering the loss for these observations; but a large, negative adjustment for that rare class can allow its logits to be larger instead. Meanwhile, for dominant classes the adjustment is smaller in magnitude, so when they appear as the false class, their logits can only make smaller gains.

The above choices each consider either only the true class y, or only the false classes j, where $j \neq y$; in (10) the true class 'pushes back' against all other classes

equally, regardless of how dominant the others are. Similarly in (11), the negatives enforce the same adjustment, no matter the true class.

One that simultaneously considers y and j, that gives a version of the 'logit-adjusted' loss [8], has:

$$\Delta_{yj} = \log\left(\frac{\pi_j}{\pi_y}\right), \tag{12}$$

which can adjust rare-dominant boundaries more than dominant-dominant or rare-rare ones. We will see that this is equivalent to a logit adjustment technique in the next chapter, and will further discuss its benefits then.

2.4 Effectiveness of weighting

It is well-known that deep neural networks are often 'overparameterised', sometimes having more learnable parameters than training examples to learn on. This gives rise to a very broad class of classifiers f, capable of overfitting to the training data, often achieving zero training error. Empirical risk is reduced further with training beyond this point, leading to more 'confident', correct predictions on the training data.

Empirically it has been shown, in Byrd and Lipton 2019 [1], that weighting the loss does not always have an effect on the learned decision boundaries, particularly for separable data and overparameterised models.

The following helps us to understand why this might be the case:

Proposition 5. *Consider the empirical risk $\hat{R}_l$ in (1), with loss l, and a weighted empirical risk, $\hat{R}_{\tilde{l}}$ as in (6). Then $\forall\varepsilon > 0\ \exists\delta > 0$ such that if $\hat{R}_l(f) < \delta$, then $\hat{R}_{\tilde{l}}(f) < \varepsilon$.*

Proof. Given an ε, we simply let

$$\delta = \varepsilon\left(\max_{j\in\mathcal{Y}} \alpha_j\right)^{-1}.$$

Then:

$$\begin{aligned}
\hat{R}_l(f) &= \frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i)) < \varepsilon\left(\max_{j\in\mathcal{Y}} \alpha_j\right)^{-1} = \delta \implies \\
\hat{R}_{\tilde{l}}(f) &= \frac{1}{n}\sum_{i=1}^{n} \alpha_{y_i} l(y_i, f(x_i)) \\
&\leq \left(\max_{j\in\mathcal{Y}} \alpha_j\right)\frac{1}{n}\sum_{i=1}^{n} l(y_i, f(x_i)) < \varepsilon
\end{aligned}$$

□

In words, a classifier learned under any weighted ERM can be learned under unweighted ERM, if the latter can achieve arbitrarily small training risk. With enough time, it is possible that the same solution is learned in both cases.

2.5 Extension

There are many ways to estimate the class probabilities, giving rise to alternative smoothed distributions, which can be used to weight the loss (as we have seen with the effective number of samples) or even in place of π_y for margin-adjusted losses.

In extremely long-tailed cases, many classes appear only once in the training set. Even worse, it is possible for a test observation's true class to be unseen in the training data — so our learned model cannot classify it correctly without just labelling it as 'unknown'. The empirical class probability estimator is poor for extremely rare classes, and for an unseen class it would be 0. Good-Turing estimation instead considers estimating the probability that a new sample belongs to a class appearing r times in the training set as follows.

Let N_j be the number of classes with frequency j in the training set. We consider the probability of drawing a sample belonging to a class seen r times other than itself in the training set. There are $(r+1)N_{r+1}$ such samples, and so we approximate the probability as $(r+1)N_{r+1}/n$. Since there are N_r classes appearing r times, the probability of a new sample belonging to a single class with frequency $r > 0$ has Good-Turing estimate:

$$\frac{(r+1)N_{r+1}}{N_r n},$$

as opposed to r/n by the empirical estimate. In practice, smoothed values for N_r are used instead, as empirically many will be zero. For $r = 0$, the probability of the new sample belonging to an unseen class, the Good-Turing estimate is simply N_1/n.

Alternatively a Bayesian approach can be taken. The Dirichlet distribution is a conjugate prior for our categorical likelihood on the classes, meaning that we obtain a tractable Dirichlet posterior. This prior, on K-class distributions, is parameterised as follows:

$$p(\pi_1, ..., \pi_K | \alpha_1, ..., \alpha_K) = \begin{cases} \Gamma\left(\sum_{k=1}^{K} \alpha_k\right) \prod_{k=1}^{K} \frac{\pi_k^{\alpha_k - 1}}{\Gamma(\alpha_k)} & \pi_1, ..., \pi_K \in (0,1), \sum_{k=1}^{K} \pi_k = 1 \\ 0 & \text{otherwise} \end{cases}.$$

A maximum a posteriori (MAP) estimate for the class distribution can be used, which acts as a version of the empirical estimate smoothed towards the chosen prior.

A natural choice of hyperparameters for the prior could be taking $\alpha_1 = ... = \alpha_K = 1$, corresponding to a uniform prior over all possible K-class categorical distributions. Domain knowledge can be incorporated into the choice of prior, for example knowledge that some classes are more frequent in the true distribution, even if they may be rarer in the training dataset.

A final modification is the use of a hyperparameter τ for 'temperature scaling'. Replacing our probability estimates π_j with π_j^τ in our modified loss functions, and tuning τ, can allow us to obtain models that perform better empirically. $\tau = 0$ corresponds to a uniform class distribution estimate, while $\tau = 1$ would correspond to the usual class-weighting. Again varying τ between 0 and 1 corresponds to an interpolation, similar to Figure 1, but increasing it above 1 has the effect of punishing dominant classes even more.

Chapter 3

Logit adjustment

We now look at methods where we can train under the original cross-entropy loss (2), then, for each input we would like to predict on, modify the resulting logits to account for the class imbalance before making our final prediction.

In this chapter, we will consider the theoretical properties of some possible modifications we can make, relate these to the margin adjustment techniques proposed in the previous chapter and discuss benefits of making adjustments to a model post-training.

3.1 Statistical grounding

Recall in (2), the input for the negative of the logarithm below. It is common to interpret these as the class distribution given an input x:

$$\mathbb{P}(Y = y|X = x) = \frac{\exp(f_y(x))}{\sum_{j=1}^{K} \exp(f_j(x))} \tag{13}$$

We do this because ERM under (2) becomes equivalent to maximising the likelihood under the model above:

$$\begin{aligned}
\arg\min_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^{n} l(y_i, f^\theta(x_i)) &= \arg\min_{\theta\in\Theta} \frac{1}{n}\sum_{i=1}^{n} -\log\left(\frac{\exp(f^\theta_{y_i}(x_i))}{\sum_{j=1}^{K}\exp(f^\theta_j(x_i))}\right) \\
&= \arg\max_{\theta\in\Theta} \sum_{i=1}^{n} \log\left(\frac{\exp(f^\theta_{y_i}(x_i))}{\sum_{j=1}^{K}\exp(f^\theta_j(x_i))}\right) \\
&= \arg\max_{\theta\in\Theta} \sum_{i=1}^{n} \log \mathbb{P}_\theta(Y = y_i|X = x_i).
\end{aligned}$$

The final term, the sum of log-likelihoods of the training examples, is just the overall log-likelihood of the training set (by independence of the examples).

The 0-1 loss for classification given by

$$l_{0-1}(y, f(x)) = \begin{cases} 1 & y \neq \arg\max_{j \in \mathcal{Y}} f_j(x) \\ 0 & \text{otherwise} \end{cases},$$

has Bayes classifier that predicts a class as follows [3]:

$$h^*(x) = \arg\max_{y \in \mathcal{Y}} \mathbb{P}(Y = y | X = x).$$

Rather than training to minimise the total error rate as usual, we instead choose to minimise (empirically) the following *balanced error rate* (BER):

$$\text{BER}(f) = \frac{1}{K} \sum_{y=1}^{K} \mathbb{P}(y \neq \arg\max_{j \in \mathcal{Y}} f_j(X) | Y = y)$$

which is the mean misclassification rate for each class. This is so that equal weight is placed on the performance of the model on each class, regardless of its dominance in the training set. While the total error rate is the risk under 0-1 loss and training distribution, the BER is the same but under the balanced test distribution as defined in (4). From the above,

$$\begin{aligned} \text{BER}(f) &= \sum_{y=1}^{K} \mathbb{P}_{test}(Y = y) \mathbb{P}(y \neq \arg\max_{j \in \mathcal{Y}} f_j(X) | Y = y) \\ &= \mathbb{P}_{test}(Y \neq \arg\max_{j \in \mathcal{Y}} f_j(X)) \\ &= \mathbb{E}_{test}(l_{0-1}(Y, f(X))). \end{aligned}$$

Hence the Bayes classifier minimising balanced error rate is (using (3) and (4)):

$$\begin{aligned} \arg\max_{y \in \mathcal{Y}} \mathbb{P}_{test}(Y = y | X = x) &= \arg\max_{y \in \mathcal{Y}} \mathbb{P}_{test}(x, Y = y) \\ &= \arg\max_{y \in \mathcal{Y}} \mathbb{P}(x | Y = y) \\ &= \arg\max_{y \in \mathcal{Y}} \frac{\mathbb{P}_{train}(x, Y = y)}{\mathbb{P}_{train}(Y = y)} \\ &= \arg\max_{y \in \mathcal{Y}} \frac{\mathbb{P}_{train}(Y = y | X = x)}{\mathbb{P}_{train}(Y = y)}. \end{aligned}$$

Since we have learned f using the training distribution, applying (13) and taking logarithms gives us that the Bayes classifier predicts according to:

$$\arg\max_{y \in \mathcal{Y}} \log \left(\frac{\mathbb{P}_{train}(Y = y | X = x)}{\mathbb{P}_{train}(Y = y)} \right) = \arg\max_{y \in \mathcal{Y}} f_y(x) - \log(\mathbb{P}_{train}(Y = y)).$$

We now see that additively updating the logits just before making the prediction by $-\log \pi_y$ is theoretically motivated, as our resulting classifier is consistent with the Bayes classifier for the balanced error. We call a classifier satisfying this property *BE* (balanced error) *consistent* from now on.

3.2 Choices of updates

The additive update above has the theoretical backing of BE consistency, but there are other ways we can penalise dominant logits. Estimating $\mathbb{P}_{train}(Y = y) = \pi_y$, our additive update is: $f_y(x) - \log \pi_y$. Explicitly writing out the loss associated with the margin adjustment in (12):

$$l_\Delta(y, f(x)) = \log \left(1 + \sum_{j \neq y} \exp\left((f_j(x) + \log \pi_j) - (f_y(x) + \log \pi_y) \right) \right), \tag{14}$$

we see that training under this loss is equivalent to learning $g_j = f_j + \log \pi_j$ under the cross-entropy loss, then applying this additive update to get the logits f_j.

A natural alternative would be to apply a multiplicative update, which upscales logits for rare classes more than for frequent classes by the inverse class probability: $\frac{f_y(x)}{\pi_y}$. This is equivalent to another modified loss proposed in Ye et al. 2020 [13] (which generalises this by including temperature scaling of the class probabilities).

Often neural network outputs are of the form $f_y(x) = w_{\theta,y}^T \varphi_\theta(x)$, linear in a parameterised feature map $\varphi_\theta(x)$, whose parameters are learned, as well as the final layer's weights $\{w_{\theta,y} : y \in \mathcal{Y}\}$. A bias term in the final layer can be incorporated by including a constant feature in $\varphi_\theta(x)$, so that the corresponding term in the learned $w_{\theta,y}$ is this bias. Intuitively we expect the classes y with a large learned $||w_{\theta,y}||$ to be dominant, as they can give rise to logits with a larger absolute value. We can rescale to account for this with an alternative multiplicative update [6]: $\frac{f_y(x)}{||w_{\theta,y}||}$. Writing the logits as:

$$f_y(x) = w_{\theta,y}^T \varphi_\theta(x) = ||w_{\theta,y}|| ||\varphi_\theta(x)|| \cos(\psi_{\theta,y}(x))$$

we can view this as an 'angular' classification:

$$\begin{aligned} \arg\max_{y \in \mathcal{Y}} \frac{f_y(x)}{||w_{\theta,y}||} &= \arg\max_{y \in \mathcal{Y}} \frac{w_{\theta,y}^T \varphi_\theta(x)}{||w_{\theta,y}|| ||\varphi_\theta(x)||} \\ &= \arg\max_{y \in \mathcal{Y}} \cos(\psi_{\theta,y}(x)) \end{aligned}$$

where each weight vector $w_{\theta,y}$ in the final layer represents a direction in the learned feature space, associated with a class y. This means we choose the class whose direction aligns most with the transformed input $\varphi_\theta(x)$. Viewing $\psi_{\theta,y}(x)$ as the angle between the two directions, its cosine measures their similarity. Since these cosines are our new discriminant functions, the resulting classifier can be trained under the corresponding cross-entropy loss, with a margin shift:

$$\log\left(1+\sum_{j\neq y}\exp\left(\cos(\psi_{\theta,j}(x))-\cos(\psi_{\theta,y}(x))+\Delta_{yj}\right)\right)$$

Analogous to a margin shift of the boundary towards a dominant class in the previous section, this enforces a rotation of the boundary in the feature space. A fixed $\Delta = m$ leads to the 'Large Margin Cosine Loss' in Wang et al. 2018 [12], which notes that this encourages further separation in the learned features' 'cosine space', whereas the previous additive updates on the logits only changes the learned biases.

3.3 Combination with loss weighting

We have so far considered separately the ideas of weighting the loss function before training, and updating the outputs after. This naturally gives rise to the following question: can we attain even better results by doing them both? That is to say, if we update the logits for a neural network trained under a weighted loss function, can we do even better?

Considering the additive update, our final outputs f are that from training under the following weighted and margin-adjusted loss:

$$l_{\alpha,\delta}(y, f(x)) = \alpha_y \log\left(1+\sum_{j\neq y}\exp\left((f_j(x)+\log\delta_j)-(f_y(x)+\log\delta_y)\right)\right) \tag{15}$$

We now give a proof of a result which guarantees BE consistency for the optimal classifier f under this loss, for certain weights α and updates, parameterised by δ.

Theorem 6. *For any choice of $\delta \in \mathbb{R}^K_+$, taking $\alpha_y = \delta_y/\pi_y$ gives a loss as in (15) whose Bayes classifier is BE consistent.*

Proof. A proof of this is given in Appendix A of Menon et al. 2019, Theorem 1 [8]. Our proof below will follow a similar path, but concluding by using what we have proven in Section 3.1.

ERM under this loss on our training set is to empirically minimise the following:

$$\begin{aligned}
\mathbb{E}_{train}(l_{\alpha,\delta}(Y, f(X))) &= \sum_{y=1}^{K} \mathbb{P}_{train}(Y = y)\mathbb{E}_{X|Y=y}(l_{\alpha,\delta}(y, f(X))) \\
&= \sum_{y=1}^{K} \pi_y \mathbb{E}_{X|Y=y}(l_{\alpha,\delta}(y, f(X))) \\
&= \sum_{y=1}^{K} \pi_y \alpha_y \mathbb{E}_{X|Y=y}(l_{1,\delta}(y, f(X))) \\
&= \sum_{y=1}^{K} \delta_y \mathbb{E}_{X|Y=y}(l_{1,\delta}(y, f(X))) \\
&\propto \sum_{y=1}^{K} \bar{\delta}_y \mathbb{E}_{X|Y=y}(l_{1,\delta}(y, f(X)))
\end{aligned}$$

where $l_{1,\delta}(y, f(x))$ is just the unweighted (or all weights equal to 1) logit-adjusted loss (14) with π replaced by δ, and $\bar{\delta}_y = \delta_y / \sum_j \delta_j$.

Note that re-scaling δ does not change $l_{1,\delta}(y, f(x))$, as it simply adjusts each logit by the same constant, so the difference between the logits remains the same. Therefore $l_{1,\delta}(y, f(x)) = l_{1,\bar{\delta}}(y, f(x))$ and so the final line is equal to:

$$\sum_{y=1}^{K} \bar{\delta}_y \mathbb{E}_{X|Y=y}(l_{1,\bar{\delta}}(y, f(X))) = \mathbb{E}_{\mathbb{Q}}(l_{1,\bar{\delta}}(Y, f(X)))$$

where $\mathbb{Q}(x, y) = \bar{\delta}_y \mathbb{P}(x|Y = y)$. We know that $l_{1,\bar{\delta}}$ is the unweighted logit-adjusted loss applied to a training set drawn from distribution $\mathbb{Q}$.

We have seen that the additive update of $-\log \pi_j$ for each f_j is equivalent to learning under the loss in (14), which is the same as $l_{1,\pi}$ in (15). This aims to minimise $\mathbb{E}_{train}(l_{1,\pi}(Y, f(X)))$, where π is the training class distribution. By Section 3.1 we have seen this gives a BE consistent classifier.

As shown above, we are now minimizing $\mathbb{E}_{\mathbb{Q}}(l_{1,\bar{\delta}}(y, f(X)))$. Since $\mathbb{P}_{train}$ and $\mathbb{Q}$ have the same conditional distribution on Y, simply apply the same reasoning with $\mathbb{P}_{train}$ and π replaced by $\mathbb{Q}$ and $\bar{\delta}$ respectively, to show that this also gives a BE consistent classifier. □

We see that taking $\delta_y = \pi_y$ gives simply the unweighted logit-adjusted loss. Meanwhile taking $\delta_y = 1$ gives the class-weighted loss, so the learned classifier under (8) is also BE consistent. Some other appropriate choices of α and δ resulting in a loss that is both weighted and margin-adjusted are $\alpha_y = \pi_y^{\tau-1}$ and $\delta_y = \pi_y^{\tau}$.

While this loss has strong theoretical foundations, as we have just shown, there may still be concern over the effect of weighting as discussed in Section 2.4. We may still learn the same classifier we would without any weighting, i.e. when $\alpha_y = 1$ for all y. This loss does not satisfy the conditions for the theorem above, so there is no guarantee of BE consistency, and we could observe worse performance.

3.4 Extension

The additive logit adjustment we have discussed, and the loss modifications, assume a test distribution uniform on the classes. Without any prior knowledge of it, this is a justifiable and interpretable assumption. However it would be advantageous to be able to incorporate knowledge or estimates of the test class distribution, should they be available.

Suppose we have two models, one trained under a modified loss and the other predicting via logit adjustment, both designed to generalise for prediction on a balanced test set. We may later realise that in fact the true test distribution is not balanced; it could be long-tailed in a similar fashion to the training set, or even inversely long-tailed — meaning frequent classes in the training set are rare in the test set and rare ones in the training set are now frequent.

One way to incorporate this into a weighted loss function (5) would be to weight by α_y which estimates $\mathbb{P}_{test}(Y = y)/\mathbb{P}_{train}(Y = y)$. For the training and test class distributions we take our estimates to be π_y, as before, and ρ_y, the class probability estimates for the test set, respectively. However our previously trained model cannot help now, because it was trained assuming a balanced test set without accounting for the now known (or estimated) test class distribution. Instead a new model would need to be trained, which computationally could be expensive.

Logit adjustment is preferred in this scenario, as our predictions can be tailored once the test distribution becomes available, without any extra training. Furthermore they can be adapted separately for different test datasets, theoretically improving performance on all of them without making any assumptions prior to training. Specifically for the additive update, we can instead predict as follows [5]:

$$\arg\max_{y \in \mathcal{Y}} f_y(x) - \log(\pi_y) + \log(\rho_y)$$

This can be interpreted as a consistent predictor for a performance measure that is related to the balanced error. Instead of each class being weighted equally in the

error rate, they are now weighted by the given test class distribution, and so this still minimizes the test risk under 0-1 loss.

Despite this, it is important to note that we could obtain different predictions for the same input, simply depending on the test set it belonged to — more specifically on that test set's class distribution. Because of this, one could argue that we have not truly learned a single model that we can apply to *any* test set, or even a single input, without knowledge that is not contained in our training data.

Chapter 4

Experimental Results

We now investigate the performance of the modifications suggested in the previous chapters for imbalanced and long-tailed classification problems in different contexts. We start by considering synthetic 2D data, which we can visualise to gain an intuitive confirmation of our claims that the modifications reduce the bias towards dominant classes.

Then we attain results of these methods applied to a 'real-world' data source, CIFAR-10 [9]. The full dataset consists of 60000 labelled images, each containing exactly one of ten different objects (the class associated with the image, such as 'airplane' or 'frog'), and there are 6000 belonging to each class. We attempt to reproduce results found in other papers on an imbalanced version of this, and make modifications to their accompanying code, to investigate some of the techniques we have discussed.

4.1 Synthetic Data - Binary

First we consider how different modifications perform both in settings where the data is linearly separable in the feature space, and where it is instead overlapping. This corresponds to an overparameterised setting where we can attain zero training error versus a simpler model where we cannot.

Suppose we have a binary classification task, which is imbalanced. We follow a setup used in Menon et al. 2021 [8], where there are 10000 independent samples of 2D points. 95% (the -1 class) are drawn from a 2D Gaussian centred at $(-1, -1)$, and the remaining 5% ($+1$ class) from one centred at $(1, 1)$, both with covariance matrix I_2, the 2-by-2 identity matrix. This training dataset is overlapping and therefore a linear classifier is severely underparameterised, as it cannot separate the classes.

Secondly we use a covariance matrix of $0.04I_2$ instead to obtain linearly separable data, so that there is a linear classifier that can achieve zero training error.

As seen in Section 3.1 any BE consistent classifier chooses the class y with the greatest class-conditional probability density $\mathbb{P}(x|Y = y)$. Since our two conditional distributions are Gaussian with the same covariance matrix, $x \in \mathbb{R}^2$ has a greater density conditional on the -1 class than $+1$ if and only if it is closer to $(-1, -1)$ than to $(1, 1)$, the means for each class. Hence the Bayes classifier's decision boundary is given by $x_1 + x_2 = 0$, the bisector of the two conditional means.

We begin with classifiers trained under different weighted loss functions.

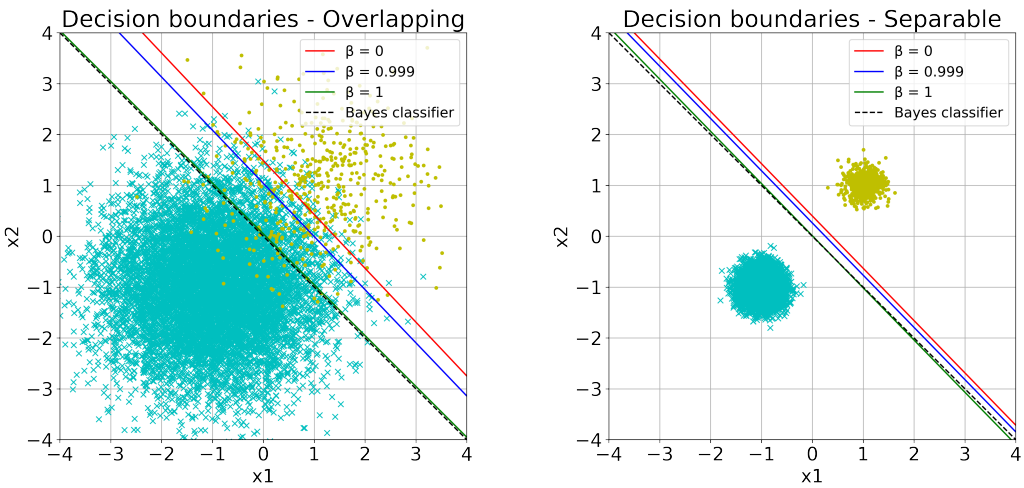

Figure 2: Classifiers trained under the loss weighted by the inverse effective class frequencies (9) for different values of β.

For the overlapping data on the left in Figure 2, we see that the learned class-weighted decision boundary is consistent with the Bayes classifier, confirming its BE consistency. We also observe the interpolating effect of varying β. We previously observed it interpolating between the uniform and empirical training class distributions. Now we see the learned classifier for $\beta = 0.999$ lying between the unweighted and class-weighed ones.

However it is less advantageous in the separable setting on the right. The three classifiers themselves are far more similar in this case. We still see that increasing β leads to the boundary moving closer to the dominant class as desired, but by less than before. This could be due to the better performance under the unweighted loss. While the optimal classifier under class-weighted loss should still be BE consistent, we see that the classifier learned in practice is still close to the one learned without

weighting. Training for infinite time, we could observe each of the three optimal classifiers becoming consistent, since it is possible to attain arbitrarily small training risk for each, so there is no guarantee that weighting is effective.

Now we compare different choices of margin adjustments. We compare results using a version of the LDAM (10) taking $C = 1$, the choice $F(j) = \log \pi_j$ for the margin in (11), and the margin in (12) resulting in the logit-adjusted loss. The second is referred to as the 'equalised' loss, in [8] and Figure 3 below. In the following plots, 'ERM' corresponds to that under the unweighted cross-entropy loss.

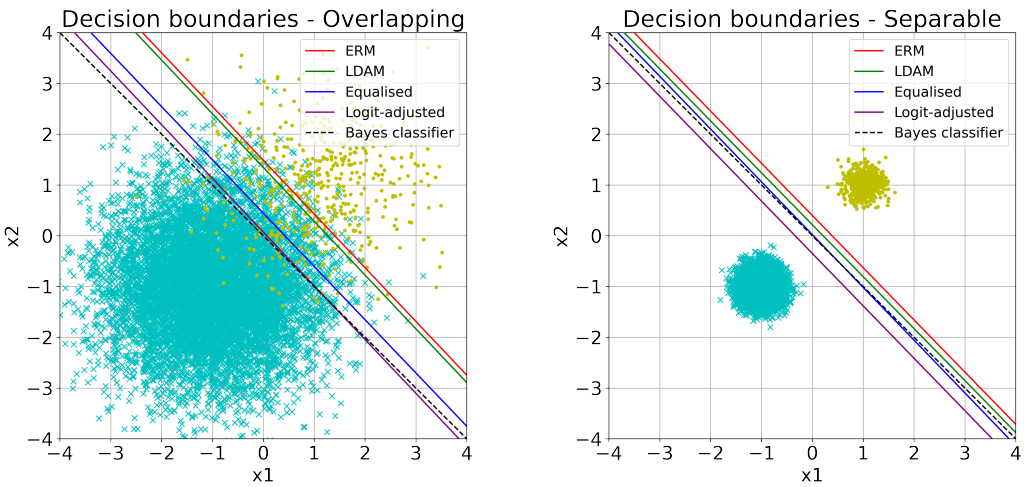

Figure 3: Classifiers trained on the margin-adjusted losses, and one without margin adjustment.

Note that the plot on the left simply reproduces the results for the same set up in [8] (see Figure 2 in that paper). We clearly see that both result in classifiers less biased towards class -1 than training without margin adjustment, as the boundary for ERM is closest to the rarer class. As with loss weighting, margin adjustment has a smaller, but still desired, effect for separable data. In the inseparable case, we see that the equalised loss performs significantly better than the LDAM - perhaps because it is of a similar form to the logit-adjusted loss, just without any adjustment accounting for the true class of the data point.

Again in the separable case, as ERM performs better than in the inseparable case, we see that margin-adjusted losses give decision boundaries even closer to the dominant class. In fact, we see that the logit-adjusted loss has now overcompensated for the class imbalance.

Given the ERM boundary is closer to the dominant class in the separable setting than in the inseparable one, we expect, and observe, that the logit-adjusted boundary exhibits the same behaviour. Therefore it changes from being empirically BE consistent for the inseparable data to being biased against the dominant class for the separable data.

Moreover we see that the ERM and logit-adjusted boundaries are now approximately reflections of eachother, in the decision boundary of the Bayes classifier. By this symmetry, we would expect them to perform equally well on a balanced test set in terms of accuracy — whereas their per-class error rates would be approximately reversed. In this case, our modification has not improved for generalisation on balanced test sets, but has, excessively, accounted for the class imbalance.

Next we look at similar plots for the logit adjustment methods in Section 3. In particular, we compare unweighted ERM to the additive update and the multiplicative update.

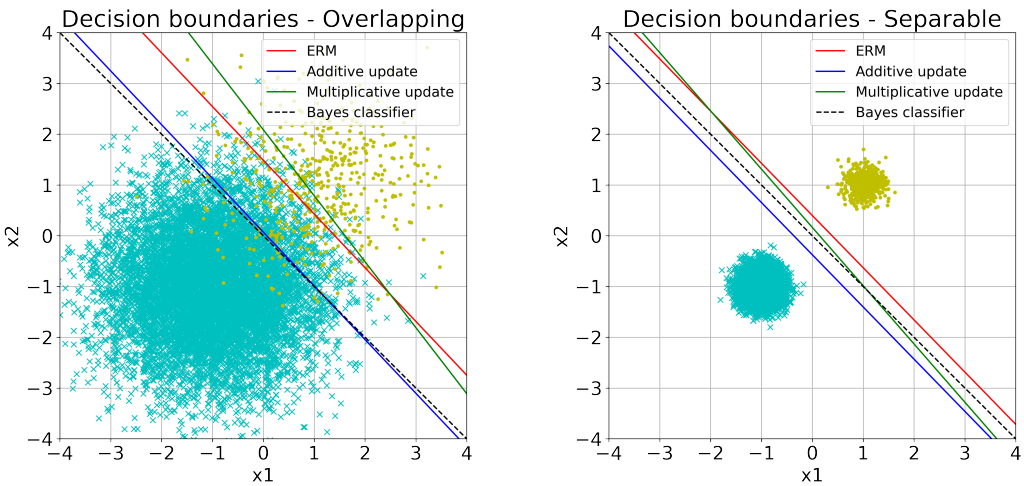

Figure 4: Classifiers resulting from logit adjustment. Multiplicative update means logits have been scaled by the inverse of class probability.

In the overlapping dataset we see the additive update clearly outperforms both the multiplicative update and ERM — again it is close to consistent with the Bayes classifier. This also empirically confirms the equivalence of the additive update and the logit-adjusted loss, as the same decision boundaries are learned, for both datasets in Figure 3 for the logit-adjusted loss and in Figure 4 for the update. Therefore, we again observe the overcompensation for the bias with the additive update in the separable dataset.

Perhaps surprisingly we see the multiplicative update performing badly. Unlike the additive update, this has no theoretical guarantee of BE consistency, so we cannot expect it to necessarily be an improvement.

We now highlight another disadvantage with multiplicative updates in general (including normalising by $||w_y||$). After training under ERM, there is a region of the graph where logits for the -1 class are positive, and ones for the $+1$ class are negative. No matter how we rescale the logits for each class, their signs remain the same and therefore inputs in that region will always be classified as -1. This is especially problematic as points originally misclassified as $+1$, despite being -1, often lie in this region. This only gets worse in the multiclass setting; when the ground truth logit is negative, if any of the false classes has a positive logit, we have the same problem. On the other hand, there is no point on the graph where it is impossible for an additive update to change the predicted class of that point, to one whose original logit was negative.

4.2 Synthetic Data - Multiclass

Having just considered imbalanced binary classification, we consider another long-tailed synthetic dataset in a multiclass setting.

Our data consists of 2000 2D points generated from 9 different Gaussian distributions (see Figure 5) centred at each point in $\{-2, 0, +2\}^2$. The proportions of the data drawn from the Gaussians with each mean are given below:

- 40%: $(0, 0)$ (cyan - dominant)
- 20%: each of $(0, \pm 2)$ (yellow - medium)
- 7.5%: each of $(\pm 2, 0)$ (green - medium)
- 1.25%: each of $(\pm 2, \pm 2)$ (red - rare)

Each distribution has covariance matrix $0.25 I_2$. We see that the rare classes and the dominant class do not overlap, while there is some overlap between the dominant and medium classes, and between medium and rare ones. The Bayes classifier for the balanced error again simply returns the class with the closest mean to the input.

Again we train a linear classifier, firstly under unweighted ERM, then we apply the additive update, and secondly with the class-weighted loss. In Figure 6 we visualise decision boundaries by colouring each point on a dense mesh, according to (the frequency of) its predicted class, as we did in Figure 5.

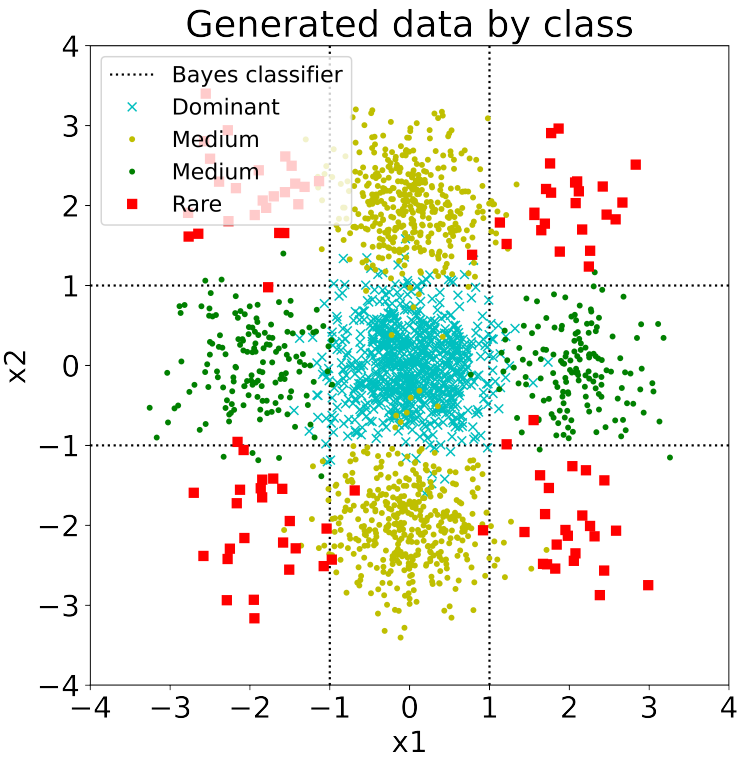

Figure 5: Data generated from 9 different Gaussians in different frequencies. Points with the same colour come from classes with the same frequency, and those with the same shape from classes with the same frequency group (dominant, medium or rare).

Here we see that the cross-entropy loss gives a model that is not overly biased towards the dominant class, though the rare classes still suffer because of their proximity to the medium ones. As we had with separable data in the binary case, the additive update is now over-penalising the dominant and medium classes to favour the rare ones. This is expected given that the unweighted model did well to classify the dominant class, so the additive update after training would be harmful to it.

Class-weighting now performs best, giving a classifier much closer to the Bayes one. In particular the dominant-medium boundaries agree almost entirely with that of the Bayes classifier, and medium-rare boundaries are now closer to the medium regions.

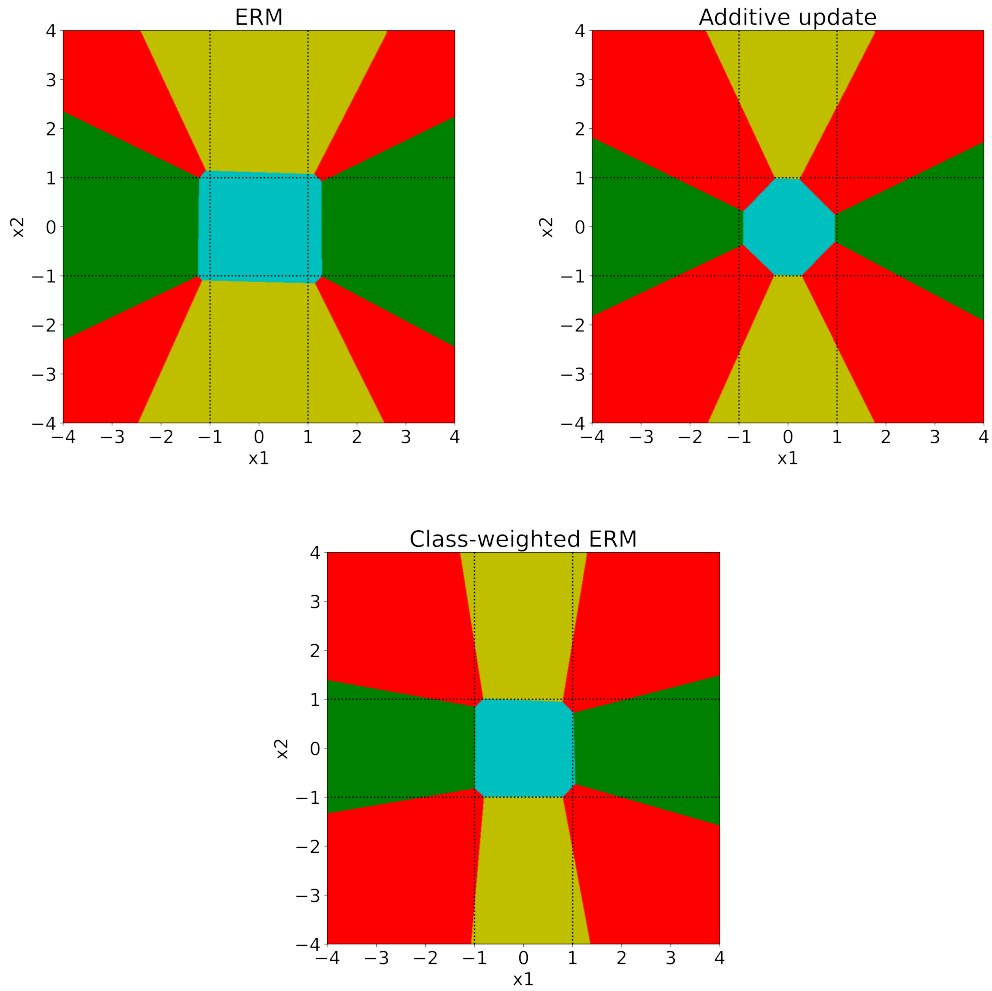

Figure 6: A coloured visualisation of $[-4, 4]^2$ indicating how the learned classifier predicts each point in the region. The dotted black line represents the decision boundary of the Bayes classifier.

4.3 CIFAR-10

We now consider results from training on a long-tailed version of CIFAR-10 with a balanced test set. The standard CIFAR-10 test dataset was used, containing 10000 images in total. The training dataset contained 12406 images for each paper considered, and was sampled such that the class distribution decayed exponentially (see Figure 1, where $\beta = 1$ represents the empirical class distribution). The (class) *imbalance ratio*, $\rho = \max_j \pi_j / \min_j \pi_j$, was 100, so that the most and least frequent classes contained 5000 and 50 training images respectively.

For each method, we provide the accuracies reported in the relevant paper, and

the accuracies attained when attempting to reproduce those results ourselves. For brevity, we now often refer to the logit-adjusted loss as 'LA'.

We reproduced the results by running code made available by the authors of the papers, changing the parameters from their default values to match those given in the papers if necessary. Each network was trained using SGD with momentum of 0.9, mini-batches of 128 observations and for 200 epochs in total. The learning rate increased linearly to a base learning rate of 0.1 over the first 5 epochs. It decayed by a multiplicative factor of 0.1 in [8] (see Table 1) and 0.01 in [2] (Table 2) at the 160th epoch, and again by the same factor at the 180th. In each case, the ResNet-32 architecture is taken as the backbone of the neural network to be trained.

Paper	**Method**	**Reported**	**Reproduced**
	ERM	72.84	69.69
Menon et. al [8]	Additive update	77.40	76.32
	LA	77.67	75.04

Table 1: Accuracies on the balanced test set. Specifically, reported results were listed in Table 3 of [8].

While we could not quite reproduce the same accuracies reported in [8], we still saw that both of the equivalent methods of the logit-adjusted loss and the post-training additive update significantly outperform the unmodified cross-entropy. We might expect these two methods to give more similar accuracies if we were to train each for more epochs.

Paper	Method	Reported	Reproduced Final	Reproduced Best
Cao et. al [2]	ERM	70.36	70.98	71.08
	LDAM	73.35	72.89	74.14
	LDAM+DRW	77.03	77.14	77.67
Cui et. al [4]	$\beta = 0.9$	69.93	70.98	71.08
	$\beta = 0.99$	71.47	71.11	72.60
	$\beta = 0.999$	70.76	72.43	73.15
	$\beta = 0.9999$	72.37	70.64	71.40
	$\beta = 1$		71.57	71.92
	LDAM+DRW ($\beta = 1$)		77.58	77.98
	LA		78.10	78.33
	LA+DRW ($\beta = 1$)		79.49	79.74

Table 2: Accuracies on the balanced test set. All reproduced results were obtained using the training setup as indicated in the corresponding papers, and based on code provided corresponding to [2]. Specifically reported results were listed in Tables 3 of [2] and 8 of [4]. Model accuracy was computed at the end of each epoch, and we report the 'final' and 'best' of these. By default, DRW uses $\beta = 0.9999$.

We now consider results for the second setup in Table 2. Results reported in [4] are obtained with an identical setup, so we use the code for [2] to reproduce these. See Code 5 in the Appendix for details on the choice of C in (10). Cao et al. also introduces a 'deferred re-weighting' (DRW) procedure that empirically gives much better results. Under DRW, we train using a given loss under unweighted ERM for the first 160 epochs, and then weighted ERM for the remaining 40, with the same learning rate schedule. By default the weighting gives a loss of the form (9) with $\beta = 0.9999$ in the provided code, though we also implement a class-weighted version.

As expected we see that LA outperforms both LDAM (with and without DRW), and weighted losses for each β. Given we observe LDAM benefits from DRW, we train LA with DRW (our class-weighted implementation) and observe a more marginal gain in accuracy.

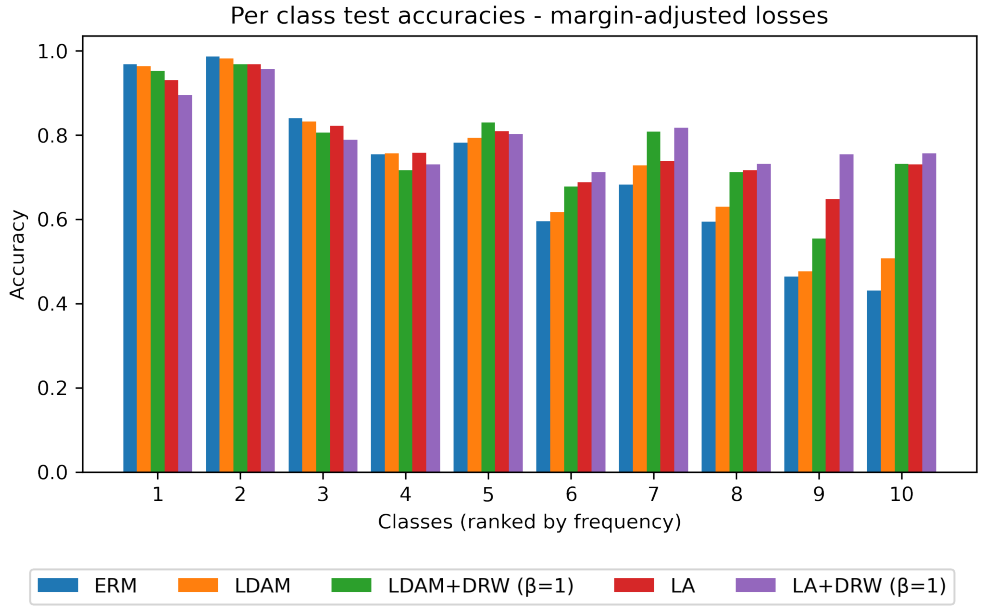

Figure 7: Per-class test accuracies from margin-adjusted losses, with and without DRW.

We now consider the effect of DRW on the learned model itself. While DRW and standard training are identical at first, in the final stages of training, DRW penalises misclassification on rare classes more. Figure 7 indicates for each of LDAM and LA, DRW results in models that are less accurate on dominant classes and more accurate on rare ones, than their counterparts trained as standard. For the rarer classes, all modifications outperform ERM under cross-entropy. LDAM benefits from the DRW far more than LA does, particularly for the rare classes. This may be because weighting LA in the final stages of training breaks its BE consistency, as it is no longer of the form required for Theorem 6. It may lead to over-penalising of dominant classes but, because of the reduced learning rate at the final stages, this is not a significant disadvantage.

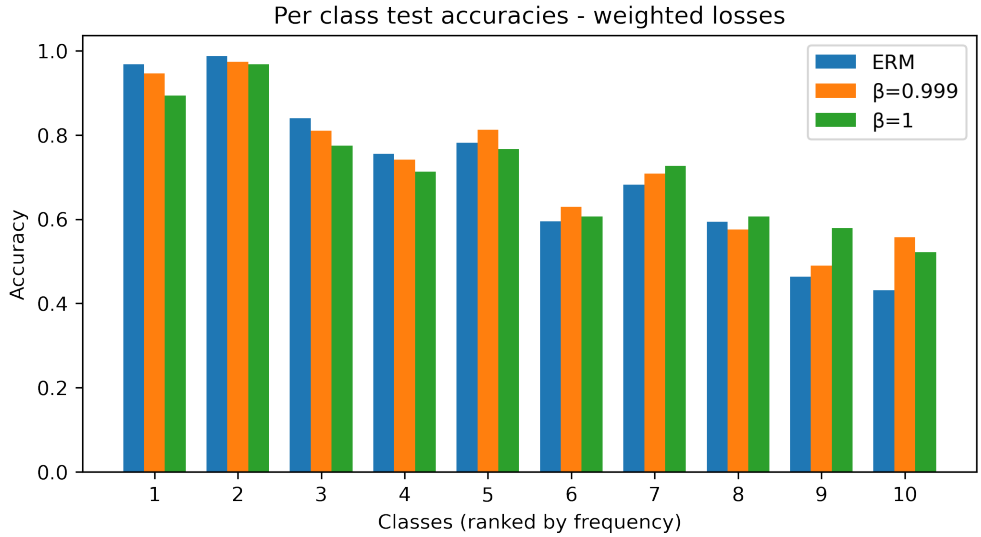

Figure 8: Per-class test accuracies from unweighted ERM, a weighted loss in [4] and the class-weighted loss ($\beta = 1$).

Next we look at the effect of weighting on per-class accuracies in Figure 8. Recall the interpolation with β of decision boundaries with our binary synthetic data in Figure 2. We observe a similar effect on per-class test accuracies; as β increases, dominant class performance decreases, while rare class performance improves in general. Despite this the overall test accuracies in Table 2 do not indicate a strong pattern as β increases.

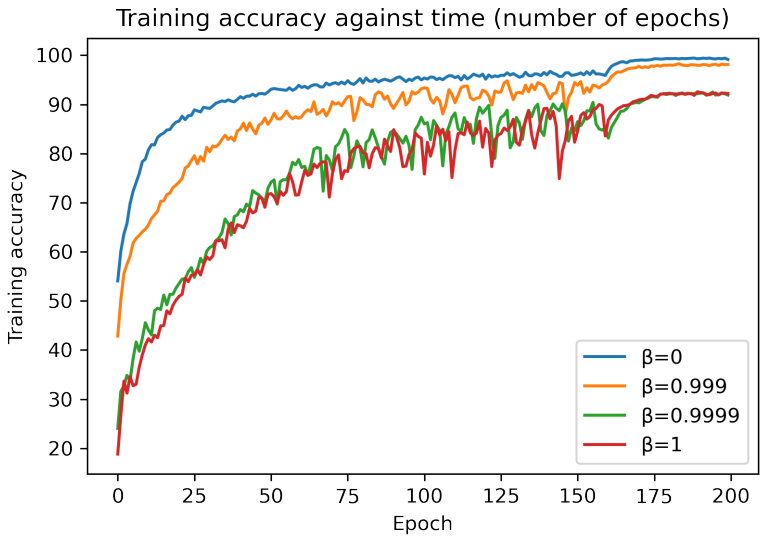

Figure 9: Training accuracies against number of epochs for different weightings of the loss. Weighting becomes more extreme with increasing β.

One might wonder why we should use DRW, where we only re-weight the loss near the end of training, as opposed to weighting it throughout. Firstly we saw that DRW consistently gives slightly worse performance on the dominant classes than not weighting at all. Using a weighted loss throughout, rather than having a delay, could make this problem worse.

Secondly, Ren et al. [10] notes that weighting can lead to unstable training. If some class weights for the loss are much larger than others, gradients with respect to model parameters can scale accordingly. As a result, parameter updates can more easily overshoot in the direction of the gradient. One solution, which we leave for study in further work, could be to use the ADAM optimizer [7] instead of SGD, which uses a different learning rate for each model parameter to compensate for this scaling.

We verify this in Figure 9, where we observe more extreme weighting with increasing β resulting in training accuracy improving less smoothly, and our final model achieving worse training accuracy.

Chapter 5

Conclusion

In this dissertation we looked at why unweighted ERM performs poorly when learning from a long-tailed training set. We outlined some methods to handle the discrepancy between training and test class distributions, and focussed on the more intuitive methods: 'cost-sensitive learning', using modified loss functions, and 'logit adjustment', training a network as usual but modifying the outputs, to encourage the rare classes.

We showed that specific methods of loss weighting and logit adjustment were each statistically motivated to address the problem from the perspective of risk minimization. Meanwhile other choices of margins to enforce into the loss function are intuitively expected to counteract the undesired bias. Literature on weighted ERM notes it can be ineffective in separable settings and unstable in training. We provided some brief mathematical reasoning that could explain why.

Finally we performed experiments to compare the performance of the different techniques. Indeed we saw that weighting provided more inconsistent and smaller gains than some margin and logit adjustment techniques, though it could still be effective for extremely underparameterised models. Weighting was effective in combination with the other techniques, when delayed until the final stages of training.

Given more time, the closely related method of resampling could be investigated. Commonly used examples of this are oversampling, drawing repeated samples from rare classes, and undersampling, dropping examples from dominant classes, to form a more balanced training set. However this, and the other techniques investigated, often pay the price for better rare class performance by losing accuracy on the dominant classes. With so few examples in the rare classes, one could argue that this is inevitable.

We could also experiment with the alternative class probability estimators' performance (Section 2.5), which could result in accuracy gains especially on a larger dataset like iNaturalist, where single-shot classes in the training set are likely.

Appendix A

Code

We now include the key parts of the code used to obtain experimental results and plots in this dissertation. This includes our own Jupyter notebooks (exported as scripts), and sections of code corresponding to other papers that we have modified or added to.

A.1 Synthetic Data

Firstly we have the code for the 2D binary problem. We only show the code for the inseparable data. The separable case has identical code, except for a different covariance matrix, filenames and titles for the resulting plots.

```
# %%
import torch
import torch.nn.functional as F
from torch import nn
import numpy as np
import matplotlib.pyplot as plt
from copy import deepcopy

# %%
import matplotlib.pylab as pylab
params = {'legend.fontsize': 'x-large',
          'axes.labelsize': 'x-large',
          'axes.titlesize':'x-large',
          'xtick.labelsize':'x-large',
          'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)

# %% [markdown]
# ### Class for simple Linear classifier

# %%
# A very simple linear neural network with input dimension 2
# and output dimension 2 (the discriminant function for each class)
```

```python

class LinearClassifier(torch.nn.Module):
    def __init__(self, input_dim=2, output_dim=2):
        super(LinearClassifier, self).__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        x = self.linear(x)
        return x

# %%
# a function that takes in a binary linear classifier's weights and biases
# and plots the decision boundary line in [-4,4] * [-4,4]

def plot_db(wb, modelType='[modelType]', lins='r'):

    w0, w1, b0, b1 = wb[0], wb[1], wb[2], wb[3]
    x0 = np.linspace(-4,4,100)
    x1 = (b1 - b0 - (w0[0] - w1[0]) * x0) / (w0[1] - w1[1])
    plt.plot(x0, x1, lins, label=f'{modelType}')

# %% [markdown]
# ### Synthetic data generation and visualisation
#
# 10,000 samples - 95% from Gaussian centered at (-1, -1) with covariance I; the remaining 5% from Gaussian centered at (1, 1) with covariance I.

# %%
N = 10000
class_prob = [0.95, 0.05]
class_freq = [int(N * i) for i in class_prob]

covariance = np.identity(2) # change to np.identity(2) * 0.04 to get separable data
np.random.seed(2022) # fix seed for a reproducable training set
sampA = np.random.multivariate_normal([-1,-1], covariance, class_freq[0])
sampB = np.random.multivariate_normal([1,1], covariance, class_freq[1])
np.random.seed()

X_values = np.concatenate((sampA, sampB), axis=0) # N-by-2 data matrix
y_flat = np.array([0] * class_freq[0] + [1] * class_freq[1]) # row vector of labels

index_shuffle = np.arange(len(X_values))
np.random.shuffle(index_shuffle)

# shuffle the rows of data matrix and labels

X_values = X_values[index_shuffle]
```

```
y_flat = y_flat[index_shuffle]

# convert to Torch tensors

X_train = torch.tensor(X_values).type(torch.FloatTensor)
y_train = torch.from_numpy(y_flat).view(-1,1).type(torch.LongTensor
    )

# %% [markdown]
# ### Training the standard classifier
#
# With single layer NN with Cross-Entropy Loss (for binary
    classification equivalent to a sigmoid classifier)

# %%
# Create a Linear neural network with randomly initialised weights
    and biases
# Each trained network begins with the same initialisation, as they
     all start
# as copies of this network

model_bin_orig = LinearClassifier()

# %%
# input: model
# output: weights and biases in an array
# optional: additive update to biases or multiplicative update to
    weight and biases

def getwb(model, add=[0,0], mult=[1,1]):
    return [model.linear.weight[0].detach().numpy()/mult[0], model.
    linear.weight[1].detach().numpy()/mult[1], (model.linear.bias
    [0].detach().numpy()+add[0])/mult[0], (model.linear.bias[1].
    detach().numpy()+add[1])/mult[1]]

# %%
# input: model, optimizer for parameters for that model, criterion
    (or loss)
# optional: additive update to outputs before passing through loss
    function
# and trains model and updates its parameters

def modelTrain(model, optimizer, criterion, adjust=0):
    for epoch in range(10000):

        # zero the gradients
        optimizer.zero_grad()

        # forward
        output = model(X_train)
        loss = criterion(output + adjust, y_train.view(-1))

        # backward
        loss.backward()
```

```
        # optimize
        optimizer.step()

# %%
# unweighted cross-entropy ERM
model_bin = deepcopy(model_bin_orig)
optimizer_bin = torch.optim.SGD(model_bin.parameters(), lr=0.1, weight_decay=0)
criterion_bin = torch.nn.CrossEntropyLoss()

# %%
modelTrain(model_bin, optimizer_bin, criterion_bin)

# %% [markdown]
# ### Loss modified
#
# Balance the loss

# %%
# class-weighted cross-entropy ERM
model_bin_bal = deepcopy(model_bin_orig)
optimizer_bin_bal = torch.optim.SGD(model_bin_bal.parameters(), lr=0.1, weight_decay=0)
criterion_bin_bal = torch.nn.CrossEntropyLoss(weight = torch.tensor(np.reciprocal(class_prob)).type(torch.FloatTensor))

# %%
modelTrain(model_bin_bal, optimizer_bin_bal, criterion_bin_bal)

# %%
beta = 0.999
cui_weights = [(1 - beta) / (1 - beta ** i) for i in class_freq] # inverse effective number for each class
cui_weights = [i / sum(cui_weights) for i in cui_weights] # normalise weights
cui_weights = torch.tensor(cui_weights).type(torch.FloatTensor)

# %%
# weighted cross-entropy with beta=0.999
model_bin_cui = deepcopy(model_bin_orig)
optimizer_bin_cui = torch.optim.SGD(model_bin_cui.parameters(), lr=0.1, weight_decay=0)
criterion_bin_cui = torch.nn.CrossEntropyLoss(weight = cui_weights)

# %%
modelTrain(model_bin_cui, optimizer_bin_cui, criterion_bin_cui)

# %% [markdown]
# ### Cao's LDAM loss

# %%
# LDAM Loss taken from Cao et. al code and modified

```

```python
class LDAMLoss(nn.Module):
    def __init__(self, class_prob, weight=None):
        super(LDAMLoss, self).__init__()
        delta = 1.0 / np.sqrt(np.sqrt(class_prob))
        delta = torch.FloatTensor(delta)
        self.delta = delta
        self.weight = weight

    def forward(self, outputs, target):
        # one-hot encodes the binary labels
        index = torch.zeros_like(outputs, dtype=torch.uint8)
        index.scatter_(1, target.data.view(-1, 1), 1)
        index_float = index.type(torch.FloatTensor)

        # column vector of the margin adjustments
        batch_m = torch.matmul(self.delta[None, :], index_float.transpose(0,1))
        batch_m = batch_m.view((-1, 1))

        # computes logits modified by subtracting the margin for each example (determined by the true class) from the each output for it
        outputs_m = outputs - batch_m

        # for each example, replace the logit for the true class by the modified one, and keep the others the same
        output = torch.where(index, outputs_m, outputs)

        # computes the mean of cross-entropy losses for each example and its adjusted output
        return F.cross_entropy(output, target, weight=self.weight)

# %%
# Train with Cao's LDAM loss
model_bin_cao = deepcopy(model_bin_orig)
optimizer_bin_cao = torch.optim.SGD(model_bin_cao.parameters(), lr=0.1, weight_decay=0)
criterion_bin_cao = LDAMLoss(class_prob)

# %%
modelTrain(model_bin_cao, optimizer_bin_cao, criterion_bin_cao)

# %% [markdown]
# ### Equalised Loss

# %%
# Loss with margin log(pi_j) for where j is the negative class

class EqLoss(nn.Module):
    def __init__(self, class_prob, weight=None):
        super(EqLoss, self).__init__()
        delta = np.log(class_prob)
        delta = torch.FloatTensor(delta)
        self.delta = delta
```

```
        self.weight = weight

    def forward(self, outputs, target):
        # one-hot encodes the binary labels
        index = torch.zeros_like(outputs, dtype=torch.uint8)
        index.scatter_(1, target.data.view(-1, 1), 1)

        # column vector of the margin adjustments
        batch_m = self.delta.repeat(outputs.shape[0],1)

        # computes logits modified by adding the margin, log(pi_j), to each logit f_j for each example
        outputs_m = outputs + batch_m

        # for each example, replace the logit for the negative classes by the modified one in outputs_m, and keep original logit for true class
        output = torch.where(index, outputs, outputs_m)

        # computes the mean of cross-entropy losses for each example and its adjusted output
        return F.cross_entropy(output, target, weight=self.weight)

# %%
# train with equalised loss
model_bin_eq = deepcopy(model_bin_orig)
optimizer_bin_eq = torch.optim.SGD(model_bin_eq.parameters(), lr=0.1, weight_decay=0)
criterion_bin_eq = EqLoss(class_prob)

# %%
modelTrain(model_bin_eq, optimizer_bin_eq, criterion_bin_eq)

# %% [markdown]
# ### Logit-Adjusted Loss

# %%
# train with logit-adjusted loss
model_bin_la = deepcopy(model_bin_orig)
optimizer_bin_la = torch.optim.SGD(model_bin_la.parameters(), lr=0.1, weight_decay=0)
criterion_bin_la = torch.nn.CrossEntropyLoss()

# %%
# in the loss, each logit f_j is replaced by f_j + log(pi_j) in the cross-entropy
la_margin_mat = torch.Tensor(np.array([np.log(class_prob) for _ in range(N)])).type(torch.float32)

# %%
modelTrain(model_bin_la, optimizer_bin_la, criterion_bin_la, adjust=la_margin_mat)

# %% [markdown]
```

```
# ### Plot the decision boundaries
#
# Decision boundaries for each linear classifier (Vanilla ERM, logit-adjusted, reweighted and modified loss).
#
# We also include the Bayes classifier decision boundaries (y = - x).

# %%
# Show the synthetic data
plt.plot(sampA[:,0], sampA[:,1], 'cx')
plt.plot(sampB[:,0], sampB[:,1], '.y')
# Plot the decision boundaries - can plot boundaries for different models
# Plotting logit-adjusted models (multiplicative and additive updates) can be done as follows:

# plot_db(getwb(model_bin, mult=class_prob), modelType='Multiplicative update', lins='g')
# plot_db(getwb(model_bin, add=-np.log(class_prob)), modelType='Additive update', lins='b')


plot_db(getwb(model_bin), modelType='\u03B2 = 0', lins='r')
plot_db(getwb(model_bin_cui), modelType=f'\u03B2 = {beta}', lins='b')
plot_db(getwb(model_bin_bal), modelType='\u03B2 = 1', lins='g')
plot_db([[0,0], [1,1], 0, 0], modelType='Bayes classifier', lins='k--')
# Add legend, title, axes labels, general formatting, saving the plot
plt.legend(loc='best', fontsize=15)
plt.title('Decision boundaries - Overlapping', fontsize=25)
plt.xlabel('x1', fontsize=20)
plt.ylabel('x2', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.grid()
plt.axis('scaled')
plt.axis([-4,4,-4,4])
figure = plt.gcf()
figure.set_size_inches(8,8)
plt.savefig('imbalOverlapWeighted.png', dpi=1200)
plt.show()
```

Code 1: This code generates the plots in Section 4.1.

Next we present the code for the multiclass problem:

```
# %%
import torch
import numpy as np
import matplotlib.pyplot as plt
from copy import deepcopy

```

```python
# %%
import matplotlib.pylab as pylab
params = {'legend.fontsize': 'x-large',
          'axes.labelsize': 'x-large',
          'axes.titlesize':'x-large',
          'xtick.labelsize':'x-large',
          'ytick.labelsize':'x-large'}
pylab.rcParams.update(params)

# %% [markdown]
# ### Class for a simple Linear classifier

# %%
# Now there are 9 classes, so 9 logits/output nodes

class LinearClassifier(torch.nn.Module):
    def __init__(self, input_dim=2, output_dim=9):
        super(LinearClassifier, self).__init__()
        self.linear = torch.nn.Linear(input_dim, output_dim)

    def forward(self, x):
        x = self.linear(x)
        return x

# %% [markdown]
# ### Synthetic data generation

# %%
class_frequencies = [32/80] + [16/80]*2 + [6/80]*2 + [1/80]*4
N = 2000

np.random.seed(2022)
covariance = np.identity(2) * 0.25
# A is the most dominant class, {B,C} are next most dominant class, then {D,E} then {F,G,H,I}
sampA = np.random.multivariate_normal([0,0], covariance, int(N*class_frequencies[0]))
sampB = np.random.multivariate_normal([0,2], covariance, int(N*class_frequencies[1]))
sampC = np.random.multivariate_normal([0,-2], covariance, int(N*class_frequencies[2]))
sampD = np.random.multivariate_normal([2,0], covariance, int(N*class_frequencies[3]))
sampE = np.random.multivariate_normal([-2,0], covariance, int(N*class_frequencies[4]))
sampF = np.random.multivariate_normal([-2,2], covariance, int(N*class_frequencies[5]))
sampG = np.random.multivariate_normal([2,2], covariance, int(N*class_frequencies[6]))
sampH = np.random.multivariate_normal([-2,-2], covariance, int(N*class_frequencies[7]))
sampI = np.random.multivariate_normal([2,-2], covariance, int(N*class_frequencies[8]))
np.random.seed()
```

```
X_values = np.concatenate((sampA, sampB, sampC, sampD, sampE, sampF, sampG, sampH, sampI), axis=0)
y_flat = np.array([])
for i in range(9):
    y_flat = np.concatenate((y_flat, np.array([i] * int(N*class_frequencies[i]))))

index_shuffle = np.arange(len(X_values))
np.random.shuffle(index_shuffle)

X_values = X_values[index_shuffle]
y_flat = y_flat[index_shuffle]

X_train = torch.tensor(X_values).type(torch.FloatTensor)
y_train = torch.from_numpy(y_flat).view(-1,1).type(torch.LongTensor)

# %% [markdown]
# #### Plots
#
# ##### This just shows how our training dataset.

# %%
# Draw decision boundaries of the Bayes classifier
plt.axvline(x = 1, color='k', linestyle=':', label='Bayes classifier')
plt.axvline(x = -1, color='k', linestyle=':')
plt.axhline(y = 1, color='k', linestyle=':')
plt.axhline(y = -1, color='k', linestyle=':')
plt.xlim([-4, 4])
plt.ylim([-4, 4])
# Plot the data
plt.plot(sampA[:,0], sampA[:,1], 'cx', label='Dominant')
plt.plot(sampB[:,0], sampB[:,1], '.y', label='Medium')
plt.plot(sampC[:,0], sampC[:,1], '.y')
plt.plot(sampD[:,0], sampD[:,1], '.g', label='Medium')
plt.plot(sampE[:,0], sampE[:,1], '.g')
plt.plot(sampF[:,0], sampF[:,1], 'sr', label='Rare')
plt.plot(sampG[:,0], sampG[:,1], 'sr')
plt.plot(sampH[:,0], sampH[:,1], 'sr')
plt.plot(sampI[:,0], sampI[:,1], 'sr')
# Formatting plot, adds title, axis labels, saves plot etc.
plt.title('Generated data by class', fontsize=25)
plt.xlabel('x1', fontsize=20)
plt.ylabel('x2', fontsize=20)
plt.xticks(fontsize=20)
plt.yticks(fontsize=20)
plt.legend(loc='upper left', fontsize=15)
plt.axis('scaled')
plt.axis([-4,4,-4,4])
figure = plt.gcf()
figure.set_size_inches(8,8)
plt.savefig('imbalMultiData', dpi=1500)
plt.show()
```

```python

# %% [markdown]
# A mesh for visualising decision boundaries

# %%
# Creates a mesh of ~800^2 points in [-4,4]^2
x1_fit = np.linspace(-4,4,800)
x2_fit = np.linspace(-4,4,800)

mesh_grid = np.meshgrid(x1_fit, x2_fit)

X_mesh = np.hstack([mesh_grid[0].reshape(-1, 1), mesh_grid[1].reshape(-1, 1)]).astype(np.float32)
X_mesh = torch.from_numpy(X_mesh)

# %%
# Produces scatter plot, with Bayes classifier. Each point on the dense mesh is coloured by the predicted class

def mesh_plot(y_pred, title, fileName):
    plt.axvline(x = 1, color='k', linestyle=':')
    plt.axvline(x = -1, color='k', linestyle=':')
    plt.axhline(y = 1, color='k', linestyle=':')
    plt.axhline(y = -1, color='k', linestyle=':')
    plt.title(title, fontsize=25)
    y_pred_colours = [['c','y','y','g','g','r','r','r','r'][i] for i in y_pred.detach().numpy()]
    plt.scatter(X_mesh[:,0], X_mesh[:,1], c = y_pred_colours, marker='.')
    plt.xlabel('x1', fontsize=20)
    plt.ylabel('x2', fontsize=20)
    plt.xticks(fontsize=20)
    plt.yticks(fontsize=20)
    plt.axis('scaled')
    plt.axis([-4,4,-4,4])
    figure = plt.gcf()
    figure.set_size_inches(8,8)
    plt.savefig(fileName, dpi=800)
    plt.show()

# %% [markdown]
# #### Linear classifier

# %%
model_orig = LinearClassifier()

# %%
def modelTrain(model, optimizer, criterion):
    for epoch in range(50000):
        # zero the gradients
        optimizer.zero_grad()

        # forward
        output = model(X_train)
```

```
        loss = criterion(output, y_train.view(-1))

        # backward
        loss.backward()

        # optimise
        optimizer.step()

# %%
model_erm = deepcopy(model_orig)
criterion_erm = torch.nn.CrossEntropyLoss()
optimizer_erm = torch.optim.SGD(model_erm.parameters(), lr=0.01)

# %%
# trains model_erm under vanilla ERM with cross-entropy
modelTrain(model_erm, optimizer_erm, criterion_erm)

# %%
y_pred_erm = model_erm.forward(X_mesh)
y_pred_erm = torch.argmax(y_pred_erm, axis = 1)
mesh_plot(y_pred_erm, 'ERM', 'imbalMultiVanilla.png')

# %% [markdown]
# ### Linear classifier with additive update

# %%
adjustments = torch.from_numpy(np.log(class_frequencies)).type(torch.FloatTensor)
# modified predictions with additive update
y_pred_adjusted = torch.argmax(model_erm.forward(X_mesh) - adjustments, axis = 1)
mesh_plot(y_pred_adjusted, 'Additive update', 'imbalMultiAdditive.png')

# %% [markdown]
# ### Linear classifier with modified loss
#
# In-built weighting of CrossEntropyLoss with weight argument in PyTorch. This gives the loss function as in Menon (4); corresponding to 'balancing' the loss.

# %%
model_w = deepcopy(model_orig)
optimizer_w = torch.optim.SGD(model_w.parameters(), lr=0.01)
# class-weight the cross-entropy
criterion_w = torch.nn.CrossEntropyLoss(weight = torch.tensor(np.reciprocal(class_frequencies)).type(torch.FloatTensor))

# %%
modelTrain(model_w, optimizer_w, criterion_w)

# %%
y_pred_w = model_w.forward(X_mesh)
y_pred_w = torch.argmax(y_pred_w, axis = 1)
```

```
mesh_plot(y_pred_w, 'Class-weighted ERM', 'imbalMultiWeighted.png')
```

Code 2: This code generates the plots in Section 4.2.

A.2 CIFAR-10

For results in Section 4.3, we modified existing code from previous papers. Firstly for results in Table 1, we ran code for [8], available at: `https://github.com/google-research/google-research/tree/master/logit_adjustment`. Datasets were downloaded as TFRecords to the /data directory. The source code was slightly modified as in Code 3, then models were trained as instructed in the README.md file.

```
'cifar10-lt':
  Dataset(
  'cifar10-lt',
  10,
  'cifar10-lt_train.tfrecord',
  'cifar10_test.tfrecord',
  12406,
  10000,
  200, # changed to 200 from 1241 as in Menon et al.
  [  # (multiplier, epoch to start) tuples
    # originally (1.0, 20), (0.1, 604), (0.01, 926), (0.001, 1128)
    (1.0, 5), (0.1, 160), (0.01, 180)
  ])
```

Code 3: This adjusts the learning rate schedule to match setup in the paper. We modified the 'dataset_mappings()' function in 'utils.py'.

For the remaining results in Table 2, we added to code for [2], available at `https://github.com/kaidic/LDAM-DRW`. First we look at the code for the training rule:

```
if args.train_rule == 'None':
  train_sampler = None
  per_cls_weights = None
elif args.train_rule == 'Resample':
  train_sampler = ImbalancedDatasetSampler(train_dataset)
  per_cls_weights = None
elif args.train_rule == 'Reweight':
  train_sampler = None
  beta = 0.9 # beta = 0.9999 (originally)
  effective_num = 1.0 - np.power(beta, cls_num_list)
  per_cls_weights = (1.0 - beta) / np.array(effective_num)
  per_cls_weights = per_cls_weights / np.sum(per_cls_weights) * len
    (cls_num_list)
  per_cls_weights = torch.FloatTensor(per_cls_weights).cuda(args.
    gpu)
elif args.train_rule == 'ClassWeight':
  train_sampler = None
  per_cls_weights = 1.0 / np.array(cls_num_list) # inverse class
    freq
```

```
17   per_cls_weights = per_cls_weights / np.sum(per_cls_weights) * len
      (cls_num_list) # rescales weights
18   per_cls_weights = torch.FloatTensor(per_cls_weights).cuda(args.
      gpu)
19 elif args.train_rule == 'DRW':
20   train_sampler = None
21   idx = epoch // 160
22   betas = [0, 0.9999]
23   effective_num = 1.0 - np.power(betas[idx], cls_num_list)
24   per_cls_weights = (1.0 - betas[idx]) / np.array(effective_num)
25   per_cls_weights = per_cls_weights / np.sum(per_cls_weights) * len
      (cls_num_list)
26   per_cls_weights = torch.FloatTensor(per_cls_weights).cuda(args.
      gpu)
27 elif args.train_rule == 'DRW_ClassWeight':
28   train_sampler = None
29   idx = epoch // 160
30   per_cls_weights = 1.0 / np.array(cls_num_list) if idx == 1 else
      np.array([1] * len(cls_num_list)) # class-weighted if epoch
      >=160, weights all equal to 1 otherwise
31   per_cls_weights = per_cls_weights / np.sum(per_cls_weights) * len
      (cls_num_list) # rescale weights
32   per_cls_weights = torch.FloatTensor(per_cls_weights).cuda(args.
      gpu)
33 else:
34   warnings.warn('Sample rule is not listed')
35
36 if args.loss_type == 'CE':
37   criterion = nn.CrossEntropyLoss(weight=per_cls_weights).cuda(args
      .gpu)
38 elif args.loss_type == 'LogAdj':
39   criterion = LogAdjLoss(cls_num_list=cls_num_list, weight=
      per_cls_weights).cuda(args.gpu)
40 elif args.loss_type == 'LDAM':
41   criterion = LDAMLoss(cls_num_list=cls_num_list, max_m=0.5, s=30,
      weight=per_cls_weights).cuda(args.gpu)
```

Code 4: The training rule cases in the training loop in cifar_train.py.

We added the 'DRW_Classweight' case to use the class-weighted loss as opposed to $\beta = 0.9999$. For weighted losses, we used 'Reweight' for $\beta < 1$ (and varied the value of β by changing line 9) and 'ClassWeight' for $\beta = 1$ (which we added). We also added the option to use the logit-adjusted loss, by writing one into the losses.py file and importing it at the start of 'cifar_train.py'. The relevant losses are given below.

```
1 class LDAMLoss(nn.Module):
2
3   def __init__(self, cls_num_list, max_m=0.5, weight=None, s=30):
4     super(LDAMLoss, self).__init__()
5     m_list = 1.0 / np.sqrt(np.sqrt(cls_num_list))
6     m_list = m_list * (max_m / np.max(m_list)) # rescale so maximum
      of the margins is max_m=0.5
7     m_list = torch.cuda.FloatTensor(m_list)
```

```
    self.m_list = m_list
    assert s > 0
    self.s = s
    self.weight = weight

  def forward(self, x, target):
    index = torch.zeros_like(x, dtype=torch.uint8)
    index.scatter_(1, target.data.view(-1, 1), 1)

    index_float = index.type(torch.cuda.FloatTensor)
    batch_m = torch.matmul(self.m_list[None, :], index_float.transpose(0,1))
    batch_m = batch_m.view((-1, 1))
    x_m = x - batch_m

    output = torch.where(index, x_m, x)
    return F.cross_entropy(self.s*output, target, weight=self.weight)
    # scales all outputs by s=30
    # so equivalent to rescaling margin by factor of 30

class LogAdjLoss(nn.Module):

  def __init__(self, cls_num_list, weight=None):
    super(LogAdjLoss, self).__init__()
    m_list = cls_num_list / np.sum(cls_num_list)
    m_list = np.log(m_list)
    m_list = torch.cuda.FloatTensor(m_list)
    self.m_list = m_list
    self.weight = weight

  def forward(self, x, target):
    batch_m = self.m_list.repeat(x.shape[0],1)
    batch_m = torch.cuda.FloatTensor(batch_m)
    output = x + batch_m
    # add log(pi_j) for each logit f_j, so that cross-entropy of updated outputs
    # is just the logit adjusted loss

    return F.cross_entropy(output, target, weight=self.weight)
```

Code 5: Losses in losses.py, which are imported and used in training

The LDAM loss now uses a different hyperparameter C, which rescales all the margins by fixing the maximum margin from a class. The per-class and overall test accuracies used in the tables and figures were logged while training.

Bibliography

[1] Jonathon Byrd and Zachary Lipton. What is the effect of importance weighting in deep learning? In *International Conference on Machine Learning*, pages 872–881. PMLR, 2019.

[2] Kaidi Cao, Colin Wei, Adrien Gaidon, Nikos Arechiga, and Tengyu Ma. Learning imbalanced datasets with label-distribution-aware margin loss. *Advances in neural information processing systems*, 32, 2019.

[3] François Caron. Lecture notes on Statistical Machine Learning, Part II Supervised Learning, March 2021.

[4] Yin Cui, Menglin Jia, Tsung-Yi Lin, Yang Song, and Serge Belongie. Class-balanced loss based on effective number of samples. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 9268–9277, 2019.

[5] Youngkyu Hong, Seungju Han, Kwanghee Choi, Seokjun Seo, Beomsu Kim, and Buru Chang. Disentangling label distribution for long-tailed visual recognition. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 6626–6636, 2021.

[6] Bingyi Kang, Saining Xie, Marcus Rohrbach, Zhicheng Yan, Albert Gordo, Jiashi Feng, and Yannis Kalantidis. Decoupling representation and classifier for long-tailed recognition. *arXiv preprint arXiv:1910.09217*, 2019.

[7] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[8] Aditya Krishna Menon, Sadeep Jayasumana, Ankit Singh Rawat, Himanshu Jain, Andreas Veit, and Sanjiv Kumar. Long-tail learning via logit adjustment. *arXiv e-prints*, pages arXiv–2007, 2020.

[9] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.

[10] Jiawei Ren, Cunjun Yu, Xiao Ma, Haiyu Zhao, Shuai Yi, et al. Balanced meta-softmax for long-tailed visual recognition. *Advances in Neural Information Processing Systems*, 33:4175–4186, 2020.

[11] Yang Song and Serge Belongie. Introducing the iNaturalist 2018 Challenge, Mar 2018. Accessed: 23-04-2022.

[12] Hao Wang, Yitong Wang, Zheng Zhou, Xing Ji, Dihong Gong, Jingchao Zhou, Zhifeng Li, and Wei Liu. Cosface: Large margin cosine loss for deep face recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5265–5274, 2018.

[13] Han-Jia Ye, Hong-You Chen, De-Chuan Zhan, and Wei-Lun Chao. Identifying and compensating for feature deviation in imbalanced deep learning. *arXiv preprint arXiv:2001.01385*, 2020.

[14] Yifan Zhang, Bingyi Kang, Bryan Hooi, Shuicheng Yan, and Jiashi Feng. Deep long-tailed learning: A survey. *arXiv preprint arXiv:2110.04596*, 2021.