MixUp as Directional Adversarial Training

A Unifying Understanding of MixUp and Adversarial Training

by

Guillaume Perrault Archambault
Supervisor: Professor Yongyi Mao

Thesis submitted to the University of Ottawa
In partial fulfillment of the requirements
For the M.A.Sc. degree in
Electrical and Computer Engineering

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

This thesis aims to contribute to the field of neural networks by improving upon the performance of a state-of-the-art regularization scheme called MixUp, and by contributing to the conceptual understanding of MixUp. MixUp is a data augmentation scheme in which pairs of training samples and their corresponding labels are mixed using linear coefficients. Without label mixing, MixUp becomes a more conventional scheme: input samples are moved but their original labels are retained. Because samples are preferentially moved in the direction of other classes we refer to this method as directional adversarial training, or DAT. We show that under two mild conditions, MixUp asymptotically convergences to a subset of DAT. We define untied MixUp (UMixUp), a superset of MixUp wherein training labels are mixed with different linear coefficients to those of their corresponding samples. We show that under the same mild conditions, untied MixUp converges to the entire class of DAT schemes. Motivated by the understanding that UMixUp is both a generalization of MixUp and a scheme possessing adversarial-training properties, we experiment with different datasets and loss functions to show that UMixUp provides improves performance over MixUp. In short, we present a novel interpretation of MixUp as belonging to a class highly analogous to adversarial training, and on this basis we introduce a simple generalization which outperforms MixUp.

Acknowledgements

I would like to thank my thesis supervisor and mentor, professor Yongyi Mao, for his dedicated hard work, inspiring words and helpful advice. Without him I would have given up many times over. I'm eternally grateful to professor Mao for putting up with me and helping me accomplish my goals, not just in his role as a technical supervisor but also as a friend, and coach.

I would like to thank Ziqiao for his uplifting friendship and never-ending willingness to help. I would like to thank doctor Harry Guo for giving me the opportunity to work at the NRC during my degree. I would like to thank the team in Beihang University for their help and support; especially Fanshuang for her patience and kindness, professor Zhang for allowing me to use Beihang resources, and the other students for their support at all hours. I would also like to thank professor Mao's students for their friendship and support, especially Masoumeh and Chenjie.

Et surtout, je voudrais remercier ma famille. Ma soeur Mathilde qui m'a toujours soutenu et m'a même aidé a relire mon texte alors quelle a 4 enfants à ses soins, et mes parents qui me nourrissent de leur amour depuis que je suis tout petit et encore aujourd'hui. Je vous aime.

Dedication

This is dedicated to Jonathan Blanchette, my dear friend who went missing on February 6, 2020.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Outline

The target reader of this thesis is anyone possessing an undergraduate level of mathematics and statistics.

As mentioned in the abstract, this thesis aims to contribute to the field of neural networks by improving upon the performance of a state-of-the-art regularization scheme called MixUp. For those readers who do not possess a background in neural networks or machine learning regularization, chapter 2 aims to fill that gap. Machine learning algorithms and models are introduced in section 2.1. Alternative formulations of the optimization problem used to achieve a 'learned' model, specifically the ML, MAP, and ERM solutions, are introduced in sections 2.2 and 2.3, with an emphasis on supervised learning. The concept of overfitting is explained in section 2.4 and its countermeasure, regularization, is presented in section 2.5. Neural network models are described in section 2.6, and two algorithms commonly adopted to solve the optimization problem – gradient descent and backpropagation – are discussed in sections 2.7 and 2.8 respectively. Finally, the cross-validation algorithm, used to evaluate learned models and neural networks in particular, is presented in section 2.9.

Chapter 3 presents the main contributions of this thesis. While section 1.2 motivates and summarizes these contributions, we present the logical flow of chapter 3 hereafter. Section 3.1 lays out the classification setting to which the chapter's contributions apply, and describes the assumptions which underly our main work. The MixUp algorithm is reviewed in section 3.2, after which our own regularization schemes DAT and UMixUp are introduced. DAT is connected to adversarial training in section 3.3, and UMixUp is connected to DAT in section 3.4. Finally, our experiments and results are presented in section 3.5.

1.2 Literature, Motivation and Contributions

Deep learning applications often require complex networks with a large number of parameters [16, 46, 9]. Although neural networks perform so well that their ability to generalize is an area of study in itself [47, 3], their high complexity nevertheless causes them to overfit their training data [22]. For this reason, effective regularization techniques are in high demand.

There are two flavors of regularization: model confining and data augmentation[1]. Model confining methods constrain models to learning in a subset of parameter space which has a higher probability of generalizing well. Notable examples are weight decay [21] and dropout [39].

Data augmentation methods add transformed versions of training samples to the original training set. Conventionally, transformed samples retain their original label, so that models effectively see a larger set of data-label training pairs. Commonly applied transformations in image applications include flips, crops and rotations.

A recently devised family of augmentation schemes called adversarial training has attracted active research interest [41, 12, 30, 4, 37, 17]. Adversarial training seeks to reduce a model's propensity to misclassify minimally perturbed training samples, or adversarials. While attack algorithms used for testing model robustness may search for adversarials in unbounded regions of input space, adversarial training schemes generally focus on perturbing training samples within a bounded region, while retaining the sample's original label [13, 37].

Another recently proposed data augmentation scheme is MixUp [49], in which new samples are generated by mixing pairs of training samples using linear coefficients. Despite its well established generalization performance [49, 14, 45], the working mechanism of MixUp is not well understood. Guo et al. [14] suggest viewing MixUp as imposing local linearity on the model using points outside of the data manifold. While this perspective is insightful, we do not believe it paints a full picture of how MixUp operates. A recent study [23] provides empirical evidence that MixUp improves adversarial robustness, but does not present MixUp as a form of adversarial training.

We build a framework to understand MixUp in a broader context: we argue that adversarial training is a central working principle of MixUp. To support this contention, we connect MixUp to a MixUp-like scheme which does not perform label mixing, and we relate this scheme to adversarial training.

Without label mixing, MixUp becomes a conventional augmentation scheme: input samples are moved, but their original labels are retained. Because samples are moved in the direction of other samples – which are typically clustered in input space – we describe this method as 'directional'. Because this method primarily moves training samples in the direction of adversarial classes, this method is analogous to adversarial training. We thus refer to MixUp without label mixing as directional adversarial training (DAT). We show

[1] Some authors describe these flavors as "data independent" and "data-dependent" [14].

that MixUp converges to a subset of DAT under mild conditions, and we thereby argue that adversarial training is a working principle of MixUp.

Inspired by this new understanding of MixUp as a form of adversarial training, and upon realizing that MixUp is (asymptotically) a subset of DAT, we introduce Untied MixUp (UMixUp), a simple enhancement of MixUp which converges to the entire family of DAT schemes, as depicted in Figure 1.1. Untied Mixup mixes data-label training pairs in a similar way to MixUp, with the distinction that the label mixing ratio is an arbitrary function of the sample mixing ratio. We perform experiments to show that UMixUp's classification performance improves upon MixUp.

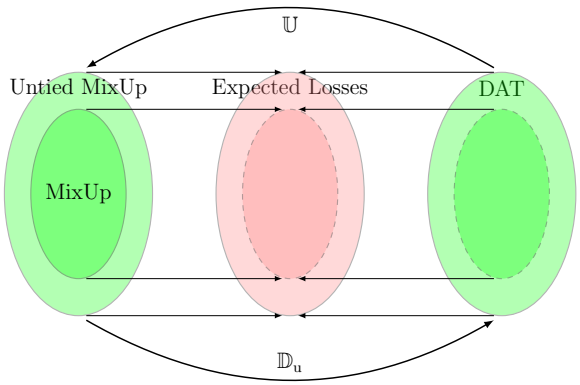


Figure 1.1: The relationship between MixUp, DAT and Untied MixUp.

In short, this research is motivated by a curiosity to better understand the working of MixUp. In-so-doing we make the following contributions:

1. Develop a generalization of MixUp called Untied MixUp (UMixUp). UMixUp is developed in section 3.2.3.

2. Establish empirically that UMixUp provides improved classification performance over MixUp. Experimental results are given in section 3.5.6.

3. Establish a new regularization scheme called Directional Adversarial Training (DAT), and explain its connection to adversarial training. DAT is developed in sections 3.2.2 and 3.3.

   (a) The contribution provided by DAT is a conceptual novelty, rather than a performance improvement. The purpose of establishing DAT is to provide insight into the working mechanism of MixUp and UMixUp.

4. Establish an equivalency between the expected loss of UMixUp and DAT. This analytical result is given in section 3.4.

   (a) Combined with contribution 3, we thus establish a connection between UMixUp and adversarial training.

(b) Putting contributions 1 and 4a together, we thus explain that adversarial training is a fundamental working mechanism of MixUp and that a larger set of schemes benefit from the same explanatory power.

5. Improve upon the published implementation of MixUp. These improvements are discussed in section 3.5.4.

6. Develop a simulation toolkit which allows management of a high volume of parallel jobs on high-performance clusters. This simulation toolkit is discussed in section 3.5.5.

7. Establish a new loss function called negative cosine loss and show that MixUp is applicable to this new loss function.

   (a) Conventionally, MixUp is only applicable to baseline models that use cross entropy loss. All analytical results we develop in this thesis are applicable to a wider family of models using any loss function which we term target-linear. We define target-linearity and introduce negative cosine loss in section 3.1 and experiment with negative cosine-loss to show its potential in section 3.5.6.

1.3 Notation

Column vectors are denoted by bold letters such as $\mathbf{m}$, and sets are denoted by blackboard typefaced uppercase characters such as $\mathbb{M}$. The component of a vector is denoted by a bracketed index. For example, $\mathbf{m}[i]$ denotes the ith component of $\mathbf{m}$. Matrices are denoted by bold, uppercase, and overlined letters such as $\overline{\mathbf{M}}$.

Elements of finite sample spaces are identified using a superscript index such as $m^j \in \mathbb{M}$. Any sequence, $(a_1, a_2, \ldots, a_n)$ are denoted by a_1^n. Likewise $(A_1, A_2, \ldots, A_n)$ is denoted by A_1^n, and a sequence of sample pairs $((\mathbf{x}_1, \mathbf{x}'_1), (\mathbf{x}_2, \mathbf{x}'_2), \ldots, (\mathbf{x}_n, \mathbf{x}'_n))$ denoted by $(\mathbf{x}, \mathbf{x}')_1^n$.

Due to typesetting limitations, certain notations will be overloaded, with the meaning specified in the appropriate context. Regular (non-calligraphic) capitalized letters such as M may denote a random variable, and their lowercase counterparts, e.g., m, denote realizations of a random variable. Alternatively, regular capitalized letters may denote a distribution, and lowercase letters may denote functions. Certain important sequences are denoted by calligraphic uppercase letters such as $\mathcal{M}$. Certain important functions may also be denoted by calligraphic uppercase letters.

For any value $a \in [0, 1]$, we use $\bar{a}$ as a shorthand notation for $1 - a$. Let $\mathbb{1}\{\text{event}\}$ represent an indicator function that returns 1 if an event happened and 0 otherwise.

Chapter 2

Machine Learning, Regularization, and Deep Learning Training

2.1 Machine Learning

Machine learning describes any algorithm which is capable of learning a task. An algorithm is said to "learn" if its ability to perform the task improves with increased exposure to experience, otherwise known as training. The algorithm's task completion ability, in turn, is quantified using some performance measure. Or as stated more formally by [29], "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E."

2.1.1 Types of machine learning algorithms

Different types of machine learning algorithms are appropriate for different learning activities (or tasks).

Supervised learning algorithms train over input-output pairs in order to predict the output of unseen input samples [36]. Formally, the "experience" of a supervised learning algorithm is given as a training dataset[1],

$$\mathcal{S} = s_1^n := \big((\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), ...(\mathbf{x}_n, y_n)\big) \tag{2.1}$$

where each input-output pair $s_i = (\mathbf{x}_i, y_i)$ is generated from an i.i.d. process. We assume that training samples s_i are realizations of some random variable S, input samples $\mathbf{x}_i$ are realizations of a random variable $\mathbf{X}$, and output samples y_i are realizations of a random variable Y. For convenience we define the training input sequence

$$\mathcal{X} = \mathbf{x}_1^n := (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n) \tag{2.2}$$

[1] Although called a 'dataset', it is in fact a sequence

and the training output sequence

$$\mathcal{Y} = y_1^n := (y_1, y_2, ..., y_n) \tag{2.3}$$

The goal of supervised learning is to develop an algorithm that either predicts the random variable Y from X, or estimates the conditional probability $p(Y|X)$. Our supervised learning model, therefore, can be given either as the prediction

$$y := F(\mathbf{x}; \theta) \tag{2.4}$$

or as an estimate of the conditional probability

$$p(y|x; \theta) := F(y, x; \theta) \tag{2.5}$$

where θ is the set of model parameters.

Supervised learning algorithms may be classed by the type of the variable Y. If Y is a numerical variable, the supervised algorithm is referred to as a numerical regression (or sometimes simply as a 'regression'). If Y is a categorical variable, the algorithm is known as a classification algorithm[6, p.3]. For example, predicting the price of a stock market asset is a regression problem, while determining the prevalence of drug use amongst at-risk subpopulations is a (binary) classification problem. In section 3.5 we describe another example of classification: assigning the correct label to images in 10-class datasets.

Unsupervised learning algorithms train over unpaired samples in order to learn the structure of any sample picked from the same underlying distribution as the training samples[6, p.3]. Formally, given a training dataset,

$$\mathcal{S} := \{y_i : i = 1, 2, ..., n\} \tag{2.6}$$

where samples y_i are again realizations of a random variable Y. The goal of supervised learning is to develop an algorithm that either generates samples of Y or estimates the probability distribution of Y. Our unsupervised learning model, therefore, can be given either as the generator

$$y := F(\theta) \tag{2.7}$$

or as the density estimate

$$p(y; \theta) := F(y, \theta) \tag{2.8}$$

where θ is the set of model parameters. For both supervised and unsupervised learning algorithms, the learning goal is

$$\theta^* = \underset{\theta}{\operatorname{argmax}}\, p(\theta|\mathcal{S}) \tag{2.9}$$

In other words, we seek to find the set of model parameters θ which are the most likely given the dataset $\mathcal{S}$.

Unsupervised learning may be used for cluster analysis, density estimation, or generation of realistic samples. For example, finding the factors that influence drug use is a cluster analysis problem, while estimation the propability of rainfall in different locations along a river is a density estimation problem.

In reinforcement learning, a learned agent modifies an environment using a limited set of actions[40]. Transitions between environment states yield a reward (or penalty), which the agent learns to maximize (or minimize). The agent's training objective is to learn a state-dependent strategy (or policy) which optimizes reward. Formally, let $\mathbb{S}$ be the set of environment states, and let $\mathbb{A}$ be the set of agent actions. Most machine learning problems are characterized by

$$p_a(s, s') = p(s_{t+1} = s' | s_t = s, a_t = a) r_a(s, s') \tag{2.10}$$

where $p_a(s, s')$ is the probability of transitioning from state s to s' if the agent takes action a, and $r_a(s, s')$ is the reward associated with said transition. Because the state s only depends on the previous state and action, and does not depend on past states and actions, this is referred to as a Markov Decision Process. The functions $p_a(s, s')$ and $r_a(s, s')$ may both be learned models. Optimization of reinforcement problems is a fascinating topic but is beyond the scope of this thesis. Examples of reinforcement learning include chess-playing neural networks and self-driving software.

This thesis focuses on supervised training algorithms, and future references to learning algorithms will refer to supervised learning algorithms, unless otherwise specified.

2.1.2 Machine Learning Model

We define "model" as the algorithm that performs the desired task, consistent with most authors in the field of neural networks [3, 4, 49, 46, 45, 47, 37]. The model contrasts with the algorithm used to optimize the task-performing model, which we hereafter refer to as the "optimizer".

There are two further flavours of learning models: parametric and non-parametric[6, p.68]. Parametric models have a fixed number of parameters (or weights), independent of the size of the training dataset. This conventionally means that model designers finely tune parametric models to ensure an adequate degree of complexity for the dataset (see overfitting/generalization section).

Non-parametric models have a number of parameters which generally increases with sample size. For example, K-nearest neighbor algorithms typically compute the (sometimes weighted) mean of the K-samples nearest to the input sample[2]. A non-weighted k-NN algorithm computes

$$F(\mathbf{x}) = \frac{1}{k} \sum_{m \in N(\mathbf{x})} y_m \tag{2.11}$$

where $N(\mathbf{x})$ is the set of k nearest neighbors of $\mathbf{x}$. Formally,

$$N(\mathbf{x}) = \left\{ i : \left(\sum_{j=1}^{N} \mathbb{1}\{||\mathbf{x} - \mathbf{x}_i|| \geq ||\mathbf{x} - \mathbf{x}_j||\} \right) \leq k, i \in \{1, 2, \ldots, n\} \right\} \tag{2.12}$$

This means that the k-NN algorithm's set of parameters is the set of all training input samples, as all input samples $\mathbf{x}_j$ are searched for any given input $\mathbf{x}$. Since the parameters of non-parametric models are typically the data samples themselves, they are generally viewed as not having parameters (hence "non-parametric").

This thesis focuses on neural network models, which are parametric. Thus the term "model" will hereafter refer to parametric models.

For any given model, we denote the set of all parameters as the column vector θ, and we characterize our model as the function $F(\theta)$. We can view our model as a family of hypotheses $\mathbb{H}$, where the function F maps a specific assignment of θ to a hypothesis. For this reason, F is sometimes referred to as the hypothesis function[20]. Figure 2.1 shows the mapping of the parameter space Θ to the hypothesis space $\mathbb{H}$.

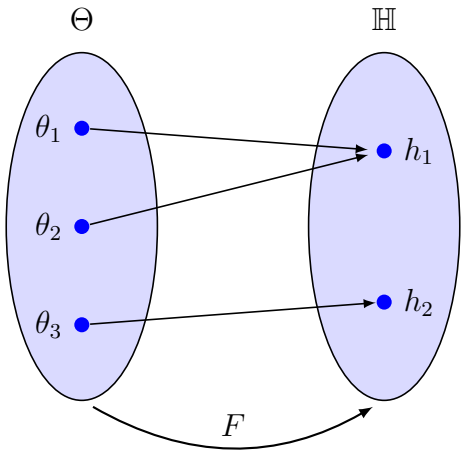


Figure 2.1: Domain (Θ) and range ($\mathbb{H}$) of hypothesis function F

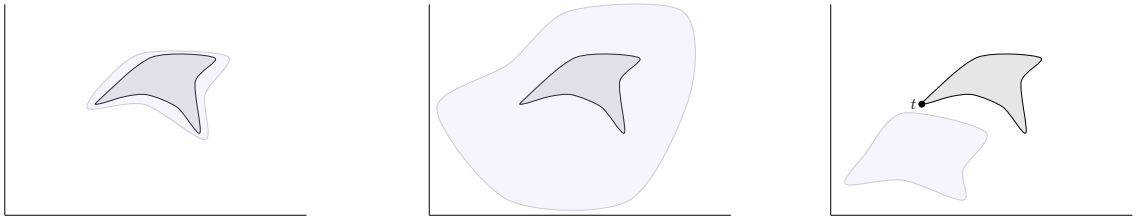
Equivalently, the range of $F(\theta)$ represents the hypothesis space $\mathbb{H}$. The goal of machine learning is to discover a hypothesis $h \in \mathbb{H}$ which adequately explains the data. Since the model state θ fully characterizes the hypothesis function, we do not distinguish between optimizing F and optimizing θ. With slight abuse of language, we will sometimes refer to θ as the hypothesis.

2.1.3 Selecting a Learning Model and Optimization Algorithm

Machine learning models have various architectures making them suitable for a myriad of different applications. All parametric models have in common that they can be represented as a function F(θ) parameterized by a set of model parameters, θ.

Ideally, the function $F(\theta)$ is chosen such that it is expressive, generalizes well, and is trainable [34]. We elaborate these factors hereafter:

1. Expressivity. A member of the hypothesis space $\mathbb{H}$ ideally performs the desired class of tasks adequately well. We refer to a 'perfect' or 'correct' task-performing function as a ground truth function. Let $\mathbb{G}$ be the set of ground-truth functions. The expressivity criterion is then a measure of the model's ability to express at least one member of $\mathbb{G}$. While different machine learning models may be suitable for different tasks, we note that neural networksmay, in theory, represent any computable function[2].



(a) Tightly expressive model (b) Loosely expressive model (c) Non-expressive model

Figure 2.2: Expressivity of the learning model

Figure 2.2 is an abstract illustration of three different scenarios of model expressivity. The gray region (□) represents the ground truth region $\mathbb{G}$, the blue region (□) represents the hypothesis space $\mathbb{H}$, and the coordinate system spans the space of all computable functions. Figure 2.2a) shows a 'tightly-expressive' model, whose hypothesis space narrowly encompasses the ground-truth region (or at least one member of the latter). Figure 2.2b) shows a 'loosely-expressive' model for which the ground-truth function is only a small subset of the hypothesis space. Figure 2.2c) shows a non-expressive model, whose hypothesis space does not contain any member of the ground-truth region.

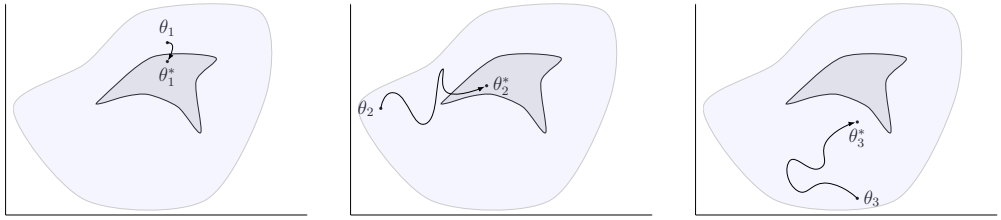
Model expressivity contrasts with model 'complexity', which refers to the size of the hypothesis space: a complex model has a larger hypothesis space than a less complex model. Higher complexity often results in better expressivity; suppose, for example, that the boundaries of $\mathbb{H}$ in figure 2.2c were stretched outwardly – it is likely that the closest member of $\mathbb{G}$ (represented by the point t) would become expressed by the model. However, high model complexity does not guarantee expressivity: one can imagine a variant of figure 2.2c in which $\mathbb{H}$ is large and does not overlap $\mathbb{G}$. Model complexity will be further discussed in section 2.4.

2. Generalization. While the model's hypothesis space should contain members capable of closely mimicking the ground truth function, the optimizing algorithm has limited experience over which to guide the model to a suitable hypothesis. Ideally, the optimization algorithm should be able to converge the model 'quickly' (over limited

[2]Formally, this is referred to as the universal approximation theory, which is discussed briefly in section 2.6)

training experience) toward a region of hypothesis space that is in or close to the ground truth function region. When a model does not generalize, we say that it overfits the training data. Overfitting, and by extension generalization, which will be further discussed in section 2.4.

3. Trainability. There is ideally an effective and efficient means of finding an adequate hypothesis. This means an optimization algorithm that is computationally efficient (in memory, compute power), and can achieve an adequate hypothesis. Depending on the initial value of θ and the path traveled during optimization, different optimization algorithms may use the same training experience to arrive at different hypotheses. This is illustrated in figure 2.3, which uses the same color coding as figure 2.2. Figures 2.3a and 2.3b show different optimization paths which result in successful generalization to $\mathbb{G}$. Figure 2.3c shows a failed optimization resulting in a model outside $\mathbb{G}$; otherwise known as an overfit model.



(a) Short path (b) Long path (c) Suboptimal path

Figure 2.3: Different θ paths for the same optimization problem.

2.2 Finding the Most Likely Hypothesis (MAP and ML Formulations)

In the context of parameterized learning models, the goal of machine learning is to 'teach' a model $F(\theta)$ to perform a class of tasks. Given that our model is exposed to limited experience, we seek to find a the set of model parameters are most likely to explain the observed experience. This approach is sometimes referred to as the *maximum a posteriori* (MAP) principle[11, Thm 2.4.3]. Suppose we have training dataset $\mathcal{S}$ and model parameters θ. The most likely set of parameters, θ^{MAP}, is

$$\begin{aligned}
\theta^{MAP} &= \underset{\theta}{\operatorname{argmax}}\, p(\theta|\mathcal{S}) \\
&= \underset{\theta}{\operatorname{argmax}}\, \log p(\theta|\mathcal{S}) \\
&= \underset{\theta}{\operatorname{argmax}}\, \log \frac{p(\mathcal{S}|\theta)p(\theta)}{p(\mathcal{S})} \\
&= \underset{\theta}{\operatorname{argmin}} \Big(-\log p(\mathcal{S}|\theta) - \log p(\theta) \Big)
\end{aligned}$$

Note that if we have no prior knowledge about the distribution of the model parameters θ, a naïve assumption is that all model parameters are equi-probable, thus making $p(\theta)$ constant. Let us refer to the hypothesis obtained from such an assumption as θ_u^{MAP} (where u stands for uniform distribution). Since $\log p(\theta)$ is then constant for all θ, the maximum *a posteriori* estimate reduces to

$$\theta_u^{MAP} = \underset{\theta}{\operatorname{argmin}} \Big(- \log p(\mathcal{S}|\theta) \Big)$$

The latter optimization is the same as the maximum data likelihood (ML) optimization[31], in which we maximize the likelihood of observing the dataset $\mathcal{S}$ given our model $F(\theta)$:

$$\begin{aligned} \theta^{ML} &= \underset{\theta}{\operatorname{argmax}}\, p(\mathcal{S}|\theta) \\ &= \underset{\theta}{\operatorname{argmin}} \Big(- \log p(\mathcal{S}|\theta) \Big) = \theta_u^{MAP} \end{aligned}$$

Therefore, maximum likelihood optimization is a special case of maximum *a posteriori* optimization, where we assume that all ML hypotheses are equiprobable. This is uncommonly an adequate assumption, however, since in practical settings it is usually possible to make basic assumptions about which subsets of hypothesis space are more likely to be correct than others. In other words, a distribution more apt than the uniform distribution can be assumed for $p(\theta)$.

In summary, the general form of the MAP and ML solutions are

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \Big(- \log p(\mathcal{S}|\theta) - \log p(\theta) \Big) \tag{2.13}$$

$$\theta^{ML} = \theta_u^{MAP} = \underset{\theta}{\operatorname{argmin}} \Big(- \log p(\mathcal{S}|\theta) \Big) \tag{2.14}$$

2.2.1 Unsupervised Setting

In an unsupervised model, θ fully characterizes $F(\theta)$, which either models the system which generates the samples $\mathcal{S}$ (if $Y = F(\theta)$) or models the distribution that the samples $\mathcal{S}$ have been picked from ($p(Y;\theta) = F(\theta)$). In either case, given the hypothesis θ, the probability density of any particular sample $p(s_i)$ depends only θ, and not on other samples. Thus the samples s_i are conditionally independent given θ.

Since $\mathcal{S} = \{s_1, s_2, s_3, ..., s_n\}$, we may write equation 2.13 as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \Big(- \log \prod_i^n p(s_i|\theta) - \log p(\theta) \Big) \tag{2.15}$$

$$= \underset{\theta}{\operatorname{argmin}} \Big(- \sum_{i=1}^{n} \log p(s_i|\theta) - \log p(\theta) \Big) \tag{2.16}$$

and we write equation 2.14 as

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(-\log \prod_{i}^{n} p(s_i|\theta) \right) = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(s_i|\theta) \right) \tag{2.17}$$

2.2.2 General Supervised Learning Solution

Similarly in a supervised model, $\mathbf{x}$ and θ fully characterize $F(\mathbf{x};\theta)$, which either models the system which generates the training output samples $\mathcal{Y}$ (if $y = F(\mathbf{x};\theta)$) or models the conditional distribution of the labels $p(y|\mathbf{x})$ (if $p(y;\theta) = F(\theta)$). In either case, given the hypothesis θ and an input sample $\mathbf{x}_i$, the probability density of any particular output sample $p(y_i)$ depends only θ and $\mathbf{x}_i$, and not on other output samples. Thus the samples y_i are conditionally independent given $\theta \cap \mathbf{x}_i$. Moreover, given the additional occurrence of other input samples $\mathbf{x}_j$, where $j \neq i$, knowledge of other output samples y_j, where $j \neq i$, still provides no information on the likelihood of y_i. Thus the samples y_i are conditionally independent given $\theta \cap \mathcal{X}$.

Therefore in a supervised setting, we may write equation 2.13 as

$$\begin{aligned}
\theta^{MAP} &= \underset{\theta}{\operatorname{argmin}} \left(-\log p(\mathcal{S}|\theta) - \log p(\theta) \right) && (2.18)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log p(\mathcal{X},\mathcal{Y}|\theta) - \log p(\theta) \right) && (2.19)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log \left(p(\mathcal{Y}|\mathcal{X},\theta) p(\mathcal{X}|\theta) \right) - \log p(\theta) \right) && (2.20)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log \left(p(\mathcal{Y}|\mathcal{X},\theta) p(\mathcal{X}) \right) - \log p(\theta) \right) && (2.21)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log p(\mathcal{Y}|\mathcal{X},\theta) - \log p(\mathcal{X}) - \log p(\theta) \right) && (2.22)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log p(\mathcal{Y}|\mathcal{X},\theta) - \log p(\theta) \right) && (2.23)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log \prod_{i}^{n} p(y_i|\mathcal{X},\theta) - \log p(\theta) \right) && (2.24)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\log \prod_{i}^{n} p(y_i|\mathbf{x}_i,\theta) - \log p(\theta) \right) && (2.25)\\
&= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i,\theta) - \log p(\theta) \right) && (2.26)
\end{aligned}$$

Since our model $F(\mathbf{X};\theta)$ predicts Y *given* $\mathbf{X}$, the probability distribution of $\mathbf{X}$ is independent of θ; thus $p(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n|\theta) = p(\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_n)$, which is used to go from 2.20

to 2.21. To go from 2.22 to 2.23, we note that $\log p(\mathcal{X})$ is a constant relative to θ. We use the previously explained $y_i \perp\!\!\!\perp \mathcal{X}$ to go from 2.23 to 2.24. Finally we use the fact that y_i does not depend on $\mathbf{x}_j$ (where $j \neq i$) to go from 2.24 to 2.25.

For clarity, we explicity write the MAP solution in terms of our model $F(\mathbf{x}; \theta)$:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log F(y_i, \mathbf{x}_i; \theta) - \log p(\theta) \right) \tag{2.27}$$

And using the same steps starting from equation 2.14, the supervised learning ML estimate becomes

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log F(y_i, \mathbf{x}_i; \theta) \right) \tag{2.28}$$

2.2.3 Supervised Learning Solution as a Special Case of Unsupervised Learning Solution

We show that the supervised learning solutions (equations 2.26 and 2.28) are a special case of the unsupervised learning solutions (equations 2.16 and 2.17).

Starting from equation 2.26:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i, \theta) - \log p(\theta) \right) \tag{2.29}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i, \theta) - \sum_{i=1}^{n} \log p(\mathbf{x}_i) - \log p(\theta) \right) \tag{2.30}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \left[\log p(y_i|\mathbf{x}_i, \theta) + \log p(\mathbf{x}_i) \right] - \log p(\theta) \right) \tag{2.31}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log \left[p(y_i|\mathbf{x}_i, \theta) p(\mathbf{x}_i) \right] - \log p(\theta) \right) \tag{2.32}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log \left[p(y_i|\mathbf{x}_i, \theta) p(\mathbf{x}_i|\theta) \right] - \log p(\theta) \right) \tag{2.33}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p((\mathbf{x}_i, y_i)|\theta) - \log p(\theta) \right) \tag{2.34}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(s_i|\theta) - \log p(\theta) \right) \tag{2.35}$$

which is the same *maximum a posteriori* solution as the unsupervised setting (equation 2.16). Following the steps but starting instead from equation 2.28 (in other words, just

removing the $-\log p(\theta)$ term) yields

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(s_i|\theta) \right) \tag{2.36}$$

which is the same maximum likelihood solution as the unsupervised setting (equation 2.17).

2.2.4 Supervised Learning Solution for Numerical Regression

In a conventional numerical regression, the output samples are treated as observations containing a measurement error added to the 'true' output[19]. Since our model attempts to predict the true output, we assume that the output samples are normally distributed around the prediction:

$$Y = F(\mathbf{X}, \theta) + Z \tag{2.37}$$

where

$$Z \sim \mathcal{N}(0,\, \sigma^2) \tag{2.38}$$

Then $Y \sim \mathcal{N}(F(\mathbf{X}, \theta),\, \sigma^2)$, and

$$p(y_i|\theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-(y - F(\mathbf{X};\theta)^2)}{2\sigma^2}\right] \tag{2.39}$$

$$p(y_i|\mathbf{x}_i, \theta) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left[\frac{-(y - F(\mathbf{x}_i;\theta)^2)}{2\sigma^2}\right] \tag{2.40}$$

Plugging equation 2.40 into equation 2.26,

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log p(y_i|\mathbf{x}_i, \theta) - \log p(\theta) \right) \tag{2.41}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \left(\log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right] + \frac{-(y_i - F(\mathbf{x}_i;\theta)^2)}{2\sigma^2} \right) - \log p(\theta) \right) \tag{2.42}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right] - \sum_{i=1}^{n} \frac{-(y_i - F(\mathbf{x}_i;\theta)^2)}{2\sigma^2} - \log p(\theta) \right) \tag{2.43}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} \frac{(y_i - F(\mathbf{x}_i;\theta)^2)}{2\sigma^2} - \log p(\theta) \right) \tag{2.44}$$

$$= \underset{\theta}{\operatorname{argmin}} \frac{1}{2\sigma^2} \left(\sum_{i=1}^{n} (y_i - F(\mathbf{x}_i;\theta)^2) - 2\sigma^2 \log p(\theta) \right) \tag{2.45}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} (y_i - F(\mathbf{x}_i;\theta)^2) - 2\sigma^2 \log p(\theta) \right) \tag{2.46}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} (y_i - F(\mathbf{x}_i;\theta)^2) - \phi \log p(\theta) \right) \tag{2.47}$$

where $\phi = 2\sigma^2$, thus $\phi \geq 0$.

Using the same steps, starting from equation 2.36

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} (y_i - F(\mathbf{x}_i; \theta)^2) \right) \tag{2.48}$$

Note that the postulate described by equations 2.37 and 2.38 is equivalent to

$$F(\mathbf{X}, \theta) = Y + Z \tag{2.49}$$

where

$$Z \sim \mathcal{N}(0,\, \sigma^2) \tag{2.50}$$

such that $F(\mathbf{X}, \theta) \sim \mathcal{N}(Y,\, \sigma^2)$.

This reformulation makes it clear that we are constraining our model to fit a normal distribution around the output samples. This constraint on the hypothesis space is a form of regularization (namely, model confining).

2.2.5 Supervised Learning Solution for Classification

Let $\mathbb{X}$ be a vector space in which the input samples $\mathbf{x}_i$ live (a.k.a. the support of X) and let $\mathbb{Y} := \{y^{(j)} : j = 1, 2, ..., m\}$ be the set of all possible labels paired with the input samples (a.k.a. the support of Y). The boldfaced $\mathbf{F}(\mathbf{x}_i; \theta)$ will represent a probability vector such that $\mathbf{F}(\mathbf{x}; \theta)[j] = F(y^{(j)}, \mathbf{x}; \theta)$.

For convenience in the context of cross-entropy, let $p_{\mathbf{x}_i}$ denote the conditional empirical probability function $p(Y|\mathbf{x}_i)$ and $q_{\mathbf{x}_i}$ denote the estimated distribution $p(Y|\mathbf{x}_i, \theta)$. Since a conditional probability distribution is itself a probability distribution, $p_{\mathbf{x}_i}$ and $q_{\mathbf{x}_i}$ are probability distributions. The boldfaced $\mathbf{p}_{\mathbf{x}_i}$ and $\mathbf{q}_{\mathbf{x}_i}$ will represent probability vectors, such that $\mathbf{p}_{\mathbf{x}_i}[j] = p_{\mathbf{x}_i}(y^{(j)})$ and $\mathbf{q}_{\mathbf{x}_i}[j] = q_{\mathbf{x}_i}(y^{(j)})$.

We also define the target distribution $t_i(y^{(j)})$, which is the empirical probability that label y_j was generated in the i^{th} term of sequence $\mathcal{S}$. This is in sharp contrast to $p(y^{(j)}|\mathbf{x}_i)$, which is the empirical probability that label y_j was generated given $X = \mathbf{x}_i$, which can occur in multiple terms of $\mathcal{S}$. The boldfaced $\mathbf{t}_i$ will represent a probability vector such that $\mathbf{t}_i[j] = t_i(y^{(j)})$.

We denote by $H(\mathbf{p}, \mathbf{q})$ the cross-entropy between two distributions p and q, and $D_{KL}(\mathbf{p}||\mathbf{q})$ the Kullback-Leibler divergence. If p and q are discrete distributions,

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{z \in \mathbb{Z}} \mathbf{p}[z]) \log \mathbf{q}[z] \tag{2.51}$$

$$D_{KL}(\mathbf{p}||\mathbf{q}) = \sum_{z \in \mathbb{Z}} \mathbf{p}[z] \log \left(\frac{\mathbf{p}[z])}{\mathbf{q}[z])} \right) = H(\mathbf{p}, \mathbf{q}) - H(\mathbf{p}, \mathbf{p}) \tag{2.52}$$

where p and q share the same support $\mathbb{Z}$

In a classification context, the output samples y_i may either be class labels or probability vectors assigning probabilities to each class. We develop supervised learning solutions for both cases.

Case: output samples are class labels.

Let us first consider the case were y_i is a class label. We have

$$t_i(y^{(j)}) = \mathbb{1}\{y^{(j)} = y_i\} \tag{2.53}$$

and

$$p(y^{(j)}|\mathbf{x}_i, \theta) = \frac{1}{\ell_i} \sum_{k=1}^{n} \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\} t_i(y_i) \tag{2.54}$$

where

$$\ell_i = \sum_{k=1}^{n} \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\} \tag{2.55}$$

Lemma 1

$$-\sum_{i=1}^{n} H(\boldsymbol{p}_{\boldsymbol{x}_i}, \boldsymbol{q}_{\boldsymbol{x}_i}) = \sum_{i=1}^{n} \log p(y_i|\boldsymbol{x}_i, \theta) \tag{2.56}$$

What follows is a proof of lemma 1 using the simplifying assumption that the training input dataset $\mathcal{X}$ is strictly a set (contains no duplicates)[3]. This assumption is not required, and we prove the general case in appendix A.1.

Given our assumption that $\mathcal{X}$ contains no duplicates, the empirical probability of a label $y^{(j)}$ given $\mathbf{x}_i$ can be given as

$$p(y^{(j)}|\mathbf{x}_i) = \mathbb{1}\{y^{(j)} = y_i\} \tag{2.57}$$

That is, the empirical probability (or dataset frequency) of input sample $X = \mathbf{x}_i$ being labeled $Y = y_i$ is 1, and the empirical probability of $\mathbf{x}_i$ being labeled anything else is zero.

We can expand equation 2.57 as follows

$$\sum_{j=1}^{m} p(y^{(j)}|\mathbf{x}_i) \log p(y^{(j)}|\mathbf{x}_i, \theta) = \log p(y_i|\mathbf{x}_i, \theta) \tag{2.58}$$

Since $p_{\mathbf{x}_i}$ and $q_{\mathbf{x}_i}$ share the support $\mathbb{Y}$ we may write the left-hand side of equation 2.58 as the cross-entropy between $p_{\mathbf{x}_i}$ and $q_{\mathbf{x}_i}$:

$$-H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) = \log p(y_i|\mathbf{x}_i, \theta) \tag{2.59}$$

Summing both sides over $i = 1, 2, ..., n$ completes the simplified proof of lemma 1.

[3]This assumption holds in most datasets studied in academic contexts, as input samples are hand-labeled and therefore unlikely to be assigned a label multiple times.

Lemma 2

$$-\sum_{i=1}^{n} H(\boldsymbol{t}_i, \boldsymbol{q}_{\boldsymbol{x}_i}) = \sum_{i=1}^{n} \log p(y_i|\boldsymbol{x}_i, \theta) \tag{2.60}$$

The proof of lemma 2 is almost identical to the simplified proof of lemma 1, replacing $p(y^{(j)}|\mathbf{x}_i)$ with $t_i(y^{(j)})$ starting from equation 2.57.

Using lemma 1 we may now write the *maximum a posteriori* solution (equation 2.26) as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) - \log p(\theta) \right) \tag{2.61}$$

Using lemma 2 we may also write equation 2.26 as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{t}_i, \mathbf{q}_{\mathbf{x}_i}) - \log p(\theta) \right) \tag{2.62}$$

Similarly, using lemma 1, the maximum likelihood solution (equation 2.36) can be written as

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) \right) \tag{2.63}$$

and using lemma 2, equation 2.36 can also be written as

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{t}_i, \mathbf{q}_{\mathbf{x}_i}) \right) \tag{2.64}$$

Thus, the data likelihood maximization problem is equivalent to cross-entropy minimization.

Case: output samples are probability vectors

We now discuss the case that output samples $\mathbf{y}_i$ represent probability vectors. We slightly redefine the R.V. Y as the class label paired with X[4].

Since $\mathbf{y}_i$ is a probability vector, the empirical distribution t_i is given by

$$t_i(y^{(j)}) = \mathbf{y}_i[j] \tag{2.65}$$

and the relationship between $p(y^{(j)}|\mathbf{x}_i, \theta)$ and $\mathbf{y}_i[j]$ is given by A.15. The boldfaced $\mathbf{t}_i$ will represent a probability vector such that $\mathbf{t}_i[j] = t_i(y^{(j)})$.

[4]Since y_i is a probability vector, our original definition of Y as the sample space of output samples y_i would have implied that Y is a random distribution of labels, not the label itself.

Let us again begin by using the simplifying assumption that the training input dataset $\mathcal{X}$ contains no duplicates. Thus the output samples $\mathbf{y}_i$ represent the empirical probability distribution $p(Y|\mathbf{x}_i)$. Formally,

$$p(y^{(j)}|\mathbf{x}_i) = p_{\mathbf{x}_i}(y^{(j)}) = \mathbf{y}_i[j] \tag{2.66}$$

In other words, we assume the probability that the label $y^{(j)}$ occurs given $X = \mathbf{x}_i$ is $\mathbf{y}_i[j]$. Since $\mathbf{y}_i$ represents an empirical distribution we may view $\mathbf{y}_i[j]$ as the frequency of label $y^{(j)}$ given input sample $\mathbf{x}_i$. We view each term of $\mathcal{S}$ as the result of a label-generating experiment. Let r_i be the number of labels generated in the i^{th} term of sequence $\mathcal{S}$. Hence output sample $\mathbf{y}_i$ means label y_j was generated $r_i \cdot \mathbf{y}_i[j]$ times in experiment i of sequence $\mathcal{S}$. The likelihood of $\mathbf{y}_i$ can therefore be seen as the likelihood that label y^1 occurred with frequency $\mathbf{y}_i[1]$, and y^2 occurred with frequency $\mathbf{y}_i[2]$, ..., and y^m occurred with frequency $\mathbf{y}_i[m]$. Given our hypothesis $p(y^{(j)}|\mathbf{x}_i, \theta) = q_{\mathbf{x}_i}(y^{(j)})$, the likelihood of this set of outcomes is $\prod_{j=1}^{m} q_{\mathbf{x}_i}(y^{(j)})^{r_i \cdot \mathbf{y}_i[j]}$ and thus the log likelihood of $\mathbf{y}_i$ is

$$\log p(\mathbf{y}_i|\mathbf{x}_i, \theta) = \log \prod_{j=1}^{m} q_{\mathbf{x}_i}(y^{(j)})^{r_i \cdot \mathbf{y}_i[j]} \tag{2.67}$$

$$= \log \prod_{j=1}^{m} q_{\mathbf{x}_i}(y^{(j)})^{r_i \cdot p_{\mathbf{x}_i}(y^{(j)})} \tag{2.68}$$

$$= \sum_{j=1}^{m} \log \left[q_{\mathbf{x}_i}(y^{(j)})^{r_i \cdot p_{\mathbf{x}_i}(y^{(j)})} \right] \tag{2.69}$$

$$= \sum_{j=1}^{m} r_i \cdot p_{\mathbf{x}_i}(y^{(j)}) \log q_{\mathbf{x}_i}(y^{(j)}) \tag{2.70}$$

$$= -r_i \cdot H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) \tag{2.71}$$

If we assume that each sample $\mathbf{y}_i$ is equally reliable, then r_i is constant with respect to i. In other words, the label-generating experiments are assumed to be of the same size for each term i in $\mathcal{S}$. Let $r = r_i$ be the fixed experiment size under this assumption.

We may then write equation 2.26 as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(r \sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) - \log p(\theta) \right) \tag{2.72}$$

$$= \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) - \kappa \log p(\theta) \right) \tag{2.73}$$

where $\kappa = \frac{1}{r}$ and thus $\kappa \in [0; 1]$

r can also be seen as a degree of confidence in the soft labels $\mathbf{y}_i$.

Equation 2.36 can be written as

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) \right) \tag{2.74}$$

The general MAP and ML solutions may also be written as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{t}_i, \mathbf{q}_{\mathbf{x}_i}) - \kappa \log p(\theta) \right) \tag{2.75}$$

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{t}_i, \mathbf{q}_{\mathbf{x}_i}) \right) \tag{2.76}$$

Equations 2.75 and 2.76 are easily proved by replacing $p_{\mathbf{x}_i}$ with t_i starting from equation 2.66.

We show in appendix A.2 that equations 2.73 and 2.74 hold true in general (when $\mathcal{X}$ contains duplicates), and that the following are the general MAP and ML solutions when output samples $\mathbf{y}_i$ represent empirical distributions of labels:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} m_i \cdot H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) - \log p(\theta) \right) \tag{2.77}$$

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} m_i \cdot H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) \right) \tag{2.78}$$

where

$$m_i = \frac{\sum_{k=1}^{n} \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\}}{\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\}} \tag{2.79}$$

2.2.6 Summary of supervised and unsupervised solutions

Both the supervised and unsupervised solutions may be expressed as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log F(s_i; \theta) - \log p(\theta) \right) \tag{2.80}$$

$$\theta^{ML} = \theta_u^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log F(s_i; \theta) \right) \tag{2.81}$$

where θ^{MAP} is the mostly likely hypothesis given the observed dataset $S = \{s_1, s_2, ..., s_n\}$ and θ^{ML} is the most likely hypothesis given that all explored hypotheses are assumed equiprobable. We omit ML solutions from the remainder of this section since they are easily obtained by removing the "$-\log p(\theta)$" term from the MAP solutions.

In a supervised context the supervised MAP solution can be written more specifically as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(-\sum_{i=1}^{n} \log F(y_i, \mathbf{x}_i; \theta) - \log p(\theta) \right) \tag{2.82}$$

In a classification context (when the variable Y is categorical) the MAP solution may be written as cross-entropy minimization, either between a) the estimated and empirical distributions:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{F}(\mathbf{x}_i; \theta)) - \log p(\theta) \right) \tag{2.83}$$

or b) the estimated and target distributions:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{t}_i, \mathbf{F}(\mathbf{x}_i; \theta)) - \log p(\theta) \right) \tag{2.84}$$

If the output sample is an empirical distribution of class labels instead of the class label itself, we refer to the output samples as 'soft' labels. The classification MAP solution for soft labels may be written as the following cross entropy minimization:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} m_i \cdot H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{F}(\mathbf{x}_i; \theta)) - \log p(\theta) \right) \tag{2.85}$$

where

$$m_i = \frac{\sum_{k=1}^{n} \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\}}{\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{\mathbf{x}_k = \mathbf{x}_i\}} \tag{2.86}$$

and $r_k \in [1; \infty)$ is the reliability of sample k. Equation 2.85 simplifies as follows if the output samples are equi-reliable:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{F}(\mathbf{x}_i; \theta)) - \kappa \log p(\theta) \right) \tag{2.87}$$

where $\kappa \in (0; 1]$ is a constant. The MAP solution for soft labels may also be written in terms of the target distribution t_i:

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} r_i \cdot H(\mathbf{t}_i, \mathbf{F}(\mathbf{x}_i; \theta)) - \log p(\theta) \right) \tag{2.88}$$

which simplifies to the form of equation 2.87 (with $\mathbf{p}_{\mathbf{x}_i}$ replaced by $\mathbf{t}_i$) when output samples are equi-reliable.

Finally, since $H(\mathbf{p}, \mathbf{q}) = D_{KL}(\mathbf{p}||\mathbf{q}) - H(\mathbf{p}, \mathbf{p})$ and since $p_{\mathbf{x}_i}$, t_i are constant with respect to θ, all cross entropy minimization solutions in this section are equivalent to Kullback-Leibler minimizations. In other words, all instances of $H(\mathbf{p}, \mathbf{q})$ may be replaced by $D_{KL}(\mathbf{p}||\mathbf{q})$.

2.3 Finding the Best Hypothesis (ERM Formulation and Loss Functions)

We have established that the supervised training goal is to find the value of θ which minimizes the model's performance error (given the training dataset $\mathcal{S}$). One approach is to find the hypothesis $F(\mathbf{X}, \theta)$ most likely to represent the true prediction function, aka the maximum a priori solution. In a typical classification setting, we have established that the MAP solution is

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{F}(\mathbf{x}_i; \theta)) - \log p(\theta) \right) \tag{2.89}$$

From a standpoint of probability theory, the MAP solution is optimal: it represents the hypothesis most likely to be correct. However, in many applications it is preferable to seek the solution with the best expected performance.

Suppose, for example, that the value Y represents daily stock market returns, and the value X represents cofactors used to predict Y. While the equation 2.89 gives the solution most likely to accurately predict Y, investors prefer a solution which predicts Y with a high degree of confidence[5]

The goal then becomes to minimize the expected loss, or population risk:

$$\theta^* = \underset{\theta}{\operatorname{argmin}} \, \mathbb{E}_{(\mathbf{x},y)\sim S} \, \ell\big((\mathbf{x}, y); \theta\big) \tag{2.90}$$

where the loss function $\ell(\mathbf{x}, y; \theta)$ measures the cost of performance errors. Since the distribution of S is unknown, the population risk is estimated by taking the expected loss over the training dataset:

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}} \, \mathbb{E}_{(\mathbf{x},y)\sim D} \, \ell\big((\mathbf{x}, y); \theta\big) \tag{2.91}$$

$$= \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{n} \ell\big((\mathbf{x}_i, y_i); \theta\big) \tag{2.92}$$

where D is a random variable representing input-output samples $(\mathbf{x}, y)$, and is distributed as the empirical distribution $p(\mathbf{x}, y)$. This formulation is typically referred to as empirical risk minimization (ERM)[44]. As the training dataset size increases the ERM estimation converges to the expected loss solution (equation 2.90). When the loss function is cross-entropy, the ERM solution is equivalent to the classification ML solution (equation 2.63). When the loss function is MSE, the ERM solution is equivalent to the numerical regression ML solution (equation 2.48). For convenience, we also define the overall lost function $\mathcal{L}(\theta)$:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell\big((\mathbf{x}_i, y_i); \theta\big) \tag{2.93}$$

[5]Investors seek to maximize expected gains, not maximize the likelihood of a gain! A portfolio highly likely to return a gain is eventually disastrous if its expected gain is negative. See Martingale bets.

So that we may reframe the ERM solution as a loss minimization problem:

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta) \tag{2.94}$$

Even for applications where the most likely hypothesis is preferred, the MAP solution requires finding the minimum value of $\sum_{i=1}^{n} H(\mathbf{p}_{\mathbf{x}_i}, \mathbf{q}_{\mathbf{x}_i}) - \log p(\theta)$, which is non-convex (since cross entropy is non-convex). Optimizers therefore return a local minimum, which may not be the optimal solution. This means that alternative loss functions that are easier to optimize may achieve superior results. For example, convex loss functions allow convex optimizers to find optimal solutions. Smooth loss functions improve the training speed and accuracy of optimizers using gradient descent.

In the context of neural networks, most loss functions have the additional property that they are differentiable[6], such that gradient descent optimization can be performed.

Since performance measurement is domain-specific, the default loss function tends to be cross-entropy, which is differentiable, and – as noted above – has the added utility that the θ^{ERM} is also the ML solution. We will see in the next section that, when combined with weight decay regularization and under reasonable assumptions, the ERM solution using cross-entropy loss yields the most likely (aka MAP) hypothesis.

2.4 Model Complexity and Overfitting

Figure 2.4 depicts the conventional relationship between model complexity and model errors $\mathbb{E}_T$ and $\mathbb{E}_V$[15]. The term $\mathbb{E}_T$, known as in-sample error or training error, is the average error of the model over the training samples. The out-of-sample error $\mathbb{E}_V$, or test error, is the average error of the model over testing samples[7]. Figure 2.4 suggests that as model complexity increases, training error decreases monotonically, and testing error decreases until an ideal model complexity is reached, after which it begins to increase.

While these relationships are not systematically observed in practice, figure 2.4 serves as a guideline for understanding the relationship between model complexity and training/test errors. The decreasing curve of $\mathbb{E}_T$ is straightforward to understand: as the model is trained to minimize $\mathbb{E}_T$, its ability to do so improves as the model becomes capable of learning arbitrarily complex patterns. As the model's capacity increases, it becomes capable of memorizing the training data, which in turn drives training error to zero.

As in section 2.1.3, let $\mathbb{H}$ be the hypothesis space and $\mathbb{G}$ be the set of ground truth functions. Let $\mathbb{G}^c = \mathbb{H} - \mathbb{G}$. As a model becomes more complex, it is likely to become more expressive, and thus testing error decreases. Once the model is sufficiently expressive (ie

[6] In reality, loss functions must be differentiable almost everywhere, which is sufficient for practical algorithms to perform gradient descent.

[7] The terms $\mathbb{E}_T$ and $\mathbb{E}_V$ will be defined formally in section 2.9.

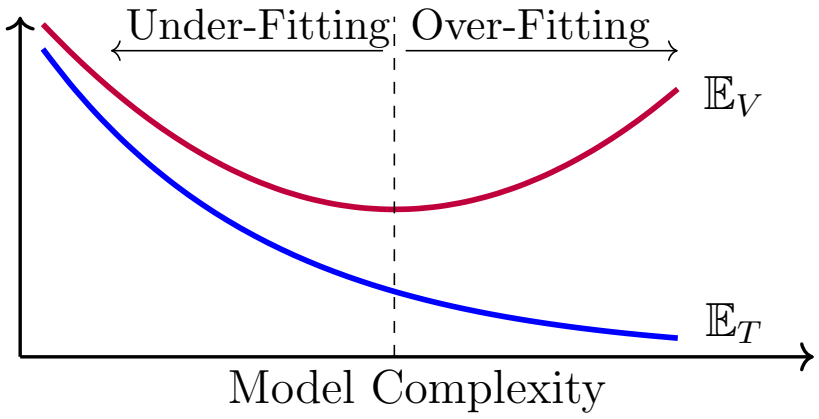


Figure 2.4: Training ($\mathbb{E}_T$) and test ($\mathbb{E}_V$) errors vs model complexity. Adapted from [27].

$\mathbb{H} \supseteq \mathbb{G}$), increasing complexity will no longer improve expressivity, as $\mathbb{H}$ already encompasses $\mathbb{G}$. Instead the chance that a model will converge to a member in $\mathbb{G}^c$ increases. This either means that training was unsuccessful, and the optimization algorithm was unable to minimize training error, and 'settled' on a member of $\mathbb{G}^c$; or that training error was successfully minimized, but the testing error remains high. The latter scenario is referred to as overfitting.

In general when a model performs well over its training data, but worse over testing data, we say that the model's parameters have 'overfit' the training data. In such a case, the model has learned a pattern that exists in the training data, but not in the population as a whole. For example, suppose a classification model is taught to distinguish between images of cats and dogs, and its training data contains only images of black cats and white dogs. If the testing data contains a white cat, the model might identify it as a dog, having identified color as a the distinguishing feature over the training data.

Alternatively, in the context of model complexity, an 'overfitting' model is sometimes defined as an insufficiently complex model – in other words, a model in which the likelihood of overfitting is not minimized. While this is the definition used in figure 2.4, this is not the definition used elsewhere in this thesis.

By contrast, an underfit model is one which insufficiently fits the training data. This may happen for several reasons. The model may be insufficiently complex or training may have failed, described in section 2.1.3. Alternatively, the model may be too regularized (see section 2.5), fitting instead the imposed constraints better than the training data.

2.5 Regularization

Regularization is an umbrella term used to describe any scheme that modifies the conventional optimization problem in an effort to improve generalization[6, p.256], or achieve some desired model characteristic such as robustness to adversarial attacks. The emphasis of this paper is on improving generalization, so we focus on this aspect of regularization.

From a probabilistic perspective, we seek to make the probability $p(\theta|\mathcal{S})$ more accurate by further conditioning on priors not already contained in $\mathcal{S}$. From an ERM perspective, we seek to make solution more accurate by constraining it on assumptions that we believe to be true. In both cases, we impose on the problem some prior knowledge not already contained in the training dataset.

There are two flavors of regularization: model confining and data augmentation[8]. Model confining methods constrain models to learning in a subset of parameter space which has a higher probability of generalizing well. Notable examples are weight decay [21] and dropout [39]. As mentioned in section 2.2.4, the standard assumption of numerical regression, is also a form of model confining, where we condition on $F(\mathbf{X}, \theta) \sim \mathcal{N}(Y, \sigma^2)$.

The expression $p(\theta)$ is the prior probability of the hypothesis θ. This probability is entirely governed by the assumptions made by the problem designer about the distribution of model parameters. In the MAP solution, the term $\log p(\theta)$ can thus be seen as a model-confining regularization term.

Data augmentation methods add transformed versions of input training samples to the original training set. Conventionally, transformed input samples retain their original label, so that models effectively see a larger set of input-output training pairs. Commonly applied transformations in image applications include flips, crops and rotations[32]. If we let the sequence of new input-output pairs be $\mathcal{T}$, the MAP solution maximizes $p(\theta|\mathcal{S}, \mathcal{T})$ instead of $p(\theta|\mathcal{S})$.

2.5.1 Weight Decay

ERM interpretation

Weight decay[21], also known as L2, Ridge, or Tikhonov regularization, consists in adding the term $\lambda||\theta||_2^2$ to the ERM solution:

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} \ell\big((\mathbf{x}, y); \theta\big) + \lambda||\theta||_2^2 \right) \tag{2.95}$$

The term $\lambda||\theta||_2^2$ heavily penalizes large weights (aka components of θ). This imposes on the optimizer the assumption that models with small weights are more likely to generalize well. To gain some intuition as to why this makes sense in the context of a classifier network, consider a single node:

$$a = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{2.96}$$

where $\mathbf{w}$, are the set of weights (or components of θ) contained in the node in question, σ is some non-linear activation function, and a is the output (or activation) of the node. If we let x be an $N \times 1$ vector, then $\mathbf{w}$ is an $1 \times N$ vector, such that prior to activation the

[8] Some authors describe these flavors as "data independent" and "data-dependent" [14].

node performs an linear combination of $\mathbf{x}$. The input $\mathbf{x}$ may be seen as a set of features from the previous layer, and the output a represents a new meta-feature.

We note that there are infinitely many possible boundaries between the training data points, and therefore infinitely many possible weight vectors $\mathbf{w}$ for a given node. Therefore there are no constraints in the original problem that prevent the neural network from learning artificial linear combinations with extremely large weights.

By further noting that each component of $\mathbf{w}$ represents the weight assigned to a particular feature, we intuitively expected most features to have small contributions, and other features to have heavy, yet bounded weights. Thus, penalizing heavy weights prevents neural networks from generating nonsense features that overfit the training dataset.

Maximum a posteriori interpretation

Weight decay may also be interpreted as imposing a gaussian prior using the *maximum a posteriori* formulation (equation 2.61) [42]. Assume that θ parameters are picked from independent, zero-centred normal distributions with a constant variance[9]. The term $p(\theta)$ is then a multi-variate gaussian with a covariance matrix given by

$$\Sigma = \sigma^2 \cdot I \tag{2.97}$$

where σ^2 is the aforementioned constant variance and I is the identity matrix. We therefore have

$$\log p(\theta) = \log \frac{\exp(-\frac{1}{2}\theta^T \Sigma^{-1} \theta)}{\sqrt{(2\pi)^k |\Sigma|}} \tag{2.98}$$

$$= -\frac{1}{2\sigma^2} \cdot \theta^T \theta - \log(\sqrt{(2\pi)^k |\Sigma|}) \tag{2.99}$$

$$= -\lambda ||\theta||_2^2 - \alpha \tag{2.100}$$

where $\lambda = \frac{1}{2\sigma^2}$ so $\lambda > 0$, and where α is constant relative to θ. Plugging equation 2.100 into to the standard classification MAP solution (equation 2.89) yields

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} H(p_{\mathbf{x}_i}, q_{\mathbf{x}_i}) + \lambda ||\theta||_2^2 \right) \tag{2.101}$$

And thus applying weight decay regularization to the cross-entropy ERM solution may be interpreted as yielding the most likely (or MAP) classification solution under the assumption that θ parameters are independent and zero-centred gaussians. Note that the MAP solution for equi-reliable soft labels (equation 2.87) also reduces to equation 2.101, with the distinction that $\lambda = \frac{\kappa}{2\sigma^2} > 0$. Since λ is a positive number to be tuned manually in both cases, this distinction has no practical consequence.

[9] Or equivalently, assume that $p(\theta)$ is a zero-mean spherical gaussian distribution.

Similarly, plugging equation 2.100 into the MAP regression solution (equation 2.47) yields

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i=1}^{n} (y_i - F(\mathbf{x}_i; \theta)^2) - \lambda ||\theta||_2^2 \right) \tag{2.102}$$

where $\lambda = \frac{\phi}{2\sigma^2} > 0$. Thus applying weight decay regularization to the cross-entropy and MSE-loss ERM solutions may be interpreted as yielding the mostly likely regression solution.

2.5.2 Adversarial Training

Conventional adversarial training schemes augment the original training dataset by searching for approximations of true adversarials within bounded regions around each training sample [13, 37]. For a training sample $\mathbf{x}$, a bounded region U known as an L_p ball is defined as $U = \{\mathbf{x} + \boldsymbol{\eta} : ||\boldsymbol{\eta}||_p < \epsilon\}$. Over this region, the loss function with respect to the true label of $\mathbf{x}$ is maximized. A typical loss function for an adversarial scheme is

$$\ell\big((\mathbf{x}, y); \theta\big) = \max_{\tilde{\mathbf{x}} \in U} \ell_b\big((\tilde{\mathbf{x}}, y); \theta\big) \tag{2.103}$$

where ℓ_b is the baseline loss function. Simply put, baseline training serves to learn correct classification over the training data, whereas adversarial training moves the classification boundary to improve generalization. Simply put, adversarial training moves the classification boundary relative to baseline training.

More generally, adversarial training may use any region U around $\mathbf{x}$ (not necessarily restricted to an L_p ball) that helps achieve the regularization goal. Using an adequately crafted region U, the classification boundary movement may serve to improve generalization, or it may serve to improve adversarial robustness.

Figure 2.5 illustrates the adversarial training regularization scheme. Consider the problem of classifying the blue points and the green points, where the dashed curve is a ground-truth classifier and the black curve indicates the classification boundary of $\mathbf{F}(\mathbf{x}; \theta)$, which overfits the training data. A training sample $\mathbf{x}$ is moved to a location within an L_p-ball around $\mathbf{x}$ while keeping its label y to further train the model; the location, denoted by $\widehat{\mathbf{x}}_1$ is chosen to maximize training loss as per equation 2.103.

In this example, the regularizing effect will be a local movement of the overfitted classifier boundary toward $\widehat{\mathbf{x}}_1$, which in turn moves it closer to the ground truth boundary.

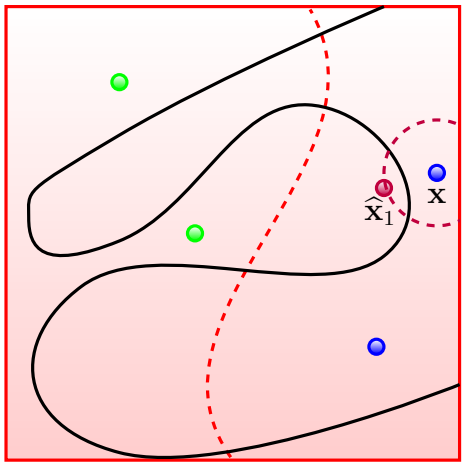


Figure 2.5: Adversarial training

2.5.3 MixUp

This section describes MixUp as first explained by the original authors in [49]. Let $\mathbb{S}_D$ be the set of unique input samples in $\mathcal{S}$. Let $(\mathbf{x}, \mathbf{y})$ and $(\mathbf{x}', \mathbf{y}')$ be two samples drawn independently from $\mathbb{S}_D$.

MixUp is a data augmentation scheme in which samples are linearly combined using some mixing ratio $\lambda \in [0, 1]$:

$$\mathbf{x}_g = \lambda \mathbf{x} + (1 - \lambda)\mathbf{x}' \tag{2.104}$$

where $\lambda \sim P^{\text{Mix}}$. A target label is generated using the same mixing ratio λ:

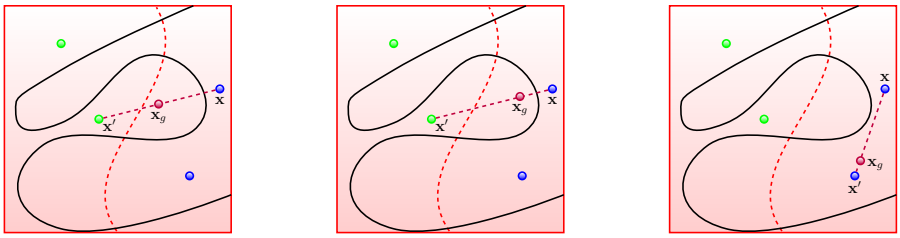
$$\mathbf{y}_g = \lambda \mathbf{y} + (1 - \lambda)\mathbf{y}' \tag{2.105}$$

The loss function for the MixUp scheme is

$$\ell^{\text{Mix}}\big((\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}'), \lambda; \theta\big) := \ell_b\big((\lambda \mathbf{x} + \overline{\lambda}\mathbf{x}', \lambda \mathbf{y} + \overline{\lambda}\mathbf{y}'); \theta\big) \tag{2.106}$$

where ℓ_b is the baseline loss function.

Figure 2.6 illustrates the input mixing procedure used in the MixUp regularization scheme. As in section 2.5.2, consider the problem of classifying the blue points and the green points, where the dashed curve is a ground-truth classifier and the black curve indicates the classification boundary of $\mathbf{F}(\mathbf{x}; \theta)$, which overfits the training data. Training samples $\mathbf{x}$ and $\mathbf{x}'$ are mixed using the mixing ratio λ indicated in the subfigure captions. Figures 2.6a and 2.6b show an example of $\mathbf{x}$ and $\mathbf{x}'$ belonging to different classes, while in figure 2.6c $\mathbf{x}$ and $\mathbf{x}'$ belong to the same class.



(a) Inter-class $\lambda = 0.5$ (b) Inter-class $\lambda = 0.8$ (c) Intra-class $\lambda = 0.9$

Figure 2.6: MixUp input sample mixing

Suppose that the blue label $\mathbf{y}_u$ and green label $\mathbf{y}_a$ correspond respectively to

$$\mathbf{y}_u = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{y}_a = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Then $\mathbf{y} = \mathbf{y}_u$ in all figures. In figures 2.6a and 2.6b $\mathbf{y}' = \mathbf{y}_a$, while in figure 2.6c $\mathbf{y}' = \mathbf{y}_u$.

The resulting mixed labels $\mathbf{y}_g$ are then

$$\text{a)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 0.5 \\ 0 \\ 0 \\ 0.5 \end{bmatrix} \qquad \text{b)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 0.8 \\ 0 \\ 0 \\ 0.2 \end{bmatrix} \qquad \text{c)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

In the inter-class mixing examples (figures 2.6a and 2.6b) the regularizing effect will be a local movement of the overfitted classifier boundary toward $\widehat{\mathbf{x}}'$, which in turn moves it closer to the ground truth boundary. In the intra-class example, the regularizing effect is unclear in terms of classifier-boundary movement. However, it may help prevent underfitting; that is, it may prevent the classifier boundary from moving 'too much'

2.6 Neural Networks and Deep Learning

In the context of artificial intelligence, a neural network is a function constructed using a set of connected components called neurons[5]. As depicted in figure 2.7, a neuron is a function of the form

$$n(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b) \tag{2.107}$$

where $\mathbf{x}$ is an input vector, $\mathbf{w}$ is an equally sized vector of weights, and b is a scalar referred to as a bias. The output $n(\mathbf{x})$ is called the activation of the neuron, and σ is

a non-linear function referred to as the activation function. The input to the activation function $z = \mathbf{w}^T\mathbf{x} + b$ is sometimes referred to as the pre-activation.

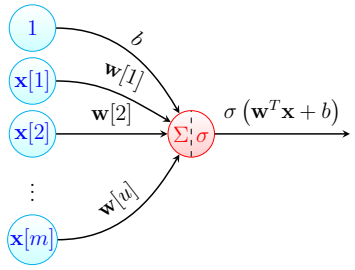


Figure 2.7: A neural network neuron. Adapted from [27].

In the context of machine learning, the weights and biases of neurons are typically learned through gradient descent (section 2.7) and backpropagation (section 2.8). Neurons are usually configured in layers whose neurons are connected to immediately preceding and following layers, but not to each other. In addition to neuron layers, the input and output layers refer to the input and output of the neural network, respectively. The neuron layers in between the input and output layers are referred to as hidden layers. A neural network with more than one hidden layer is sometimes called a 'deep' network, and training of such networks is called *deep learning*. A node refers to an input, output, or neuron – in other words, any member of a layer.

In addition to neurons, neural networks may contain other components which are typically static (as opposed to learnable), and are usually differentiable so as to be usable in the backpropagation algorithm. Such components are typically organized in layers. Common components include convolution layers, rank filters, label embedding, and gates. While these are common design components, they are beyond the scope of this thesis.

The simplest class of neural networks is called a feedforward neural network, which is a network containing an acyclic configuration of neurons. A fully connected network is a network in which all neurons are connected to each node of the preceding layer. A fully connected feedforward neural network is called a multi-layer perceptron, or MLP. An MLP with two hidden layers is depicted in figure 2.8.

Let the function $\boldsymbol{\sigma}$ be a vectorized version of σ, where $\boldsymbol{\sigma}(\mathbf{x})[j] = \sigma(\mathbf{x}[j])$. We define the softmax function $\mathbf{s} : \mathbb{R}^m \to \mathbb{R}^m$ where

$$\mathbf{s}(\mathbf{x})[j] = \frac{e^{\mathbf{x}[j]}}{\sum_{w=1}^{m} e^{\mathbf{x}[w]}} \tag{2.108}$$

The function representing a particular layer l can therefore be written as

$$\mathbf{m}^{(l)}(\mathbf{x}) = \boldsymbol{\sigma}(\overline{\mathbf{W}}^{(l)}\mathbf{x} + \mathbf{b}^{(l)}) \tag{2.109}$$

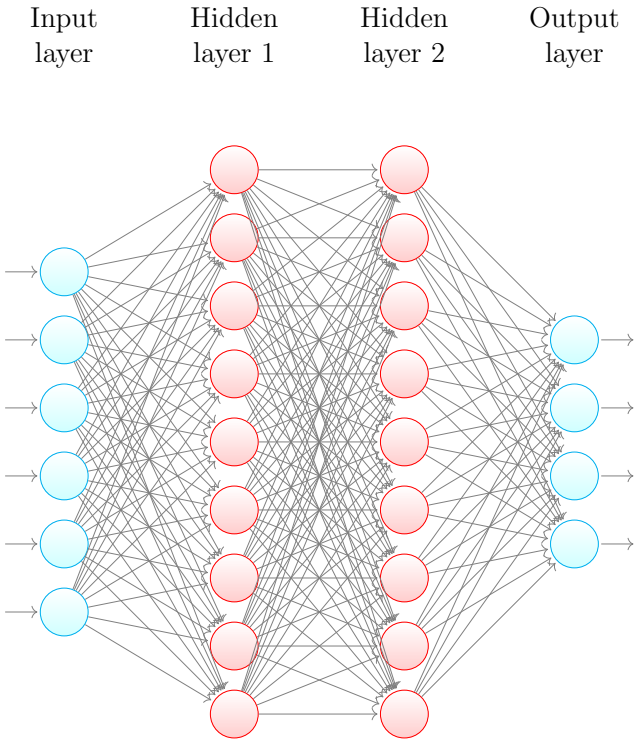


Figure 2.8: A multi-layer perceptron. Borrowed from [27].

where

$$\overline{\mathbf{W}}^{(l)} = \begin{bmatrix} \mathbf{w}_1^{(l)T} \\ \mathbf{w}_2^{(l)T} \\ \vdots \\ \mathbf{w}_{u(l)}^{(l)\ T} \end{bmatrix} \tag{2.110}$$

and where $\mathbf{w}_k^{(l)}$ is the of weight vector of neuron k in layer l, and layer l has $u(l)$ neurons.

The global function of an MLP can therefore be written as

$$F(\mathbf{x};\theta) = \mathbf{s}\left(\mathbf{m}^{(L)}\left(\mathbf{m}^{(L-1)}\left(\dots \mathbf{m}^{(1)}(\mathbf{x})\right)\right)\right) \tag{2.111}$$

$$= \mathbf{s}\left(\boldsymbol{\sigma}\left(\overline{\mathbf{W}}^{(L)}\boldsymbol{\sigma}\left(\overline{\mathbf{W}}^{(L-1)}\dots\boldsymbol{\sigma}(\overline{\mathbf{W}}^{(1)}\mathbf{x}+\mathbf{b}^{(1)})\cdots+\mathbf{b}^{(L-1)}\right)+\mathbf{b}^{(L)}\right)\right) \tag{2.112}$$

By contrast to a feedforward network, a recurrent neural network (RNN) refers to a network containing synchronous, cyclic node connections[10], which thus allow RNNs to possess memory. An asynchronous loop is mathematically possible (eg. an infinite geometric

[10] A synchronous cycle requires at least two time steps to complete.

sum), but would require an acyclic implementation since execution time is finite – thus RNNs are the only practical implementations of cyclic networks. This in turn makes RNNs the only practical alternativeto feedforward networks.

There are two sub-types of RNN networks: finite impulse response (FIR), and infinite impulse response (IIR) networks[11]. FIR networks store information from a finite number of past inputs, while the internal state of IIR systems is indefinitely affected by past inputs. From a graph-theory perspective, FIR networks contain only undirected cycles, while IIR networks contain one or more directed cycles.

The Elman network [10] was one of the first published RNNs, and is often referred to as a 'vanilla' RNN. The Elman network can be characterized by the following two equations

$$\mathbf{h}_t = \boldsymbol{\sigma}(\overline{\mathbf{W}}\mathbf{x}_t + \overline{\mathbf{U}}\mathbf{h}_{t-1} + \mathbf{b}) \tag{2.113}$$

$$\mathbf{o}_t = \overline{\mathbf{V}}\mathbf{h}_t + \mathbf{c} \tag{2.114}$$

where $\mathbf{x}_t$ is the network's input at time t, and $\mathbf{h}_t$ is the network's state at time t, and $\mathbf{o}_t$ is the output at time t. The previous state of the network, $\mathbf{h}_{t-1}$, is sometimes referred to as the context, so-called because past information is retained via context nodes.

To perform gradient descent, RNN networks use a variant of backpropagation called backpropagation-through-time (BPTT). In BPTT, the RNN network is unfolded at each time step such that it becomes a single deep feedfoward net. The derivative chain rule is applied in a similar way to standard backpropagation to derive the network's gradient with respect to its weights.

Note that because the resulting unfolded network is very deep, it is susceptible unstable gradients (vanishing or exploding). This can be seen by inspecting equation 2.142: if terms $\sum_{k=1}^{u(l-1)} \frac{\partial a_j^{(l)}}{\partial a_k^{(l-1)}}$ have magnitude less than 1, the first layers' have a very small gradient. Similarly if $||\sum_{k=1}^{u(l-1)} \frac{\partial a_j^{(l)}}{\partial a_k^{(l-1)}}|| > 1$, the first layers may have very large gradients.

The universal approximation theory states that under mild assumptions on the activation function, a feedforward network may approximate any function. Many variants of this theory exist [8, 24, 26]. The effect is that feedforward neural networks are highly expressive with the appropriate choice of activation, width, and depth. RNNs, on the other hand, are provably turing-complete [38]. This means that for any input sequence and for any algorithm over this sequence, there exists an RNN that can implement this algorithm. In other words suppose $\mathbb{I}$ and $\mathbb{O}$ are input and output sequence spaces respectively. Then for all computable functions $f : \mathbb{I} \to \mathbb{O}$ there exists an RNN that can implement f.

[11]Some authors use the term RNN to refer strictly to IIR networks, and define 'FIR networks' as a separate category. Per our earlier definition of RNN, we include both FIR and IIR nets in the RNN family.

2.7 Gradient Descent

We have established the following training goal

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta) \tag{2.115}$$

In other words, we seek to minimize $\mathcal{L}(\theta)$. We assume that the loss function is differentiable almost everywhere, such that the gradient $\nabla_\theta \mathcal{L}(\theta)$ can be found for all practical values of θ. Gradient descent is based on two observations about multi-variable differentiable functions: 1) The points where $\nabla_\theta \mathcal{L}(\theta) = 0$ represent local minima, local maxima, or stationary points. 2) The negative gradient $-\nabla_\theta \mathcal{L}(\theta_0)$, where θ_0 is some specific value of θ, represents the direction of steepest ascent at θ_0. In other words, it represents the direction to move the vector θ at point θ_0 to maximally decrease the value of $\mathcal{L}(\theta)$.

Thus if $\nabla_\theta \mathcal{L}(\theta) \neq 0$, for a sufficiently small learning rate τ, we may achieve a smaller value of $\mathcal{L}(\theta)$ by performing the following operation:

$$\theta_{new} = \theta_0 - \tau_g \nabla_\theta \mathcal{L}(\theta_0) \tag{2.116}$$

such that

$$\mathcal{L}(\theta_{new}) < \mathcal{L}(\theta_0) \tag{2.117}$$

Gradient descent consists of repeating equation 2.116 until $\nabla_\theta \mathcal{L}(\theta)$ is very small. By adapting τ_g strategically, one can ensure that $\mathcal{L}(\theta)$ converges to a local minimum. Typically, strategies for updating the learning rate involve decreasing τ_g as the magnitude of $\nabla_\theta \mathcal{L}(\theta)$ decreases. Adaptive learning rates are an area of research unto themselves, but this is beyond the scope of this thesis.

We note that the major weakness of gradient descent is that it does not guarantee that the global minimum of $\mathcal{L}(\theta)$ is achieved. When the local minimum achieved is unsatisfactory, training may be repeated by starting from a new random value of θ, until the local minimum is satisfactory. Gradient descent may thus be used to achieve a reasonably accurate solution to equation 2.115.

Figure 2.9 depicts an example of gradient descent. The x and y axes are represented with red arrows pointing to the right and into the plane of the paper, respectively. The z-axis is represented with a yellow arrow pointing up. The x-y plane is shown as a grid at the floor of the diagram.

In this example, θ is two-dimensional[12] and the x,y axes represent $\theta[1]$ and $\theta[2]$ respectively. The loss function $\mathcal{L}(\theta)$ is plotted as a red surface in the z-axis, above the x-y plane. The dashed yellow line represents the gradient descent curve, with each dash representing an update.

An in depth review of gradient descent can be found in [35]

[12] In other words, the model has two weights.

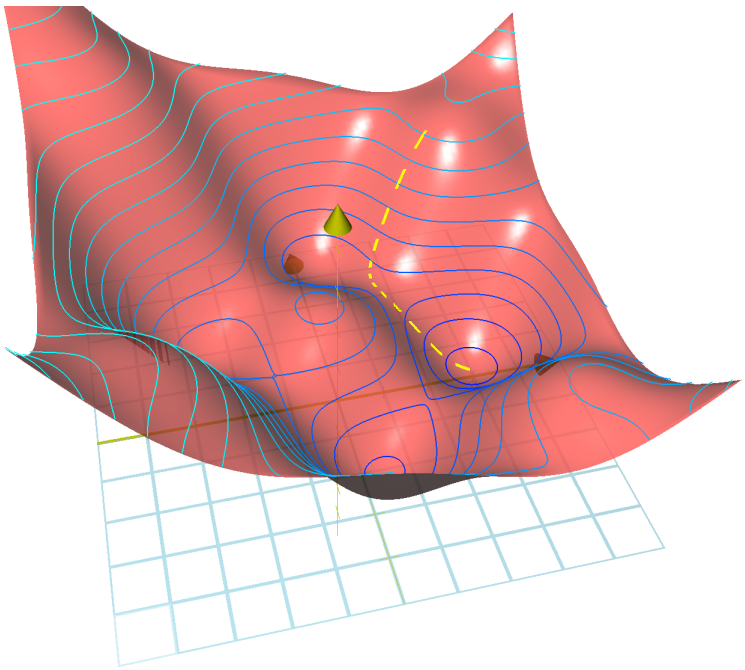


Figure 2.9: Gradient descent over a loss function. Plot code provided by [1].

2.7.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a generalization of gradient descent in which an estimation of the gradient $\nabla_\theta \mathcal{L}(\theta_0)$ is used to improve convergence speed[7]. In the gradient descent method described earlier in this section, the true gradient $\nabla_\theta \mathcal{L}(\theta_0)$ is used in equation 2.116 to update the value of θ. In SGD, we will substitute the gradient in equation 2.116 with the approximation $\widetilde{\nabla}_\theta \mathcal{L}(\theta_0)$.

The derivative of a sum is the sum of the derivatives. Thus the gradient at some specific θ_0 is expressed as

$$\nabla_\theta \mathcal{L}(\theta_0) = \nabla_\theta \left[\frac{1}{n} \sum_{i=1}^{n} \ell\Big((\mathbf{x}_i, y_i); \theta_0 \Big) \right] \tag{2.118}$$

$$= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \left[\ell\Big((\mathbf{x}_i, y_i); \theta_0 \Big) \right] \tag{2.119}$$

$$= \cdot \mathbb{E}_{(\mathbf{x},y)\sim p(\mathbf{x},y)} \nabla_\theta \left[\ell\Big((\mathbf{x}, y); \theta_0 \Big) \right] \tag{2.120}$$

where $p(\mathbf{x}, y)$ is the empirical distribution of samples $(\mathbf{x}, y)$ in $\mathcal{S}$.

The overall loss gradient $\nabla_\theta \mathcal{L}(\theta_0)$ is thus the expected value of the sample loss gradients $\nabla_\theta \Big[\ell\Big((\mathbf{x}, y); \theta_0 \Big) \Big]$. We may estimate the expected value using a subset of the samples $(\mathbf{x}, y) \in S$. Let $\zeta_1^b := (\zeta_1, \zeta_2, \ldots, \zeta_b)$ be a batch, or sequence of values drawn i.i.d. from

$p(\mathbf{x}, y)$, where the batch size b is $\leq n$. We may then estimate $\nabla_\theta \mathcal{L}(\theta_0)$ as

$$\widetilde{\nabla}_\theta \mathcal{L}(\theta_0) = \frac{1}{b} \cdot \sum_{i=1}^{b} \nabla_\theta \Big[\ell\Big(\zeta_i; \theta_0\Big) \Big] \tag{2.121}$$

where we note that $\widetilde{\nabla}_\theta \mathcal{L}(\theta_0) = \nabla_\theta \mathcal{L}(\theta_0)$ when $b = n$, such that gradient descent is a special case of SGD.

The gradient descent update equation (equation 2.116) becomes

$$\theta_{new} = \theta_0 - \tau_s \widetilde{\nabla}_\theta \mathcal{L}(\theta_0) \tag{2.122}$$

where $\tau_s = \frac{1}{b}\tau_s$, and is adapted similarly to τ_g. Since τ_s is adaptive, the scalar $\frac{1}{b}$ be ignored, and the update step is algorithmically identical for both gradient descent and SGD.

Similarly to gradient descent, it can be proven that SGD converges to a local minimum of $\mathcal{L}(\theta)$. While the SGD gradient is less precise and thus a larger number of updates (via equation 2.122) is required to achieve a local minimum, the execution time of each update is $\frac{1}{b}$ faster. In typical problems, this results in a vast speedup in convergence to the local minimum.

Figure 2.10 depicts an example of stochastic gradient descent. It depicts the same loss function $\mathcal{L}(\theta)$ as that shown in figure 2.9. In contrast with figure 2.9, gradient descent updates are performed using stochastic gradient descent (with $b < n$).

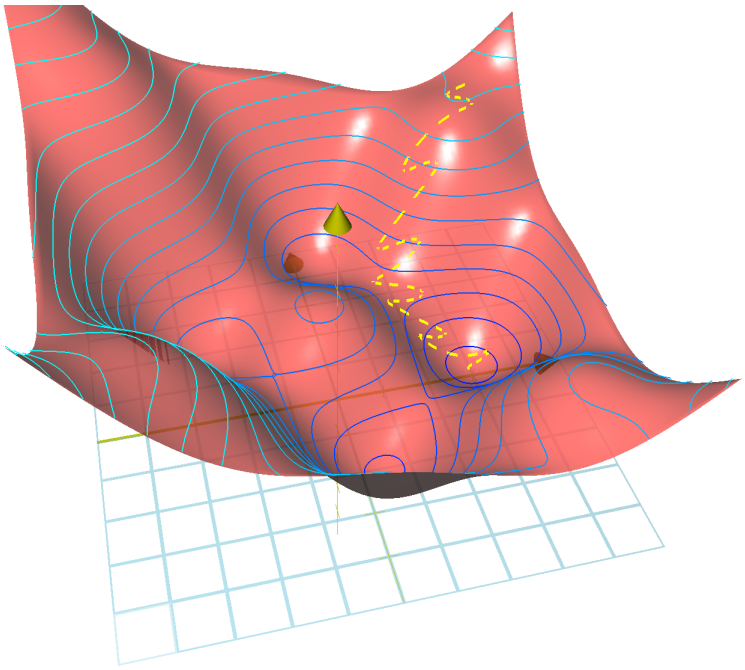


Figure 2.10: Stochastic gradient descent over a loss function. Plot code provided by [1].

2.8 Backpropagation

We have established that the gradient $\nabla_\theta \mathcal{L}(\theta)$ allows us to solve the ERM formulation (equation 2.115) using gradient descent. Backpropagation, in turn, allows us to compute $\nabla_\theta \mathcal{L}(\theta)$. More precisely, backpropagation is an automatic differentiation algorithm which leverages the derivative chain rule to compute the gradient $\nabla_\theta \mathcal{L}(\theta)$[18]. The loss $\mathcal{L}(\theta)$ is first evaluated by 'forward-propagation' of the inputs $\mathbf{x}_i$ through the network, and the gradient is computed by backward accumulation of intermediate derivatives. In this section we illustrate backpropagation using a standard MLP neural network. We also note that backpropagation can be used to compute $\widetilde{\nabla}_\theta \mathcal{L}(\theta)$ for stochastic gradient descent, by merely summing over the sequence ζ_1^b instead of the full training dataset s_1^n.

We begin with the full gradient expression of equation 2.119:

$$\nabla_\theta \mathcal{L}(\theta_0) = \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \Big[\ell(\mathbf{x}_i, y_i; \theta_0) \Big] \tag{2.123}$$

For clarity we express $\nabla_\theta \Big[\ell(\mathbf{x}_i, y_i; \theta_0) \Big]$ in vector notation:

$$\nabla_\theta \Big[\ell(\mathbf{x}_i, y_i; \theta_0) \Big] = \begin{bmatrix} \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial \theta[1]} \\ \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial \theta[2]} \\ \vdots \\ \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial \theta[\rho]} \end{bmatrix} \tag{2.124}$$

where ρ is the number of weights in θ. Thus to find $\nabla_\theta \mathcal{L}(\theta_0)$ we must find $\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial \theta[k]}$ for every permutation of (i, j) where $i = 1, 2, ..., n$ and $k = 1, 2, ..., \rho$. That is, we must find the partial derivative of $\ell(\mathbf{x}_i, y_i; \theta)$ for every training sample $(\mathbf{x}_i, y_i)$ and with respect to every weight $\theta[k]$.

Since we are discussing neural networks, the learnable parameters of θ reside in neurons, which in turn reside in layers. We denote by $a_k^{(l)}$ the activation of neuron j in layer l, by $w_{jk}^{(l)}$ the weight connecting neuron j of layer $l-1$ to neuron k of layer l, and by $b_k^{(l)}$ the bias of neuron k in layer l. For notational convenience, we define $a_k^{(0)} = \mathbf{x}_i[k]$. Thus,

$$a_k^{(l)} = \sigma(\mathbf{w}_k^{(l)^T} \mathbf{a}^{(l-1)} + b_k^{(l)}) \tag{2.125}$$

$$= \sigma(w_{1k}^{(l)} a_1^{(l-1)} + w_{2k}^{(l)} a_2^{(l-1)} + ... + w_{uk}^{(l)} a_u^{(l-1)} + b_k^{(l)}) \tag{2.126}$$

where σ is some non-linear activation function and layer l contains u neurons.

We may thus restate our goal as finding $\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial w_{jk}^{(l)}}$ for all $i \in [1..n]$ and all weights $w_{jk}^{(l)}$ in the network, and $\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial b_j^{(l)}}$ for all $i \in [1..n]$ and all biases $b_k^{(l)}$ in the network.

For convenience, we denote the input to the activation function (aka the preactivation) of neuron k, layer l as $z_k^{(l)}$:

$$z_k^{(l)} = \mathbf{w}_k^{(l)^T} \mathbf{a}^{(l-1)} + b_k^{(l)} \tag{2.127}$$

We also define

$$o_j = F(y^{(j)}, \mathbf{x}_i; \theta_0) \tag{2.128}$$

Using the chain rule, we have

$$\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial w_{jk}^{(l)}} = \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} \cdot \frac{\partial a_k^{(l)}}{\partial z_k^{(l)}} \cdot \frac{\partial z_k^{(l)}}{\partial w_{jk}^{(l)}} \tag{2.129}$$

$$= \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} \cdot \sigma'(z_k^{(l)}) \cdot a_j^{(l-1)} \tag{2.130}$$

and

$$\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial b_k^{(l)}} = \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} \cdot \frac{\partial a_k^{(l)}}{\partial z_k^{(l)}} \cdot \frac{\partial z_k^{(l)}}{\partial b_k^{(l)}} \tag{2.131}$$

$$= \frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} \cdot \sigma'(z_k^{(l)}) \tag{2.132}$$

where $\sigma'(z_k^{(l)})$ is the derivative of the activation function evaluated at $z_k^{(l)}$, whatever this happens to be depending on the selection of activation function.

The only remaining task is to find $\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}}$ for all $a_k^{(l)}$.

Let us work out a typical classifier example where an L-layer network $F(y_i, \mathbf{x}_i; \theta_0)$ is a multi-layer perceptron with a softmax output layer, and cross entropy loss is used. We then have

$$\ell(\mathbf{x}_i, y_i; \theta_0) = H(t_i, F(\mathbf{x}_i; \theta_0)) \tag{2.133}$$

$$= -\sum_{j=1}^{m} t_i(j) \log o_j \tag{2.134}$$

and

$$F(y^{(j)}, \mathbf{x}_i; \theta_0) = o_j = \frac{e^{a_j^{(L)}}}{\sum_{w=1}^{m} e^{a_w^{(L)}}} \tag{2.135}$$

Using equation 2.134 and the chain rule we have

$$\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} = -\sum_{j=1}^{m} t_i(j) \frac{\partial \log o_j}{\partial o_j} \cdot \frac{\partial o_j}{\partial a_k^{(l)}} \tag{2.136}$$

$$= -\sum_{j=1}^{m} t_i(j) \frac{1}{o_j} \cdot \frac{\partial o_j}{\partial a_k^{(l)}} \tag{2.137}$$

Using equation 2.135 and the quotient rule,

$$\frac{\partial o_j}{\partial a_k^{(l)}} = \frac{\frac{\partial e^{a_j^{(L)}}}{\partial a_k^{(l)}} \cdot \sum_{w=1}^{m} e^{a_w^{(L)}} - e^{a_j^{(L)}} \cdot \frac{\partial \sum_{w=1}^{m} e^{a_w^{(L)}}}{\partial a_k^{(l)}}}{\left(\sum_{w=1}^{m} e^{a_w^{(L)}}\right)^2} \tag{2.138}$$

$$= \frac{e^{a_j^{(L)}} \frac{\partial a_j^{(L)}}{\partial a_k^{(l)}} \cdot \sum_{w=1}^{m} e^{a_w^{(l)}} - e^{a_j^{(L)}} \cdot \sum_{w=1}^{m} \left(e^{a_w^{(L)}} \frac{\partial a_w^{(L)}}{\partial a_k^{(l)}}\right)}{\left(\sum_{w=1}^{m} e^{a_w^{(L)}}\right)^2} \tag{2.139}$$

$$= o_j \cdot \left[\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}} - \frac{\sum_{w=1}^{m} \left(e^{a_w^{(L)}} \frac{\partial a_w^{(L)}}{\partial a_k^{(l)}}\right)}{\sum_{w=1}^{m} e^{a_w^{(L)}}}\right] \tag{2.140}$$

Plugging equation 2.140 into equation 2.137 yields

$$\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}} = -\sum_{j=1}^{m} t_i(j) \cdot \left[\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}} - \frac{\sum_{w=1}^{m} \left(e^{a_w^{(L)}} \frac{\partial a_w^{(L)}}{\partial a_k^{(l)}}\right)}{\sum_{w=1}^{m} e^{a_w^{(L)}}}\right] \tag{2.141}$$

We now have a solution to $\frac{\partial\, \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}}$ where the only remaining unknowns after forward propagation are the terms $\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}}$.

By recursive application of the chain rule to equation 2.126 we get

$$\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}} = \sum_{s=1}^{u(L-1)} \frac{\partial a_j^{(L)}}{\partial a_s^{(L-1)}} \cdot \sum_{v=1}^{u(L-2)} \frac{\partial a_s^{(L-1)}}{\partial a_v^{(L-2)}} \cdots \sum_{w=1}^{u(l+1)} \frac{\partial a_r^{(l+2)}}{\partial a_w^{(l+1)}} \cdot \frac{\partial a_w^{(l+1)}}{\partial a_k^{(l)}} \tag{2.142}$$

where $u(l)$ is the number of neurons in layer l, and r indexes neurons in layer $l+2$.

We thus require only to find all the 'immediate' partial derivatives (of the form $\frac{\partial a_j^{(l)}}{\partial a_k^{(l-1)}}$), ie the partial derivatives of activations with respect to their counterparts in the immediately preceding layer. Applying the chain rule just once to equation 2.126 yields

$$\frac{\partial a_j^{(l)}}{\partial a_k^{(l-1)}} = \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} \tag{2.143}$$

$$= \sigma'(z_j^{(l)}) \cdot w_{kj}^{(l)} \tag{2.144}$$

We can now express $\nabla_\theta \mathcal{L}(\theta_0)$ in terms of values that are known after forward propagation (activations $a_j^{(l)}$, preactivations $z_j^{(l)}$, weights $w_{jk}^{(l)}$, biases b_j).

As a reminder, we must solve $\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}}$ for every valid permutation of (j, k, l). Inspecting equation 2.140, it becomes evident that it is more computationally efficient to solve the

immediate partials starting from the final layer ($\frac{\partial a_j^{(L)}}{\partial a_k^{(L-1)}}$) and moving backward to first the layer ($\frac{\partial a_j^{(1)}}{\partial a_k^{(0)}}$), to avoid redundant computations. This means only the right-hand partial ($\frac{\partial a_w^{(l+1)}}{\partial a_k^{(l)}}$) needs to be computed in equation 2.142, since the other immediate partials have already been computed. The 'backward' order of solving the immediate partials explains the term 'backward propagation'.

Algorithm 1 summarizes the backpropagation algorithm applied to an MLP algorithm.

Algorithm 1 Backpropagation of MLP Neural Network

procedure BACKPROP ▷ Goal of backpropagation: Find $\nabla_\theta \mathcal{L}(\theta_0)$
  Initialization:
    $\mathbf{F}(\mathbf{x}, \theta)$: MLP model, where $\mathbf{x}$ input, θ the set of learned parameters
    θ_0: current set of learned parameters
    $\mathbf{x}_1^n := (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n)$: training input dataset
    $y_1^n := (y_1, y_2, ..., y_n)$: training output dataset
    $\ell(\mathbf{x}_i, y_i; \theta) = -\sum_{j=1}^m t_i(j) \log o_j$: loss function
    $\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^n \ell(\mathbf{x}_i, y_i; \theta)$: overall lost function
    L: Number of layers in $\mathbf{F}(\mathbf{x}, \theta)$
    $u(l)$: Number of neurons in layer l
    $a_j^{(l)}$: Activation of neuron j in layer l
    $z_j^{(l)}$: Prectivation of neuron j in layer l
    $w_{jk}^{(l)}$: Weight connecting neuron j of layer $l-1$ to neuron k of layer l
    $b_k^{(l)}$: Bias of neuron k in layer l
  Main:
    for $i \leftarrow 1$ to n **do** ▷ Iterate through training samples
      Perform forward-propagation to get all $a_j^{(l)}$, $z_j^{(l)}$, $w_{jk}^{(l)}$, $b_k^{(l)}$
      for $l \leftarrow L$ to 1 **do** ▷ Iterate through layers in reverse order
        for $j \leftarrow 1$ to m **do** ▷ Iterate through all possible labels
          for $k \leftarrow 1$ to $u(l)$ **do** ▷ Iterate through neurons in layer l
            Compute $\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}}$ using eq 2.142 and all previous $\frac{\partial a_\alpha^{(r+1)}}{\partial a_\beta^{(r)}}$ (where $r \geq l$)
            for $s \leftarrow 1$ to $u(l-1)$ **do** ▷ Iterate through neurons in layer $l-1$
              Compute $\frac{\partial a_k^{(l)}}{\partial a_s^{(l-1)}}$ using eq 2.144
        Use eq 2.141 and all $\frac{\partial a_j^{(L)}}{\partial a_k^{(l)}}$ to compute all $\frac{\partial \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial a_k^{(l)}}$
        Use eqs 2.130 and 2.132 to compute $\frac{\partial \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial w_{jk}^{(l)}}$, $\frac{\partial \ell(\mathbf{x}_i, y_i; \theta_0)}{\partial b_k^{(l)}}$

2.9 Evaluation of Learned Model: Cross-Validation

As established in section 2.3, the learning goal is to minimize population risk:

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{(\mathbf{x},y)\sim S}\, \ell\Big((\mathbf{x}, y); \theta\Big) \tag{2.145}$$

But since we only have a subset of the population to train the model, we minimize the loss over the training samples instead

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{(\mathbf{x},y)\sim D}\, \ell\Big((\mathbf{x}, y); \theta\Big) \tag{2.146}$$

However, due to the problem of overfitting, a low expected loss over the training sequence (the minimized term in equation 2.146) does not necessarily guarantee a low population risk. In other words, the learned model may perform poorly over unseen samples $(\mathbf{x}, y) \in S$. To evaluate the model, a common practice is to split the training dataset $\mathcal{S}$ into a two sequences $\mathcal{T}$ and $\mathcal{V}$[33]. The goal of cross-validation is to train the model on the sequence $\mathcal{T}$ and evaluate the model on the validation sequence $\mathcal{V}$. Formally, let $\mathcal{T} = t_1^a := (t_1, t_2, ...t_a)$ and $\mathcal{V} = v_1^b := (v_1, v_2, ...v_b)$, where a,b are integers and $a + b = n$.

The samples t_i are drawn at random without replacement from $\mathcal{S}$[13]. Once the samples t_i are drawn, samples v_i are drawn in the same fashion from the remaining unpicked samples in $\mathcal{S}$. We define the in-sample error $\mathbb{E}_{\mathcal{T}}$ as the expected loss over the sequence $\mathcal{T}$:

$$\mathbb{E}_{\mathcal{T}} := \frac{1}{a}\sum_{i=1}^{a} \ell\Big(t_i; \theta\Big) \tag{2.147}$$

And the out-of-sample error $\mathbb{E}_{\mathcal{V}}$ as the expected loss over the sequence $\mathcal{V}$:

$$\mathbb{E}_{\mathcal{V}} := \frac{1}{b}\sum_{i=1}^{b} \ell\Big(v_i; \theta\Big) \tag{2.148}$$

Cross-validation entails minimizing the in-sample error, and validating the model's performance by comparing against the in-sample error to out-of-sample error. The training goal under cross-validation is thus

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{\mathcal{T}} := \underset{\theta}{\operatorname{argmin}} \sum_{i=1}^{a} \ell\Big(t_i; \theta\Big) \tag{2.149}$$

Since the validation samples are unseen by the model, they are taken as a proxy for the population S, and $\mathbb{E}_{\mathcal{V}}$ is a proxy for the expected population loss of equation 2.145. Since minimizing the latter is our true learning goal, a good model should therefore have a low out-of-sample error, ideally $\mathbb{E}_{\mathcal{V}} = 0$. Furthermore, a model should generalize well, so we desire $\mathbb{E}_{\mathcal{T}} \approx \mathbb{E}_{\mathcal{V}}$.

[13] Since $\mathcal{S}$ is a sequence and not a set, this means that for each i, a number j is picked without replacement from the set $[1..n]$ and we let $t_i = s_j$.

Chapter 3

MixUp as DAT

3.1 Preliminaries

Classification Setting

Consider a classification problem similar to that described in section 2.2.5. The training sequence $\mathcal{S}$ and training input sequence $\mathcal{X}$ are as defined in equations 2.1 and 2.2, respectively. The training output sequence $\mathcal{Y}$ is defined as in equation 2.3 with the distinction that samples $\mathbf{y}_i$ are vectors.

We assume that training samples s_i are realizations of some random variable S, input samples $\mathbf{x}_i$ are realizations of a random variable $\mathbf{X}$, and output samples $\mathbf{y}_i$ are realizations of a random variable $\mathbf{Y}$. Let $\mathbb{S}$ be the space in which the training samples s_i live, let $\mathbb{X}$ be the vector space in which the input samples $\mathbf{x}_i$ live and let $\mathbb{Y}$ be the space in which the output samples $\mathbf{y}_i$ live; that is $\mathbb{S}$, $\mathbb{X}$, $\mathbb{Y}$ are the support of S, $\mathbf{X}$, $\mathbf{Y}$, respectively. Finally, let $\mathbb{S}_D$ be the set of unique input samples in $\mathcal{S}$, such that $\mathbb{S}_D \subseteq \mathbb{S}$; $\mathbb{X}_D$ be the set of unique input samples in $\mathcal{X}$, such that $\mathbb{X}_D \subseteq \mathbb{X}$; and $\mathbb{Y}_D$ be the set of unique input samples in $\mathcal{Y}$ such that $\mathbb{Y}_D \subseteq \mathbb{Y}$.

Let $\mathbf{F}$ be a neural network function, parameterized by θ, which maps $\mathbb{X}$ to $\mathbb{Y}$.

In the space $\mathbb{Y}$, we refer to $\mathbf{F}(\mathbf{x};\theta)$ as the *model's prediction.*Let $\ell : \mathbb{X} \times \mathbb{Y} \to \mathbb{R}$ be a *loss function*, using which we define an *overall loss function* as

$$\mathcal{L}(\theta) := \frac{1}{n}\sum_{i=1}^{n} \ell\big((\mathbf{x}_i, \mathbf{y}_i); \theta\big) \tag{3.1}$$

The learning goal is to minimize $\mathcal{L}$ with respect to its characterizing parameters θ:

$$\theta^{ERM} = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta) \tag{3.2}$$

Label Mapping In a typical problem, the sequence of output samples is given as

$$\mathcal{Z} = z_1^n := (z_1, z_2, ..., z_n) \tag{3.3}$$

where samples $z_i \in \mathbb{Z}$ and $\mathbb{Z} := \{z^{(j)} : j = 1, 2, ..., m\}$; in other words, output samples are scalar labels. One can easily map the labels z_i to a vector $\mathbf{y}_i$ using some label mapping function $\boldsymbol{\varphi}$. Let $\boldsymbol{\varphi} : \mathbb{Z} \to \mathbb{Y}$ map a scalar label in $\mathbb{Z}$ to an element in $\mathbb{Y}$ such that for any $z, z' \in \mathbb{Z}$, if $z \neq z'$, then $\boldsymbol{\varphi}(z) \neq \boldsymbol{\varphi}(z')$. The specific expression for $\boldsymbol{\varphi}$ depends on the loss function used to train $\mathbf{F}(\mathbf{x}; \theta)$. Examples of $\boldsymbol{\varphi}$ will be given for the two loss functions discussed hereafter. Note that examples of $\boldsymbol{\varphi}$ are provided for instruction; this section assumes that mapping from $\mathbb{Z}$ to $\mathbb{Y}$ has already taken place, such that the training label sequence is $\mathcal{Y}$, not $\mathcal{Z}$.

Target-Linear Loss Functions We say that a loss function $\ell\big((\mathbf{x}, \mathbf{y}); \theta\big)$ is *target-linear* if for any scalars α and β,

$$\ell\big((\mathbf{x}, \alpha\mathbf{y}_1 + \beta\mathbf{y}_2); \theta\big) = \alpha\ell\big((\mathbf{x}, \mathbf{y}_1); \theta\big) + \beta\ell\big((\mathbf{x}, \mathbf{y}_2); \theta\big) \tag{3.4}$$

Target-linear loss functions arise naturally in many settings, for which we now provide two examples. For convenience, we define the vector $\mathbf{v} = \mathbf{F}(\mathbf{x}; \theta)$.

Cross-Entropy Loss The conventional cross-entropy loss function, written in our notation, is defined as:

$$\ell_{\text{CE}}\big((\mathbf{x}, \mathbf{y}); \theta\big) := \sum_{j=1}^{dim(\mathbb{Y})} \mathbf{y}[j] \log \mathbf{v}[j] \tag{3.5}$$

where $\mathbf{v}$ and $\mathbf{y}$ are constrained to being probability vectors. In typical problems where $z \in \mathbb{Z}$ the label mapping function is defined such that the target label $\mathbf{y}$ is a one-hot vector: specifically $\mathbf{y}[j] = \boldsymbol{\varphi}(\boldsymbol{z})[j] = \mathbb{1}\{z^{(j)} = z_i\}$. Constraining $\mathbf{v}$ to being a probability vector is achieved using a softmax output layer. Note that, as shown in section 2.2.5, overall loss function constructed using cross-entropy loss may be interpreted as performing ML and MAP optimizations.

Negative-Cosine Loss The "negative-cosine loss", so called due to its resemblance to cosine similarity[1], is defined as:

$$\ell_{\text{NC}}\big((\mathbf{x}, \mathbf{y}); \theta\big) := -\frac{\mathbf{v}^{\text{T}}\mathbf{y}}{||\mathbf{v}||} \tag{3.7}$$

We ensure instances of $\mathbf{y}$ are sufficiently separated by limiting the range of $\boldsymbol{\varphi}$ to an orthonormal basis, making it a conventional label embedding function. Specifically, for any $z, z' \in \mathbb{Z}$, if $z \neq z'$, then $\boldsymbol{\varphi}(z) \perp\!\!\!\perp \boldsymbol{\varphi}(z')$ and for all z $||\boldsymbol{\varphi}(z)|| = 1$.

The cross-entropy loss ℓ_{CE} and the negative-cosine loss ℓ_{NC} are both target-linear. Proof

[1] Cosine similarity is defined as CITGUO COSINE SIMILARITY:

$$f(\mathbf{v}, \mathbf{y})) := \frac{\mathbf{v}^{\text{T}}\mathbf{y}}{||\mathbf{v}||\,||\mathbf{y}||} \tag{3.6}$$

Since we chose an orthonormal basis, all training (unmixed) labels have $||y|| = 1$, such that for conventional training without use of label mixing, negative-cosine loss is strictly the negative of cosine-similarity.

of ℓ_{CE} target-linearity:

$$\ell_{\text{CE}}\big((\mathbf{x}, \alpha\mathbf{y}_1 + \beta\mathbf{y}_2); \theta\big) = \sum_{j=1}^{dim(\mathcal{Z})} (\alpha\mathbf{y}_1[j] + \beta\mathbf{y}_2[j]) \log \mathbf{v}[j] \tag{3.8}$$

$$= \alpha \sum_{j=1}^{dim(\mathcal{Z})} \mathbf{y}_1[j] \log \mathbf{v}[j] + \beta \sum_{j=1}^{dim(\mathcal{Z})} \mathbf{y}_2[j] \log \mathbf{v}[j] \tag{3.9}$$

$$= \alpha\ell_{\text{CE}}\big((\mathbf{x}, \mathbf{y}_1); \theta\big) + \beta\ell_{\text{CE}}\big((\mathbf{x}, \mathbf{y}_2); \theta\big) \tag{3.10}$$

Proof of ℓ_{NC} target-linearity:

$$\ell_{\text{NC}}\big((\mathbf{x}, \alpha\mathbf{y}_1 + \beta\mathbf{y}_2); \theta\big) = -\frac{\mathbf{v}^{\text{T}}(\alpha\mathbf{y}_1 + \beta\mathbf{y}_2)}{||\mathbf{v}||} \tag{3.11}$$

$$= -\frac{\alpha\mathbf{v}^{\text{T}}\mathbf{y}_1}{||\mathbf{v}||} - \frac{\beta\mathbf{y}^{\text{T}}\mathbf{y}_2}{||\mathbf{v}||} \tag{3.12}$$

$$= \alpha\ell_{\text{NC}}\big((\mathbf{x}, \mathbf{y}_1); \theta\big) + \beta\ell_{\text{NC}}\big((\mathbf{x}, \mathbf{y}_2); \theta\big) \tag{3.13}$$

Assumptions The theoretical development of this section relies on two fundamental assumptions, which we call "axioms".

Axiom 1 *(Target linearity) The loss function ℓ used for the classification setting is target-linear.*

That is, the study of MixUp in this thesis in fact goes beyond the standard MixUp, which uses the cross-entropy loss.

Much of the development in this chapter concerns drawing sample pairs (ν, ν') from $\mathbb{S}_D \times \mathbb{S}_D$. Suppose that $(\nu, \nu')_1^r$ is a length-r sequence of sample pairs drawn from $\mathbb{S}_D \times \mathbb{S}_D$. A sequence $(\nu, \nu')_1^r$ is said to be symmetric if for every $(a, b) \in \mathbb{S}_D \times \mathbb{S}_D$, the number of occurrences of (a, b) in the sequence is equal to that of (b, a). A distribution Q on $\mathbb{S}_D \times \mathbb{S}_D$ will be called *exchangeable*, or *symmetric*, if for any $(\nu, \nu') \in \mathbb{S}_D \times \mathbb{S}_D$, $Q((\nu, \nu')) = Q((\nu', \nu))$.

Axiom 2 *(Symmetric pair-sampling distribution) Whenever a sample pair (ν, ν') is drawn from a distribution Q, Q is assumed to be symmetric.*

In the standard MixUp discussed in section 2.5.3, two samples are drawn independently from $\mathbb{S}_D$ to form a pair, satisfying this condition.

3.2 MixUp, DAT, Untied Mixup

Let (ν, ν') be drawn from Q, where $\nu = (\mathbf{x}, \mathbf{y})$ and $\nu' = (\mathbf{x}', \mathbf{y}')$.

3.2.1 MixUp

As discussed in section 2.5.3, MixUp is a data augmentation scheme in which samples are linearly combined using some mixing ratio $\lambda \in [0, 1]$:

$$\mathbf{x}_g = \lambda \mathbf{x} + (1 - \lambda)\mathbf{x}' \tag{3.14}$$

where $\lambda \sim P^{\text{Mix}}$. A target label is generated using the same mixing ratio λ:

$$\mathbf{y}_g = \lambda \mathbf{y} + (1 - \lambda)\mathbf{y}' \tag{MixUp}$$

The loss function for the MixUp scheme is

$$\ell^{\text{Mix}}\big((\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}'), \lambda; \theta\big) := \ell_b\big((\lambda \mathbf{x} + \overline{\lambda}\mathbf{x}', \lambda \mathbf{y} + \overline{\lambda}\mathbf{y}'); \theta\big) \tag{3.15}$$

where ℓ_b is the baseline loss function.

In MixUp, we refer to P^{Mix} as the *mixing policy*.

See section section 2.5.3 for a more complete discussion of MixUp with illustrated example.

3.2.2 Directional Adversarial Training (DAT)

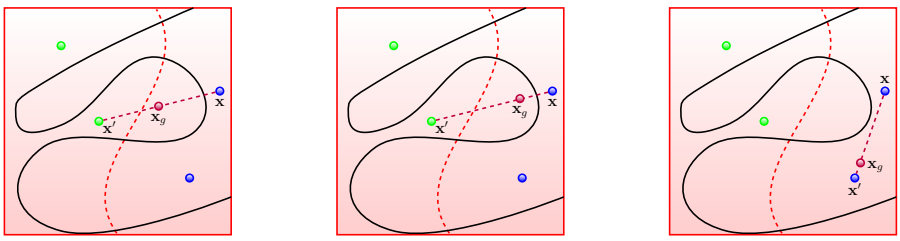
DAT uses the same method as MixUp (described by equation 3.14) for generating input samples, with the distinction that $\lambda \sim P^{\text{DAT}}$, where P^{DAT} may be chosen to be different than P^{Mix}. DAT differs from MixUp in its generated target labels. Unlike MixUp, DAT retains the sample's original label:

$$\mathbf{y}_g = \mathbf{y} \tag{DAT}$$

The loss function for the DAT scheme is

$$\ell^{\text{DAT}}\big((\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}'), \lambda; \theta\big) := \ell_b\big((\lambda \mathbf{x} + \overline{\lambda}\mathbf{x}', \mathbf{y}); \theta\big) \tag{3.16}$$

Since DAT uses the same input mixing procedure as MixUp, we repeat figure 2.6 for convenience. See section 2.5.3 for an explanation of the illustration.



(a) Inter-class $\lambda = 0.5$ (b) Inter-class $\lambda = 0.8$ (c) Intra-class $\lambda = 0.9$

Figure 3.1: MixUp, DAT, UMixUp input sample mixing

Unlike the analogous MixUp example in section 2.5.3, $\mathbf{y}_g = \mathbf{y}$ for all examples (a,b,c). That is, the label remains unchanged despite movement of the input sample from $\mathbf{x}$ to $\mathbf{x}'$.

Intuitively, we expect P^{DAT} distributions to be a monotonically increasing, such that $P^{\text{DAT}}(0)$ is small and $P^{\text{DAT}}(1)$. This fits with the notion of DAT as a form of adversarial training (see section 3.3). By augmenting the dataset with a nearby samples retaining the same label (ie, by having a high probability of picking λ close to 1), DAT serves to minimizing overfitting.

3.2.3 Untied MixUp

UMixUp is a superset of MixUp which also uses the same mixing procedure (equation 3.14) as MixUp for generating input samples, with $\lambda \sim P^{\text{uMix}}$. UMixUp differs from both MixUp and DAT in its target labels.

UMixUp's label mixing ratio is a function γ of λ:

$$\mathbf{y}_g = \gamma(\lambda)\mathbf{y} + (1 - \gamma(\lambda))\mathbf{y}' \tag{UMixUp}$$

where γ maps $[0, 1]$ to $[0, 1]$. The loss function for the UMixUp scheme is

$$\ell^{\text{uMix}}\big((\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}'), \lambda; \theta\big) := \ell_b\big((\lambda\mathbf{x} + \overline{\lambda}\mathbf{x}', \gamma(\lambda)\mathbf{y} + \overline{\gamma(\lambda)}\mathbf{y}'); \theta\big) \tag{3.17}$$

where ℓ_b is the baseline loss function. As evidenced by the schemes' respective loss functions, MixUp is a special case of UMixUp where $\gamma(\lambda) = \lambda$, as shown in figure 3.2.

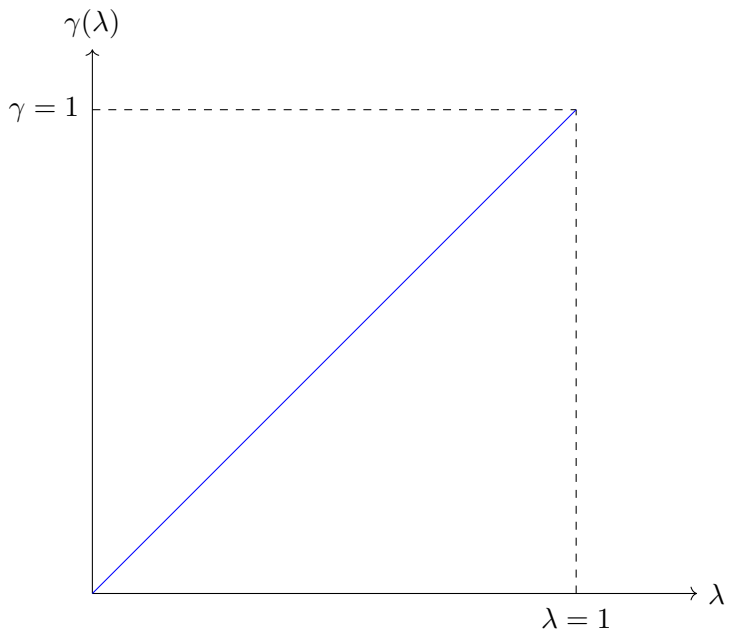


Figure 3.2: Curve of γ for special case of MixUp ($\gamma(\lambda) = \lambda$)

If we assume that the dataset is correctly labeled and that sample pairs are drawn from a symmetric distribution, then it is reasonable to expect the γ function to possess the following properties:

1. $\gamma(0) = 0$ and $\gamma(1) = 1$: this is consistent with our assumption that the dataset is correctly labeled. In other words, we expect an input sample $\mathbf{x}$ to retain its original label $\mathbf{y}$ when $\mathbf{x}$ is not moved.

2. $\gamma(\lambda)$ is a monotonically increasing function: as we move from an input sample $\mathbf{x}$ to an input sample $\mathbf{x}'$ (for example, along the dashed line in figure 3.1a), we make the assumption that we traverse only one class boundary. Thus, one expects that the interpolated points are progressively classified less strongly as $\mathbf{y}$ and more strongly as $\mathbf{y}'$. While it is possible to pick two input samples $\mathbf{x}$ and $\mathbf{x}'$ in which the interpolated line traverses multiple class boundaries, it is highly unlikely that assuming such class crossings for all interpolations in $\mathbb{X}_D \times \mathbb{X}_D$[2] would be beneficial for regularization.

3. $\gamma(1-\lambda) = 1 - \gamma(\lambda)$: this is consistent with our assumption that sample pairs are drawn from a symmetric distribution. Sample pair (ν, ν') is equally likely to be drawn as (ν', ν). Let the sample generated using (ν, ν') be $(\mathbf{x}_a, \mathbf{y}_a)$ and the sample generated using (ν', ν) be $(\mathbf{x}_b, \mathbf{y}_b)$, so that

$$\mathbf{x}_a = \lambda_a \mathbf{x} + (1 - \lambda_a)\mathbf{x}' \tag{3.18}$$
$$\mathbf{x}_b = \lambda_b \mathbf{x}' + (1 - \lambda_b)\mathbf{x} \tag{3.19}$$
$$\mathbf{y}_a = \gamma(\lambda_a)\mathbf{y} + (1 - \gamma(\lambda_a))\mathbf{y}' \tag{3.20}$$
$$\mathbf{y}_b = \gamma(\lambda_b)\mathbf{y}' + (1 - \gamma(\lambda_b))\mathbf{y} \tag{3.21}$$

   where λ_a is the mixing ratio drawn when mixing (ν, ν') and λ_b is the mixing ratio drawn when mixing (ν', ν). Suppose now that by random chance, $\lambda_b = 1 - \lambda_a$, yielding $\mathbf{x}_a = \mathbf{x}_b$. It is reasonable to expect that the output samples be equal, ie $\mathbf{y}_a = \mathbf{y}_b$. Otherwise, this would result in unnecessary gradient stochasticity when performing gradient descent[3].

   Thus we expect $\gamma(\lambda_a) = 1 - \gamma(\lambda_b) = 1 - \gamma(1 - \lambda_a)$.

   Note that the cumulative density function (CDF) of any distribution whose probability density function (PDF) is symmetric around the vertical line $\lambda = 0.5$ satisfies this property[4]. Thus any the CDF of a Beta distribution having the form $B(\alpha, \alpha)$ satisfies this property, where $\alpha > 0$.

We note that the standard MixUp scheme's label mixing mechanism (where $\gamma(\lambda) = \lambda$) satisfies all three properties.

[2]Remember, we use the same γ function to indiscriminately mix all sample pairs

[3]The model would otherwise train on a sample $(\mathbf{x}_a, \mathbf{y}_a)$ and later on $(\mathbf{x}_a, \mathbf{y}_b)$ where $\mathbf{y}_a \neq \mathbf{y}_b$ introducing gradient stochasticity at the input $\mathbf{x}_a$.

[4]See appendix A.3 for a proof

To illustrate UMixUp, we use the same mixing examples shown in figure 3.1:

$$\text{a)}\ \ \mathbf{x}_g = 0.5\mathbf{x} + 0.5\mathbf{x}' \qquad \text{b)}\ \ \mathbf{x}_g = 0.8\mathbf{x} + 0.2\mathbf{x}' \qquad \text{c)}\ \ \mathbf{x}_g = 0.9\mathbf{x} + 0.1\mathbf{x}'$$

where $\mathbf{x}$ and $\mathbf{x}'$ belong to different classes in examples a and c (blue and green respectively), and belong to the same class in example c (blue).

To illustrate UMixUp's label mixing procedure in the context of figure 3.1, let us suppose that $\gamma(\lambda)$ is the CDF of $B(0.4, 0.4)$, ie

$$\gamma(\lambda) = I_\lambda(0.4, 0.4) \tag{3.22}$$

where $I_\lambda(a, b)$ is the regularized incomplete beta function. The curve of $I_\lambda(0.4, 0.4)$ is shown in figure 3.3.

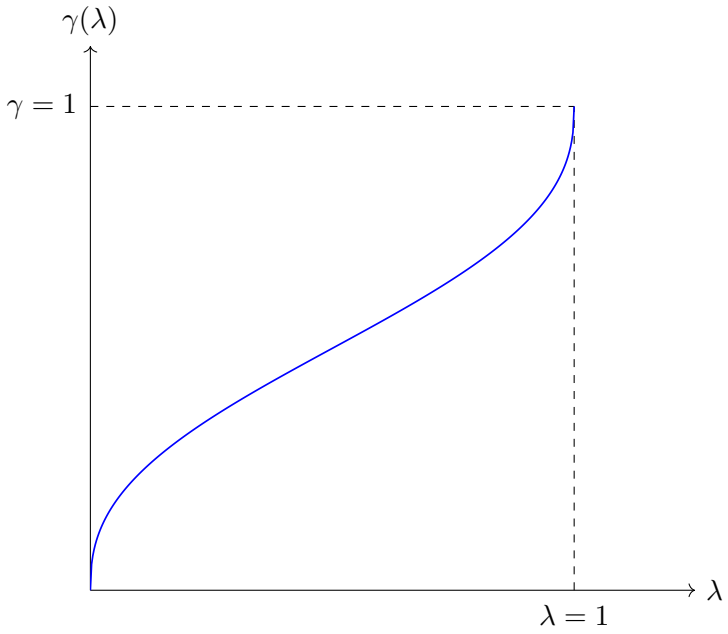


Figure 3.3: Curve of $\gamma(\lambda) = I_\lambda(0.4, 0.4)$

Suppose, as in section 2.5.3, that the blue label $\mathbf{y}_u$ and green label $\mathbf{y}_a$ correspond respectively to

$$\mathbf{y}_u = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad\qquad \mathbf{y}_a = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Then $\mathbf{y} = \mathbf{y}_u$ in all subfigures of figure 3.1. In figures 3.1a and 3.1b $\mathbf{y}' = \mathbf{y}_a$, while in figure 3.1c $\mathbf{y}' = \mathbf{y}_b$.

The resulting mixed labels $\mathbf{y}_g$ are then

$$\text{a)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 0.5 \\ 0 \\ 0 \\ 0.5 \end{bmatrix} \qquad \text{b)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 0.75 \\ 0 \\ 0 \\ 0.25 \end{bmatrix} \qquad \text{c)}\ \ \mathbf{y}_g = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Intra-class mixing example (as in example c) results in no movement of the label; this is no different than standard MixUp. Inter-class mixing (as in examples a and b) may result in a different generated label. In example a, the generated target label $\mathbf{y}_g$ is the same as MixUp. In example b, however, $\mathbf{y}_g$ is more aggressively regularized: while the generated input sample $\mathbf{x}_g$ has only moved by 0.2[5], the label has moved by 0.25[6]. The regularization effect in example b will be a stronger local movement of the overfitted classifier boundary toward $\mathbf{x}'$ than in standard MixUp.

In MixUp, the label mixing ratio is always equal (or 'tied') to the input mixing ratio, ie $\gamma(\lambda) = \lambda$. In Untied MixUp, the label mixing ratio is 'untied' from the sample mixing ratio, such that we may have $\gamma(\lambda) \neq \lambda$. We will refer to γ as the *weighting function*. An Untied MixUp scheme is specified both by the its mixing policy P^{uMix} and a weighting function γ.

3.2.4 Overall Loss Functions

To draw comparisons between MixUp, DAT, and Untied MixUp schemes, we establish a framework for characterizing their optimization problems. To that end, we define the models' respective overall loss functions $\mathcal{L}^{\text{Mix}}$, $\mathcal{L}^{\text{DAT}}$, $\mathcal{L}^{\text{uMix}}$, in terms of their loss functions (which have been defined in equations 2.106, 3.16, 3.17, respectively). We denote the expected value of a scheme's overall loss, $\mathcal{L}_E$, with respect to its mixing ratio Λ, and we give expressions for the expected overall losses $\mathcal{L}_E^{\text{Mix}}$, $\mathcal{L}_E^{\text{DAT}}$, $\mathcal{L}_E^{\text{uMix}}$. Two schemes are equivalent when their overall loss functions are equal, and we will later ascertain under which conditions such equivalence occurs.

Let r be a positive integer. In every scheme, a sequence

$$(\nu, \nu')_1^r := ((\nu_1, \nu_1'), (\nu_2, \nu_2'), \ldots, (\nu_r, \nu_r'))$$

of sample pairs is drawn i.i.d. from Q.

MixUp A sequence $\lambda_1^r := (\lambda_1, \lambda_2, \ldots, \lambda_r)$ of values is drawn i.i.d. from P^{Mix}. We denote the overall loss $\mathcal{L}^{\text{Mix}}\left((\nu, \nu')_1^r, \lambda_1^r\right)$ and the expected overall loss $\mathcal{L}_E^{\text{Mix}}\left((\nu, \nu')_1^r\right)$:

$$\mathcal{L}^{\text{Mix}}\left((\nu, \nu')_1^r, \lambda_1^r; \theta\right) := \frac{1}{r} \sum_{k=1}^{r} \ell^{\text{Mix}}(\nu_k, \nu_k', \lambda_k) \tag{3.23}$$

$$\mathcal{L}_E^{\text{Mix}}\left((\nu, \nu')_1^r; \theta\right) := \mathbb{E}_{\lambda_1^r \overset{\text{iid}}{\sim} P^{\text{Mix}}} \mathcal{L}^{\text{Mix}}\left((\nu, \nu')_1^r, \lambda_1^r; \theta\right) \tag{3.24}$$

[5] of the distance between $\mathbf{x}$ and $\mathbf{x}'$, starting from $\mathbf{x}$

[6] of the distance between $\mathbf{y}$ and $\mathbf{y}'$, starting from $\mathbf{y}$

Directional Adversarial Training (DAT) A sequence $\lambda_1^r := (\lambda_1, \lambda_2, \ldots, \lambda_r)$ of values is drawn i.i.d. from P^{DAT}. We denote the overall loss $\mathcal{L}^{\text{DAT}}\left((\nu, \nu')_1^r, \lambda_1^r\right)$ and the expected overall loss $\mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r\right)$:

$$\mathcal{L}^{\text{DAT}}\left((\nu, \nu')_1^r, \lambda_1^r; \theta\right) := \frac{1}{r}\sum_{k=1}^{r} \ell^{\text{DAT}}\left(\nu_k, \nu_k', \lambda_k\right) \tag{3.25}$$

$$\mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right) := \mathbb{E}_{\lambda_1^r \overset{\text{iid}}{\sim} P^{\text{DAT}}} \mathcal{L}^{\text{DAT}}\left((\nu, \nu')_1^r, \lambda_1^r; \theta\right) \tag{3.26}$$

In DAT, we refer to P^{DAT} as the *adversarial policy*.

Untied MixUp (UMixUp)

A sequence $\lambda_1^r := (\lambda_1, \lambda_2, \ldots, \lambda_r)$ of values is drawn i.i.d. from P^{uMix}. We denote the overall and expected overall loss functions $\mathcal{L}^{\text{uMix}}\left((\nu, \nu')_1^r, \lambda_1^r, \gamma\right)$ and $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma\right)$ respectively:

$$\mathcal{L}^{\text{uMix}}\left((\nu, \nu')_1^r, \lambda_1^r, \gamma; \theta\right) := \frac{1}{r}\sum_{k=1}^{r} \ell^{\text{uMix}}(\nu_k, \nu_k', \lambda, \gamma; \theta) \tag{3.27}$$

$$\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma; \theta\right) := \mathbb{E}_{\lambda_1^r \overset{\text{iid}}{\sim} P^{\text{uMix}}} \mathcal{L}^{\text{uMix}}\left((\nu, \nu')_1^r, \lambda_1^r, \gamma; \theta\right) \tag{3.28}$$

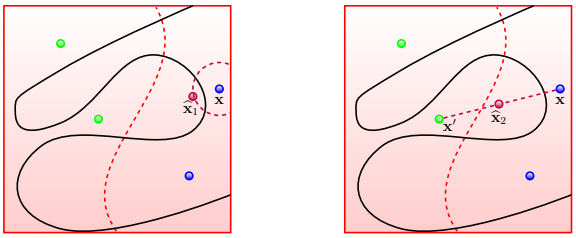
3.3 DAT as Analogous to Adversarial Training

Both MixUp and UMixUp will be shown to converge to DAT as the number of mixed sample pairs, r, tends to infinity. Prior to developing this result, we provide insight into DAT, in terms of its similarity to adversarial training.

As established in section 2.5.2, conventional adversarial training schemes augment the original training dataset by searching for approximations of true adversarials within bounded regions around each training sample.

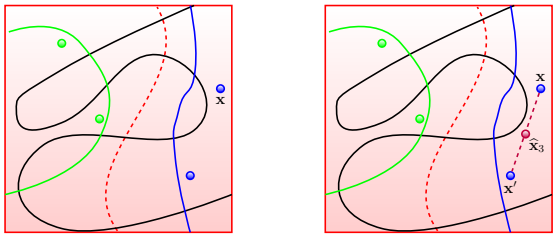
DAT, on the other hand, combines intra-class mixing (mixing two samples of the same class) and inter-class mixing (mixing samples of different classes). Intra-class mixing serves to learn classification boundaries in a fashion similar to unregularized (aka baseline) training: generated input samples generally reside within the class region. Inter-class mixing, on the other hand, perturbs training samples in the direction of adversarial classes. Similar to adversarial training, interclass mixing moves a training sample away from its original class, while retaining its original label.

Figures 3.4 and 3.5 illustrate the connection between standard adversarial training and DAT. Consider, as in previous examples, the problem of classifying the blue points and the green points, where the dashed curve is a ground-truth classifier and the black curve indicates the classification boundary of $\mathbf{F}(\mathbf{x}; \theta)$, which overfits the training data. In adversarial training, a training sample $\mathbf{x}$ is moved to a location within an L_p-ball around $\mathbf{x}$ while keeping its label to further train the model; the location, denoted by $\widehat{\mathbf{x}}_1$ in figure 3.4b, is chosen to maximize training loss.



(a) Adversarial training (b) Inter-class mixing

Figure 3.4: Interclass mixing as a form of adversarial training



(a) Baseline training (b) Intra-class mixing

Figure 3.5: Intraclass mixing as a form of baseline training

In DAT, a second sample $\mathbf{x}'$ governs the direction in which $\mathbf{x}$ is perturbed. If $\mathbf{x}'$ is chosen from a different class (aka inter-class mixing) as shown in figure 3.4b, then the generated sample $\widehat{\mathbf{x}}_2$ is used to further train the model. In both adversarial training and inter-class mixing, the training sample $\mathbf{x}$ is moved in an adversarial direction and the original label is retained.

In baseline training as shown in 3.5a, the training sample $\mathbf{x}$ is directly used to training the model. In DAT, if $\mathbf{x}'$ is chosen from the same class (aka intra-class mixing) as shown in figure 3.5b, then the sample $\widehat{\mathbf{x}}_3$ is used in further training. In both baseline training and inter-class training, generated samples usually reside inside the class boundary. The green and blue class boundaries are shown in figure 3.5 to make this intuition more evident.

Inter-class mixing dwarves intra-class mixing by volume of generated samples seen by the learning model in most many-class learning problems (by a 9-1 ratio in balanced 10-class problems for instance). DAT, which primarily consists of inter-class mixing, can therefore be seen as analogous to adversarial training. The key distinction between conventional adversarial training and inter-class mixing is that MixUp movement is determined probabilistically within a bounded region, while adversarial movement is at a defined location within a bounded region (namely, the location which which maximizes loss).

3.4 Untied MixUp as Asymptotically Equivalent to DAT

We now show that Untied MixUp and DAT are equivalent when r tends to infinity. A consequence of this equivalence is that it infuses both MixUp and UMixUp with the intuition of adversarial training. To that end, we relate the Untied MixUp loss function, ℓ^{uMix}, with the DAT loss function, ℓ^{DAT}.

Lemma 3 *For any $(\nu, \nu') \in \mathbb{S}_D \times \mathbb{S}_D$ and any $\lambda \in [0,1]$,*

$$\ell^{\text{uMix}}(\nu, \nu', \lambda, \gamma; \theta) = \gamma(\lambda)\ell^{\text{DAT}}(\nu, \nu', \lambda; \theta) + \overline{\gamma(\lambda)}\ell^{\text{DAT}}(\nu', \nu, \overline{\lambda}; \theta)$$

This result follows directly from the target-linearity of the loss function. Without loss of generality, let $\nu = (\mathbf{x}, \mathbf{y})$, $\nu = (\mathbf{x}', \mathbf{y}')$, then

$$\begin{aligned}
\ell^{\text{uMix}}(\nu, \nu', \lambda, \gamma; \theta) &= \ell^{\text{uMix}}\big((\mathbf{x}, \mathbf{y}), (\mathbf{x}', \mathbf{y}'), \lambda; \theta\big) && (3.29)\\
&:= \ell_b\big((\lambda\mathbf{x} + \overline{\lambda}\mathbf{x}', \gamma(\lambda)\mathbf{y} + \overline{\gamma(\lambda)}\mathbf{y}'); \theta\big) && (3.30)\\
&= \gamma(\lambda)\ell_b\big((\lambda\mathbf{x} + \overline{\lambda}\mathbf{x}', \mathbf{y}); \theta\big) + \overline{\gamma(\lambda)}\ell_b\big((\lambda\mathbf{x} + \overline{\lambda}\mathbf{x}', \mathbf{y}'); \theta\big) && (3.31)\\
&= \gamma(\lambda)\ell^{\text{DAT}}(\nu, \nu', \lambda; \theta) + \overline{\gamma(\lambda)}\ell^{\text{DAT}}(\nu', \nu, \overline{\lambda}; \theta) && (3.32)
\end{aligned}$$

The next two lemmas show that as r tends to infinity, the overall loss of both DAT and UMixUp converge in probability to their respective overall expected losses.

Lemma 4 *As r increases, $\mathcal{L}^{\text{DAT}}\left((\nu, \nu')_1^r, \Lambda_1^r; \theta\right)$ converges to $\mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right)$ in probability.*

Lemma 5 *As r increases, $\mathcal{L}^{\text{uMix}}\left((\nu, \nu')_1^r, \Lambda_1^r, \gamma; \theta\right)$ converges to $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma; \theta\right)$ in probability.*

These two lemmas have similar proofs, thus only the proof of Lemma 4 is given in appendix A.6.

Next we show that as r tends to infinity, UMixUp converges in probability to a subset of DAT, and DAT converges in probability to a subset of UMixUp. In other words, we show that as r increases, UMixUp converges to being equivalent to the entire class of DAT schemes.

For that purpose, let $\mathbb{F}$ denote the space of all functions mapping $[0,1]$ to $[0,1]$, and let $\mathbb{P}$ be the space of all distributions over $[0,1]$. Each configuration in $\mathbb{P} \times \mathbb{F}$ defines an Untied MixUp scheme.

We now define $\mathcal{U}$, which maps a DAT scheme to an Untied MixUp scheme. Specifically $\mathcal{U}$ is a map from $\mathbb{P}$ to $\mathbb{P} \times \mathbb{F}$ such that for any $p \in \mathbb{P}$, $\mathcal{U}(p)$ is a configuration $(p', g) \in \mathbb{P} \times \mathbb{F}$, where

$$p'(\lambda) := \frac{1}{2}\left(p(\lambda) + p(1-\lambda)\right) \text{ and } g(\lambda) := \frac{p(\lambda)}{p(\lambda) + p(1-\lambda)} \tag{3.33}$$

Lemma 6 *Let $(\nu, \nu')_1^r$ be a sequence of sample pairs on which an Untied MixUp scheme specified by $(P^{\text{uMix}}, \gamma)$ and a DAT scheme with policy P^{DAT} will apply independently. If $(\nu, \nu')_1^r$ is symmetric and $\left(P^{\text{uMix}}, \gamma\right) = \mathcal{U}(P^{\text{DAT}})$, then $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma; \theta\right) = \mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right)$.*

Lemma 6 is proven in appendix A.7.

We now define another map $\mathcal{D}_{\text{u}}$ that maps an Untied MixUp scheme to a DAT scheme. Specifically $\mathcal{D}_u$ is a map from $\mathbb{P} \times \mathbb{F}$ to $\mathbb{P}$ such that for any $(p, g) \in \mathbb{P} \times \mathbb{F}$, $\mathcal{D}_{\text{u}}(p, g)$ is a configuration $p' \in \mathbb{P}$, where

$$p'(\lambda) := \left(g(\lambda)p(\lambda) + \overline{g(\overline{\lambda})}p(1-\lambda)\right)$$

It is easy to verify that $\int_0^1 p'(\lambda)d\lambda = 1$. Thus p' is indeed a distribution in $\mathbb{P}$ and $\mathcal{D}_{\text{u}}$ is well defined.

Lemma 7 *Let $(\nu, \nu')_1^r$ be a sequence of sample pairs on which an Untied MixUp scheme specified by $(P^{\text{uMix}}, \gamma)$ and a DAT scheme with policy P^{DAT} will apply independently. If $(\nu, \nu')_1^r$ is symmetric and $P^{\text{DAT}} = \mathcal{D}_{\text{u}}\left(P^{\text{uMix}}, \gamma\right)$, then $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma\right) = \mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right)$.*

Lemma 7 is proven in appendix A.8.

Lemmas 4, 5, 6 and 7 provide the building blocks for theorem 1, which we state subsequently. As r increases, both DAT and UMixUp converge in probability toward their respective expected loss (lemmas 4 and 5). Since as r increases, the sequence $(\nu, \nu')_1^r$ becomes arbitrarily close to the symmetric sampling distribution Q, then by lemma 6 the family of all DAT schemes converges in probability to a subset of all UMixUp schemes[7]. Lemma 7 proves the converse, i.e. that as r increases the family of UMixUp schemes converges in probability to a subset of DAT schemes. **As r increases, the family of UMixUp schemes therefore converges in probability to the entire family of DAT schemes**.

Theorem 1 *Let $(\nu, \nu')_1^\infty$ be drawn i.i.d. from Q. On this sample-pair data, an Untied MixUp scheme specified by (P^{Mix}, γ) and a DAT scheme specified by P^{DAT} will apply. In the Untied MixUp scheme, let Λ_1^∞ be drawn i.i.d. from P^{Mix}; in the DAT scheme, let Υ_1^∞ be drawn i.i.d. from P^{DAT}. If $P^{\text{DAT}} = \mathcal{D}_{\text{u}}\left(P^{\text{Mix}}, \gamma\right)$ or $\left(P^{\text{Mix}}, \gamma\right) = \mathcal{U}(P^{\text{DAT}})$, then $\left|\mathcal{L}^{\text{Mix}}\left((V, V')_1^r, \Lambda_1^r, \gamma; \theta\right) - \mathcal{L}^{\text{DAT}}\left((V, V')_1^r, \Upsilon_1^r; \theta\right)\right| \xrightarrow{\text{p}} 0$, as $r \to \infty$.*

An immediate consequence is that MixUp, as a subset of UMixUp, converges to a subset of DAT schemes. MixUp thus benefits from the same adversarial intuition as DAT and UMixUp.

The equivalence between UMixUp and DAT also indicates that there are DAT schemes that do not correspond to a MixUp scheme. These DAT schemes correspond to Untied MixUp schemes beyond standard MixUp. The relationship between MixUp, DAT and Untied MixUp is shown in Figure 1.1.

[7] since Lemma 6 shows that for each DAT scheme characterized by $P^{\text{DAT}} \in \mathbb{P}$, there exists a UMixUp scheme characterized by $(P^{\text{uMix}}, \gamma) \in \mathbb{P} \times \mathbb{F}$ -- specifically $\mathbb{U}(P^{\text{uMix}})$ -- such that $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma; \theta\right) = \mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right)$; ie $\mathcal{L}_E^{\text{uMix}}\left((\nu, \nu')_1^r, \gamma; \theta\right) = \mathcal{L}_E^{\text{DAT}}\left((\nu, \nu')_1^r; \theta\right)$ as $r \to \infty$.

3.5 Experiments

3.5.1 Experiment Setup

We consider an image classification task on the Cifar10, Cifar100, MNIST and Fashion-MNIST datasets. The baseline classifier is PreActResNet18 (see [25]), noting the same choice was made by the authors of Mixup[49]. The goal is to compare UMixUp, MixUp and the baseline model in terms of the models' ability to generalize. To evaluate a scheme's ability to generalize, each model is trained with 50,000 samples drawn at random from a set of 60,000 training samples, and we compare the misclassification rate when tested over the remaining 10,000 samples. The experiments are performed using two target-linear loss functions: cross-entropy and negative cosine loss. Results are reported for both loss functions.

The goal of each experiment is to search the scheme's mixing policy space to find the policy which achieves the best out-of-sample classification performance. The MixUp space is the set $\mathbb{P}$ of all probability distributions over $[0, 1]$ in which the mixing policy P^{Mix} resides. Like the authors of Mixup[49], we restrict of our search of $\mathbb{P}$ to the family of Beta distributions $B(\alpha, \alpha)$ – that is, the set of Beta distributions whose PDF $f(\lambda)$ is symmetric around the line $\lambda = 0.5$. It is sufficient to consider only symmetric distributions since, by symmetry of the sample pair distribution Q, all distributions are effectively symmetrical anyway[8].

The UMixUp space is the set $\mathbb{P} \times \mathbb{F}$ in which the mixing policy P^{uMix} and the weighting function γ reside. We restrict our search of $\mathbb{P} \times \mathbb{F}$ to $\mathcal{U}(B(\alpha, \beta))$, where $\alpha > 1$ and $\beta \leq 1$. In other words, we search DAT mixing policies belonging to the family $B(\alpha > 1, \beta \leq 1)$, to which we apply the mapping function $\mathcal{U}$ described in 3.33. This choice of UMixUp schemes accords to the γ function the desired properties described in section 3.2.3; this is proven in appendix A.5. In other words, $\gamma(0) = 0$ and $\gamma(1) = 1$, $\gamma(\lambda)$ is monotonically increasing, and $\gamma(1 - \lambda) = 1 - \gamma(\lambda)$.

The baseline model does not require any search since no mixing is performed.

3.5.2 Classification Error Reporting

We define a trial as a single run in which a model is trained over the training data for 200 epochs, and evaluated over the testing data at the end of each epoch. For each trial, we report the classification error rate as the average test error rate over the final 10 epochs of the trial. Because the model weights are randomly initialized, and training samples are randomly shuffled at the start of each epoch, the result of a trial is random. In order to improve our confidence in the reported classification error, we perform 100 trials for each particular setting. We refer to 100 trials using the same setting as an experiment.

To ascertain a confidence interval for the classification error over an experiment, we first need to determine the distribution of the classification errors. In figure 3.6, we plot

[8] See appendix A.4 for a brief demonstration (not a full proof but sufficient for providing intuition).

a histogram of classification errors over a 100-trial experiment. The mean classification error in this experiment is 4.219% with a standard deviation of 0.138%. The PDF of the gaussian distribution $N(4.219, 0.138)$ is overlayed on this histogram. Given that the

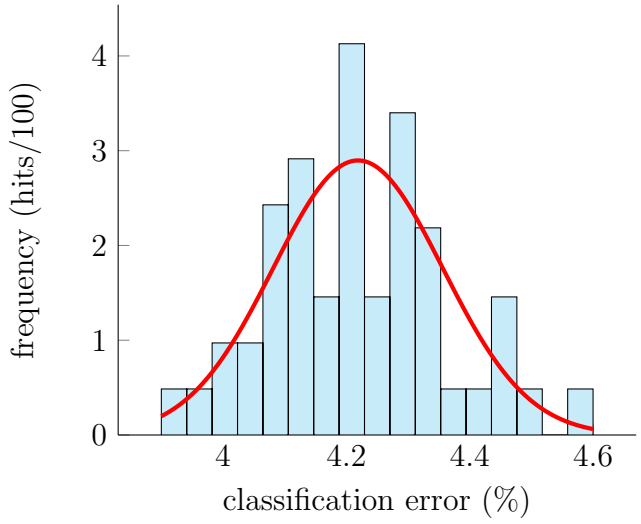


Figure 3.6: Histogram of test errors with $N(4.219, 0.138)$ overlay

Gaussian PDF is a roughly good fit, and given that a Gaussian distribution of results is a standard assumption in experimental studies, we conclude that classification error is likely to be normally distributed, and this is the assumption upon which we proceed. Thus for each experiment, we report the 95% confidence interval as $2 * \sigma$, where σ is the standard deviation of classification errors for the given experiment.

In summary, each trial is repeated 100 times, with the mean and 95%-confidence interval reported using the standard assumption that results are normally distributed.

3.5.3 Search Methodology

As described in section 3.5.1, the MixUp policy search is restricted to the family of Beta distributions $B(\alpha, \alpha)$, which is thus a one-parameter search. Due to hardware limitations, we restrict our search to $\alpha \in 0.1, 0.2, \ldots, 1.0$. For each α, we perform an experiment (100 trials) in which we train the model using the MixUP scheme with $\lambda \sim B(\alpha, \alpha)$. We report the experiment yielding the lowest mean test error rate, using the reporting method described in section 3.5.2.

As again described in section 3.5.1, the Untied Mixup policy search is restricted to $\mathcal{U}(B(\alpha, \beta))$, where $\alpha > 1$ and $\beta \leq 1$. This is thus a search over a subset of the space of pairs (α, β). Since the search space is much larger, we employ a more economical strategy given limited hardware resources. We begin by doing a grid search near $(\alpha = 2.0, \beta = 1.0)$. The pair $(\alpha = 2.0, \beta = 1.0)$ is chosen as a starting point because $\mathcal{U}(B(2.0, 1.0))$ is identical to MixUp with $\lambda \sim U(0, 1)$; in other words, the most conventional form of MixUp.

A typical grid consists of performing 12-trial experiments over all permutations of (α, β), where $\alpha \in 1.8, 1.9, 2.0, 2.1, 2.2$ and $\beta \in 0.6, 0.7, 0.8, 0.9, 1.0$. Past results may call for a different choice of starting grid, as a different starting grid may be more likely to contain the UMixUp scheme with the lowest test error rate. If the first grid yields a clear best result[9] within in the inner boundary (such as at $(1.9, 0.6)$), a second grid search is not required. If however the grid search yields a clear best result on the grid boundary (such as at $(2.1, 0.6)$, a new grid search is performed centered around the boundary best result[10].

If the best results are within the confidence interval of other settings in the grid, all results within the confidence interval of the best result are deemed the 'best' result, and a new grid search is performed if any of these belong to the grid boundary. This method is repeated until all 'best' results are inner points of the latest grid search. Once all grid searches are completed, all results within the confidence interval of the best result are repeated using a 100-trial experiment. We report the 100-trial experiment yielding the lowest mean test error rate, using the report method described in section 3.5.2.

3.5.4 Implementation and Improvements over Original Code

We use the original others' [49] MixUp implementation and training environment[11] as the baseline upon which we build our DAT and UMixUp training environments. The baseline code deploys two highly efficient regularization schemes[12] in addition to MixUp. Our implementation[13] retains these regularization schemes in order to allow fair comparison between our results and state-of-the-art publications (which also use such baseline regularization).

Two target-linear loss functions are essayed: cross-entropy (CE) loss and the negative-cosine (NC) loss as defined in equations 3.5 and 3.7, respectively. We implement CE loss similarly to previous works, which use CE loss to implement the baseline model.

We implemented negative cosine loss similar to equation 3.7[14]. In our implementation of the NC loss model, an orthonormal matrix $\overline{\mathbf{A}}$ of size $m \times d$ is constructed using singular value decomposition, where dimension d is fixed during training. Denote row x by $\overline{\mathbf{A}}[x]$. The label mapping function is $\boldsymbol{\varphi}(z) = \left(\overline{\mathbf{A}}\left[\sum_{j=1}^{m} j \mathbb{1}\{z = z^j\}\right]\right)^T$; in other words, $\boldsymbol{\varphi}$ is a lookup table of d-length orthornormal vectors, using the label z as the lookup index. The feature map of the original PreActResNet18 is linearly transformed to a d-dimensional

[9] in other words, an experiment yielding a mean test error rate that is below the confidence interval of all other experiments in the grid search

[10] but skipping those settings already tested in the first grid search

[11] https://github.com/facebookresearch/mixup-cifar10

[12] weight decay and conventional image augmentation (rotations, flips, and cropping)

[13] https://github.com/mixupAsDirectionalAdversarial/mixup_as_dat

[14] Unfortunately, due to a bug in our code, we implemented NC loss as

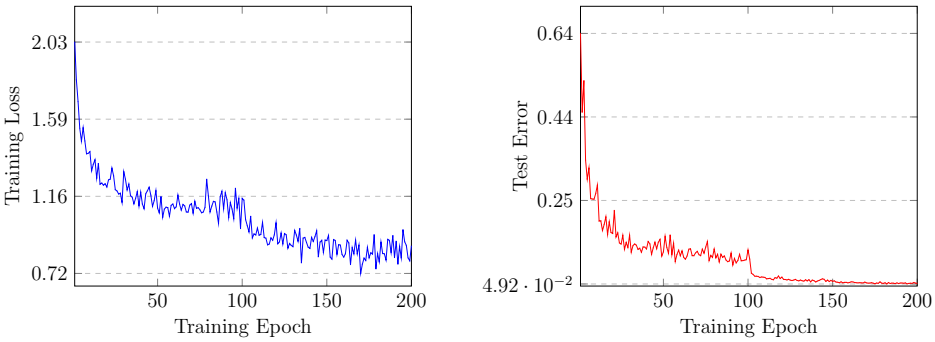
$$\ell_{\mathrm{NC}}\big((\mathbf{x}, \mathbf{y}); \theta\big) := -\frac{\mathbf{v}^{\mathrm{T}}\mathbf{y}}{||\mathbf{v}||\,||\mathbf{y}||} \tag{3.34}$$

Notice the extra term $||\mathbf{y}||$ in the denominator.

vector. The dimension d is chosen as 300 for Cifar10, MNIST and Fashion-Mnist (which have one black-and-white channel) and 700 for Cifar100 (which has 3 colored channels).

Two notable changes were made to the baseline code to improve performance. The original authors' implementation samples only one λ per mini-batch, giving rise to unnecessarily higher stochasticity of the mini-batched gradient $\widetilde{\nabla}_\theta \mathcal{L}(\theta)$. Our implementation samples λ independently for each sample.

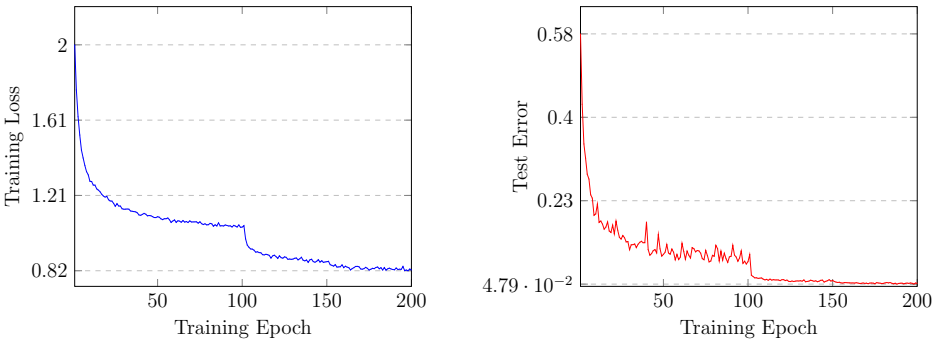
Figure 3.7 shows the training loss and test losses over the course of the training process, using the authors' original code. Spikes are caused by large swings in the gradient $\widetilde{\nabla}_\theta \mathcal{L}(\theta)$, which in turn causes the loss function to swing after each update of the parameters θ as per equation 2.122. This stochasticity in turn slows convergence of gradient descent algorithm, which results in a higher test error rate.



(a) Training loss curve (b) Test loss curve

Figure 3.7: Training and test losses using original λ-sampling

Figure 3.8 shows the training loss and test losses over the course of the training process, using our λ sampling method. The training and test loss curves are observably smoother, and we observe a lower test loss.



(a) Training loss curve (b) Test loss curve

Figure 3.8: Training and test losses using corrected λ-sampling

Additionally, the original code combines inputs by mixing a mini-batch of samples with a shuffled version of itself. This approach introduces a dependency between sampled pairs

and again increases the stochasticity of training. Our implementation creates two shuffled copies of the entire training dataset prior to each epoch, pairs them up, and then splits them into mini-batches. This gives a closer approximation to i.i.d. sampling and makes training smoother. While these implementation improvements have merit on their own, they do not provide a theoretical leap in understanding, and so we do not quantify their exact impact in our results analysis.

All models examined are trained using mini-batched backpropagation, for 200 epochs.

3.5.5 Simulation Management

Due to the compute intensive nature of the grid searches described in section 3.5.3, many thousands of simulations had to be run in order to obtain the results presented in section 3.5.6. Manually launching, tracking, and retrieving results for such a sizable volume of simulations would have been an impossible task for one person. As a result, we implemented[15] a simulation management toolkit compatible with SLURM, a widely used cluster scheduling software in the scientific community[16]. Since our toolkit is not the focus of this thesis, we only provide a summary of its features here:

1. *Parallel job launching.* The toolkit can handle launching hundreds of jobs in parallel in seconds.

2. *Regression relaunch.* The toolkit handles relaunching of incomplete or interrupted regressions. Failed simulations can be restarted and interrupted simulations picked off from where they left off.

3. *Sandboxed simulations.* All simulations are run in a separate autogenerated directories

   (a) Sandboxed simulations do not interfere with each other (eg scripts may write to a file with the same name in their output directory).

   (b) Snapshotting source code: The user's source code directory is copied to the autogenerated output, and it is this copied version which is executed. This flow ensures that users can continue editing their source code without affecting pending and running jobs.

   (c) Reproducible simulations: Snapshotting further ensures that simulations are fully reproducible, since all source code is snapshotted at the time of the regression launch. The regression command, slurm commands and simulation output are all logged, allowing the user to retrieve any arguments and parameters used in a given simulation.

4. *Regression monitoring.* The toolkit handles automatic reporting of the status of a regression (running, completed, failed) with a breakdown of each job.

[15] https://github.com/gobbedy/slurm_simulation_toolkit

[16] Notably, SLURM is used to schedule workloads on clusters operated by Compute Canada, the organization providing free HPC cluster resources to Canadian academics.

5. *Results processing.* The user can add callback functions to allow automatic processing of their regression results.

6. *Argument cascading.* Arguments following '--' are cascaded down to the user's base script, ensuring that the user does not need to modify the toolkit itself to pass down arguments to programs they execute.

7. *Automatic generation of regression cancellation script.* An autogenerated script kills the appropriate SLURM jobs if and when the user decides to cancel their regression. This both saves the user's time in tracking down running jobs to cancel, as well as helps maximize the use of computer resources for other users.

8. *Option to enforce a maximum of number of jobs in parallel for the current user.* This is useful for SLURM systems that don't use a fairshare system (eg. in Beta testing phase of a new cluster.)

9. *Option to run multiple simulations per GPU.* This helps maximize use of compute resources when GPU memory exceeds the model's needs (eg we found that ResNet with 128 batch size uses fewer GPU hours when running 2 processes per GPU on a 32GB GPU).

10. *Configurability.* Users can override default job parameters by supplying their own default job parameters.

3.5.6 Results: UMixUp vs MixUp vs Baseline

The results of our key experiments are given in tables 3.1 to 3.4. The 'model' column gives the training scheme used and loss function used. The 'policy' column gives the mixing policy used in the case of MixUp and UMixUp. The 'MEAN' column gives average test classification error rate over 100 trials. The 'ConfInt' column gives the 95%-confidence interval for the classification error rate. The 'Z-score' column gives the standard score of a one-tailed Z-test used to test the statistical significance of improvement in mean test error between the current row and the previous one. The 'p-value' gives the probability of the null hypothesis under the Z-test.

model	policy	runs	MEAN	ConfInt	Z-score	p-value
baseline-CE	–	100	5.476%	0.027%	–	–
mixUp-CE	B(0.9, 0.9)	100	4.199%	0.023%	48.181	<0.001
uMixUp-CE	$\mathcal{U}$(B(2.2, 0.9))	100	4.177%	0.025%	1.271	0.102
baseline-NC	–	100	5.605%	0.030%	–	–
mixUp-NC	B(1.0, 1.0)	100	4.508%	0.022%	59.086	<0.001
uMixUp-NC	$\mathcal{U}$(B(1.8, 1.0))	100	**4.455%**	0.025%	3.138	**0.001**

Table 3.1: Test error rate on CIFAR10.

model	policy	runs	MEAN	ConfInt	Z-score	p-value
baseline-CE	–	100	24.848%	0.060%	–	–
mixUp-CE	$\mathrm{B}(0.9, 0.9)$	100	22.020%	0.050%	72.185	<0.001
uMixUp-CE	$\mathcal{U}(\mathrm{B}(1.4, 0.7))$	100	**21.884%**	0.051%	3.809	**<0.001**
baseline-NC	–	100	25.270%	0.050%	–	–
mixUp-NC	$\mathrm{B}(0.9, 0.9)$	100	24.298%	0.051%	27.216	<0.001
uMixUp-NC	$\mathcal{U}(\mathrm{B}(1.3, 0.9))$	100	**23.819%**	0.054%	12.832	**<0.001**

Table 3.2: Test error rate on CIFAR100.

model	policy	runs	MEAN	ConfInt	Z-score	p-value
baseline-CE	–	100	0.816%	0.007%	–	–
mixUp-CE	$\mathrm{B}(1.0, 1.0)$	100	0.632%	0.005%	44.484	<0.001
uMixUp-CE	$\mathcal{U}(\mathrm{B}(1.7, 1.0))$	100	**0.609%**	0.005%	6.713	**<0.001**
baseline-NC	–	100	0.720%	0.007%	–	–
mixUp-NC	$\mathrm{B}(1.0, 1.0)$	100	0.607%	0.004%	28.115	<0.001
uMixUp-NC	$\mathcal{U}(\mathrm{B}(1.3, 0.9))$	100	**0.592%**	0.005%	4.600	**<0.001**

Table 3.3: Test error rate on MNIST.

model	policy	runs	MEAN	ConfInt	Z-score	p-value
baseline-CE	–	100	5.060%	0.027%	–	–
mixUp-CE	$\mathrm{B}(1.0, 1.0)$	100	4.585%	0.013%	31.678	<0.001
uMixUp-CE	$\mathcal{U}(\mathrm{B}(1.7, 0.8))$	100	4.570%	0.013%	1.605	0.054
baseline-NC	–	100	5.083%	0.016%	–	–
mixUp-NC	$\mathrm{B}(1.0, 1.0)$	100	4.767%	0.013%	30.091	<0.001
uMixUp-NC	$\mathcal{U}(\mathrm{B}(1.3, 0.9))$	100	**4.613%**	0.011%	17.726	**<0.001**

Table 3.4: Test error rate on Fashion-MNIST.

From these results, we see that the Untied MixUp schemes each outperform their MixUp counter-parts. In 6 of the 8 cases (those printed in **bold font**), the confidence interval of Untied MixUp is completely disjoint from that of the corresponding MixUp scheme; for all such cases, the one-tailed Z-score shows that the difference in the mean test accuracy is highly signification (with a p-value ≤ 0.001). In the two other cases, UMixUp outperforms MixUp, the confidence intervals of the MixUp schemes overlaps that of the UMixUp scheme.

To contextualize our results, we highlight the difficulty of achieving such results:

1. The baseline model (PreActResNet18) has been designed with highly focused inductive bias for image classification tasks, which leaves less room for effective regularization.

2. As noted in section 3.5.4, the original authors' baseline code already deploys two highly efficient regularization schemes, which we also used in all our experiments. Data augmentors[17] in particular provide effective regularization. We performed an

[17] flips, rotations and crops

experiment in which we ran the baseline code with and without data augmentors over 400 epochs. On average, the classification error without augmentation was 10.38%, while the classification error with augmentation was 4.16%. Thus, Untied MixUp is used in combination with already effective regularization schemes.

3. We made further improvements to the baseline implementation (discussed in section 3.5.4), which we applied to the MixUp scheme. Thus, we compare UMixUp against a more effective scheme than the one published by the original authors.

Untied MixUp is therefore applied to a model which is highly optimized to image classification, in combination with effective regularization schemes, and is compared against an improved method. All these factors contribute to leaving only a small room for additional regularization[18].

We further note that these UMixUp improvements are shown to be statistically significant. In other words, UMixUp's slight improvement over MixUp comes with high confidence, and competes with highly optimized solutions to the image classification task. In light of this context, we consider the improvement of Untied MixUp over MixUp to be significant.

The results also provide empirical evidence that MixUp and Untied MixUp both improve performance of the NC loss models. This validates our generalization of MixUp (and Untied MixUp) to models built with target linear losses.

3.5.7 Degree of Adversarial Movement

In this section we aim to understand the degree to which adversarial movement is involved in the UMixUp schemes reported in section 3.5.6. We plot and discuss two schemes which exemplify the more extreme scenarios: the $\mathcal{U}(B(2.2, 0.9))$ scheme is conservative in its adversarial movement while the $\mathcal{U}(B(1.7, 1.0))$ is more aggressively adversarial. We provide the plots of the remaining schemes in appendix A.9.

As a basis for comparison, we first discuss the UMixUp policy $\mathcal{U}(B(2.0, 1.0))$. Figure 3.9 plots the PDF of the P^{DAT} policy $B(2.0, 1.0)$ (blue line) as well as the PDF of the UMixUp policy P^{Mix} (green line) and weighting function $\gamma(\lambda)$ (red line), where $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$.

In this special case $\gamma(\lambda) = \lambda$ and P^{uMix} is the uniform distribution $U(0, 1)$. Thus $\mathcal{U}(B(2.0, 1.0))$ is identical to the most standard form of MixUp, wherein $\lambda \sim U(0, 1)$. A few key observations:

1. A DAT scheme whose PDF is $f(x) = 2x$[19] achieves similar regularization to standard MixUp (identical when $r \to \text{inf}$). Thus standard MixUp may be approximately achieved without label mixing, using $P^{\text{DAT}} = B(2.0, 1.0)$.

[18] Or stated alternatively, UMixUp is applied to a problem in which overfitting has already been largely eliminated, making even minor improvements significant

[19] The PDF of $B(2.0, 1.0)$ is $f(x) = 2x$.

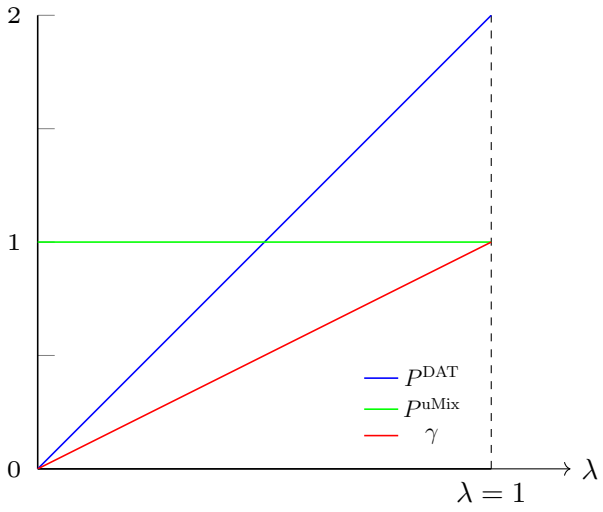


Figure 3.9: Curves of $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$ where $P^{\text{DAT}} = B(2.0, 1.0)$

2. The expressivity of the $B(\alpha, \beta)$ distribution is sufficient to allow our experiments to 'discover' standard Mixup via the UMixUp scheme $\mathcal{U}(B(2.0, 1.0))$, re-affirming this choice of P^{DAT} family.

3. The $B(2.0, 1.0)$ DAT scheme is much more likely to pick higher values of λ than the standard MixUp scheme. This makes sense intuitively, since a high value of λ (close to 1) represents a small movement away from the original input sample $\mathbf{x}$. Since the sample's original label is retained in DAT, we expect successful regularization schemes to generate nearby samples; applying the original label to distant samples would cause the classifier to underfit the data.

In the CIFAR10 experiments shown in table 3.1, the UMixUp cross-entropy loss experiment yielding the lowest error rate is the mixing scheme $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(B(2.2, 0.9))$. Figure 3.10 plots the P^{DAT} policy $B(2.2, 0.9)$ as well as the UMixUp equivalents $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$.

The $P^{\text{DAT}} = B(2.2, 0.9)$ distribution has a higher probability mass at higher values of λ than that of standard MixUp shown in figure 3.9. Since this is the best mixing policy after a thorough search, this suggests that in this particular classification task, standard MixUp 'moves' the samples too much, underfitting the data. In other words, introducing *less* adversarial movement than standard MixUp yields a superior regularization scheme.

This translates to a P^{uMix} density function which 'smiles'; in other words, which has higher density near $\lambda = 0$ and $\lambda = 1$. Given that $\gamma(0) = 0$ and $\gamma(1) = 1$, the generated samples are more likely to be close to one or the other of the training pairs used for mixing[20]. This is ultimately the same observation as the DAT-equivalent observation: 'moving' the samples less than standard MixUp will yield a superior regularization scheme.

[20]To see why this is true: let the mixing samples be $(\mathbf{x}, \mathbf{y})$ and $(\mathbf{x}', \mathbf{y}')$ and let the generated sample be

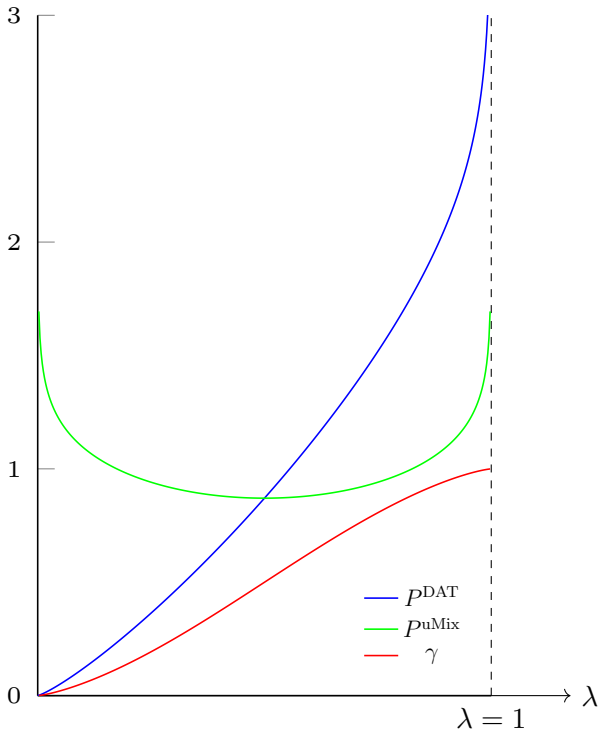


Figure 3.10: Curves of $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$ where $P^{\text{DAT}} = B(2.2, 0.9)$

In the MNIST experiments shown in table 3.3, the UMixUp cross-entropy-loss experiment yielding the lowest error rate is the mixing scheme $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(B(1.7, 1.0))$. Figure 3.11 plots the P^{DAT} policy $B(1.7, 1.0)$ as well as the uMixUp equivalents $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$.

The $P^{\text{DAT}} = B(1.7, 1.0)$ distribution, in contrast with the $B(2.2, 0.9)$ distribution discussed earlier, has a higher probability mass at lower values of λ than that of standard MixUp shown in figure 3.9. This suggests that in this particular classification task, standard MixUp 'moves' the samples too little, overfitting the data. In other words, introducing *more* adversarial movement than standard MixUp yields a superior regularization scheme.

This translates to a P^{uMix} density function which 'frowns'; in other words, which has higher density near $\lambda = 0.5$ and lower density near $\lambda = 0$ and $\lambda = 1$. This is again the same observation as the DAT-equivalent observation: 'moving' the samples more than standard MixUp in this particular task will yield a superior regularization scheme.

Appendix A.9 contains the $\mathcal{U}(P^{\text{DAT}})$ curves for all the UMixUp schemes reported in

$(\mathbf{x}_g, \mathbf{y}_g)$. Then we have

$$\mathbf{x}_g = \lambda \mathbf{x} + (1 - \lambda)\mathbf{x}'$$
$$\mathbf{y}_g = \gamma(\lambda)\mathbf{y} + \gamma(1 - \lambda)\mathbf{y}'$$

Given that $\gamma(0) = 0$ and $\gamma(1) = 1$, when $\lambda = 0$ then $(\mathbf{x}_g, \mathbf{y}_g) = (\mathbf{x}', \mathbf{y}')$ and when $\lambda = 1$ then $(\mathbf{x}_g, \mathbf{y}_g) = (\mathbf{x}, \mathbf{y})$. Thus when λ is near 0 or 1, the generated samples are close to one or the other of the training samples used for mixing.

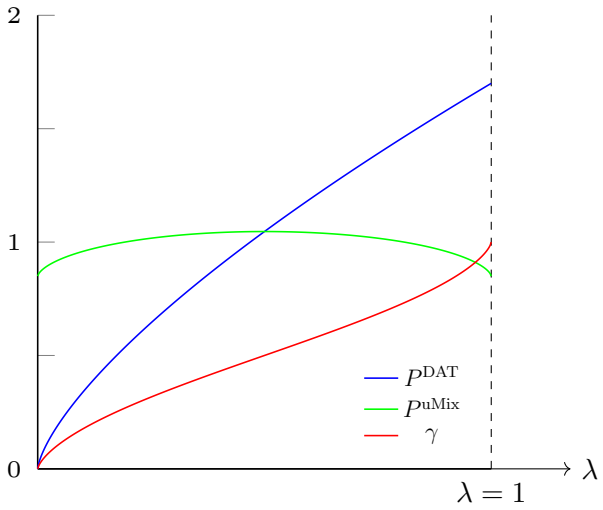


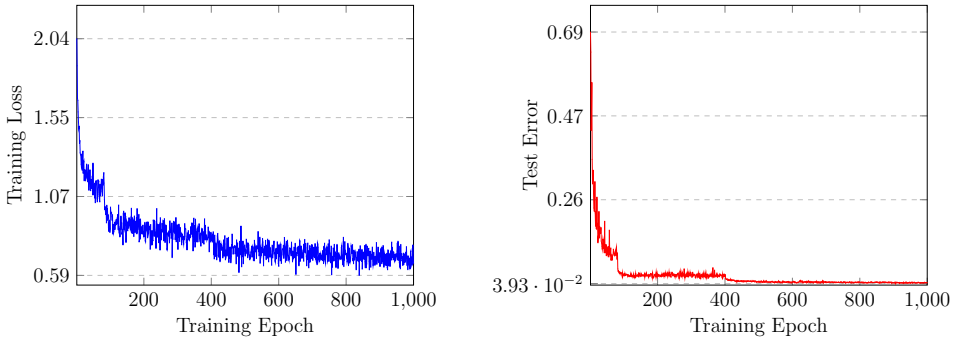
Figure 3.11: Curves of $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$ where $P^{\text{DAT}} = B(1.7, 1.0)$

tables 3.2 to 3.4.

3.5.8 Comparison of DAT and MixUp

We performed an experiment to compare the performance of DAT to MixUp. We compared the DAT policy $B(2.0, 1.0)$ with the UMixUp policy $\mathcal{U}(B(2.0, 1.0))$. As explained in section 3.5.7, $\mathcal{U}(B(2.0, 1.0))$ is nothing other than 'standard' MixUp, ie MixUp with $\lambda \sim U(0, 1)$. Figure 3.12. Thus the chosen DAT policy has the same expected loss as standard MixUp. We trained each model for 1000 epochs.

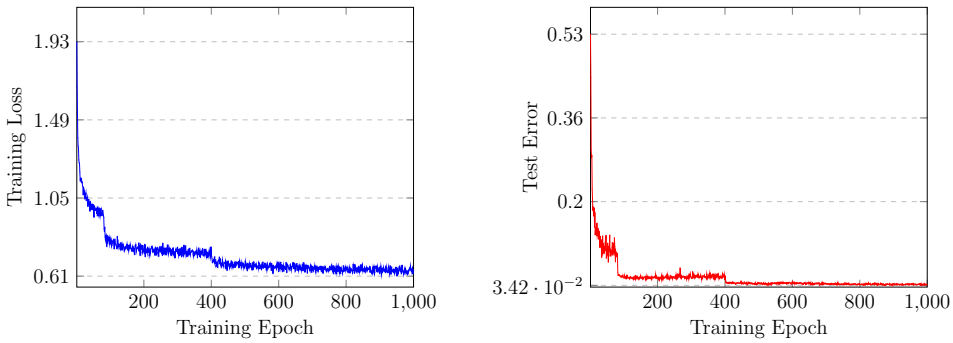
Figures 3.12 and 3.13 show the training loss and test losses over the course of training the DAT-regularized and MixUp-regularized models, respectively.



(a) Training loss curve (b) Test loss curve

Figure 3.12: Training and test losses using DAT

We see that the training and test losses using the DAT scheme are more stochastic, which results in a higher test error. This can be explained by the fact that DAT's label



(a) Training loss curve (b) Test loss curve

Figure 3.13: Training and test losses using MixUp

is more stochastic. As an example, suppose the generated label for some generated $\mathbf{x}_g$ in MixUp is

$$\mathbf{y}_g = \begin{bmatrix} 0 \\ 0.6 \\ 0 \\ 0 \\ 0.4 \end{bmatrix} \tag{3.35}$$

Then for the same input $\mathbf{x}_g$, the generated DAT label can be one of

$$\text{a)} \quad \mathbf{y}_g = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad \text{or b)} \quad \mathbf{y}_g = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

with a 60% chance of occurrence of a) and 40% of b). Thus, the gradient is more stochastic, which in turn causes a slower convergence and ultimately a high test error.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

We successfully developed a superset of MixUp, called Untied MixUp (UMixUp), which improves upon the highly optimized performance of MixUp in image classification tasks. We showed that the best member of UMixUp, as achieved by a systematic grid search of the UMixUp family, outperforms the best member of MixUp, as achieved by a similar grid search of the MixUp family. We thus showed that UMixUp provides a superior capacity to generalize classifier models to that of MixUp.

We developed an improved implementation of MixUp, whose training loss is more smooth and whose performance is improved. We also developed a simulation management toolkit which allowed us to run the heavy volume of simulations we required to obtain statistically significant results.

We developed a regularization scheme called Directional Adversarial Training, which provides useful insight into the working mechanism UMixUp, and by extension MixUp. We proved that as the number of mixed samples r tends to infinity, UMixUp is equivalent to DAT:

1. We provided mathematical proof that for any target-linear loss function, the expected loss of any member of UMixUp is equal to the expected loss of a member of DAT. We proved the converse, that any member of DAT is equal to the the expected loss of a member of UMixUp.

2. We proved that as the number of mixed samples r tends to infinity, the expected losses of UMixUp and DAT converge in probability to their respective overall losses.

3. Combining 1 and 2, we proved that as r tends to infinity, the respective families of UMixUp and DAT schemes are equivalent. In other words we proved that as $r \to \infty$, UMixUp is equivalent to DAT.

We explained a connection between DAT and adversarial training. By extension, we showed that adversarial training is a working mechanism UMixUp, and therefore of MixUp as well[1].

Putting this all together, we discovered a generalized MixUp algorithm, which carries the same explanatory power of adversarial training as MixUp.

Finally, we developed a new target-linear loss function. We showed empirically that MixUp and UMixUp carry similar improvements over baseline training using this new loss function. Thus, we provided validation that any target-linear loss function should allow MixUp to retain its beneficial adversarial properties.

4.2 Future Work

We showed a connection between DAT's interclass pair mixes and adversarial training, and between intraclass pair mixes and baseline training. We used this to justify our claim that MixUp employs adversarial training (via interclass training). While we believe the merit of this connection theoretically, our claims could be strengthened experimentally. We hypothesize that MixUp performed with only interclass training alone would be more adversarially robust, and show worse classification performance than DAT. This would be consistent with pure adversarial training, which is adversarially robust, and tends to show inferior generalization than baseline training [43, 48]. We further hypothesize that MixUp performed with only intraclass training would show similar performance to baseline training. Together, such experiments would improve our claims, which connect MixUp to adversarial training theoretically.

Furthemore, despite the development in this work, we believe that the current designs of MixUp and Untied MixUp are far from optimal. In particular, we believe a better design should allow an individualized weighting function for each training pair. In other words, we believe the $\gamma(\lambda)$ function should be replaced with a $\gamma(\mathbf{x}, \mathbf{x}', \lambda)$ function, which allows for a custom label for each generated sample. How this new γ function can be implemented remains open at this time.

Finally, the weakness of neural networks is its requirement of large datasets to generalize well. Pure adversarial training appears to improve generalization the best with small datasets [43]. Since MixUp employs adversarial training, we might expect MixUp to work especially well on small datasets, which in turn could make neural networks become relevant in applications with smaller datasets. Our improved implementation of MixUp may also help expand the effectiveness of MixUp to smaller datasets.

[1] since MixUp is a special case of UMixUp

APPENDICES

Appendix A

Proofs

A.1 Proof of Lemma 1

We prove that lemma 1 in the general case that $\mathcal{X}$ is a sequence (may contain duplicates).

Since ℓ_i (defined in equation 2.55) is the same as the number of elements in $\mathcal{X}$ which are identical to x_i, it is also the total number times a label is assigned to $X = x_i$, and is a constant given x_i. We can thus express $p(y^j|x_i)$ as

$$p(y^j|x_i) = \frac{\sum_{k=1}^{n} \mathbb{1}\{x_k = x_i\}\mathbb{1}\{y_k = y^j\}}{\ell_i} \tag{A.1}$$

where the numerator is the number of times label y^j is assigned to x_i.

$$H(p_{x_i}, q_{x_i}) = -\sum_{j=1}^{m} p(y^j|x_i) \log p(y^j|x_i, \theta) \tag{A.2}$$

$$= -\sum_{j=1}^{m} \left[\sum_{k=1}^{n} \mathbb{1}\{x_k = x_i\} \mathbb{1}\{y_k = y^j\} \frac{1}{\ell_i} \right] \log p(y^j|x_i, \theta) \tag{A.3}$$

$$= -\frac{1}{\ell_i} \sum_{j=1}^{m} \left[\sum_{k=1}^{n} \mathbb{1}\{x_k = x_i\} \mathbb{1}\{y^j = y_k\} \log p(y^j|x_i, \theta) \right] \tag{A.4}$$

$$= -\frac{1}{\ell_i} \sum_{k=1}^{n} \mathbb{1}\{x_k = x_i\} \log p(y_k|x_i, \theta) \tag{A.5}$$

We define $\mathbb{D}(\mathcal{X})$ as the set of sets of indices corresponding to identical items in $\mathcal{X}$, eg. if

$$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\} = \{47, 29, 71, 29, 17, 71, 71, 13\}$$

then

$$\mathbb{D}(\mathcal{X}) = \{\{1\}, \{2, 4\}, \{3, 6, 7\}, \{5\}, \{8\}\}$$

Formally, define the set of identical indices of x as

$$d(x) = \{j \in [1..n] : x_j = x\} \tag{A.6}$$

and the set of sets of identical indices

$$\mathbb{D}(\mathcal{X} = x_1^n) = \{d(x_j) : j \in [1..n]\} \tag{A.7}$$

Note that since $\mathbb{D}(\mathcal{X})$ is a set, duplicate versions of $d(x_j)$ are ignored.

We further note that if $\mathbb{G} \in \mathbb{D}(\mathcal{X})$ and $g \in \mathbb{G}$, then $|\mathbb{G}| = \ell_g$; i.e. $|\mathbb{G}|$ is the number of terms x in the sequence $\mathcal{X}$ that are identical to x_g.

Plugging equation A.5 into the left-hand side of lemma 1 yields

$$-\sum_{i=1}^{n} H(p_{x_i}, q_{x_i}) = -\sum_{i=1}^{n}\left[-\frac{1}{\ell_i}\sum_{k=1}^{n}\mathbb{1}\{x_k = x_i\}\log p(y_k|x_i,\theta)\right] \tag{A.8}$$

$$= \sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\left(\sum_{g\in\mathbb{G}}\left[\frac{1}{\ell_g}\sum_{k=1}^{n}\mathbb{1}\{x_k = x_g\}\log p(y_k|x_g,\theta)\right]\right) \tag{A.9}$$

$$= \sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\left(\sum_{g\in\mathbb{G}}\left[\frac{1}{|\mathbb{G}|}\sum_{k\in\mathbb{G}}\log p(y_k|x_g,\theta)\right]\right) \tag{A.10}$$

$$= \sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\left(|\mathbb{G}|\cdot\frac{1}{|\mathbb{G}|}\left[\sum_{k\in\mathbb{G}}\log p(y_k|x_k,\theta)\right]\right) \tag{A.11}$$

$$= \sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\left(\sum_{k\in\mathbb{G}}\log p(y_k|x_g,\theta)\right) \tag{A.12}$$

$$= \sum_{i=1}^{n}\log p(y_i|x_i,\theta) \tag{A.13}$$

A.13 is the right-hand side of lemma 1, and thus the lemma's proof is complete.

Note that x_g is constant over all $g \in \mathbb{G}$ so that the square-bracketed term in equation A.10 is constant. This is used to go from equation A.10 to equation A.11. □.

A.2 General MAP and ML solutions when output samples are soft labels

In the general case that $\mathcal{X}$ is a sequence, let w_i be the total number of times a label is assigned to $X = x_i$. For the k^{th} term in sequence $\mathcal{S}$, we imagine an experiment where r_k labels are generated using our hypothesis $p(y^j|x_k,\theta) = q_{x_k}(y^j)$. Then w_i can be given as

$$w_i = \sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_i\} \tag{A.14}$$

We can thus express $p(y^j|x_i)$ as

$$p(y^j|x_i) = \frac{\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_i\}\mathbf{y}_k[j]}{w_i} \tag{A.15}$$

where the numerator is the number of times label y^j is assigned to x_i in dataset $\mathcal{S}$.

$$H(p_{x_i}, q_{x_i}) = -\sum_{j=1}^{m} p(y^j|x_i) \log p(y^j|x_i, \theta) \tag{A.16}$$

$$= -\sum_{j=1}^{m} \left[\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_i\} \mathbf{y}_k[j] \frac{1}{w_i} \right] \log p(y^j|x_i, \theta) \tag{A.17}$$

$$= -\frac{1}{w_i} \sum_{j=1}^{m} \left[\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_i\} \mathbf{y}_k[j] \log p(y^j|x_i, \theta) \right] \tag{A.18}$$

The log likelihood of $\mathbf{y}_k$ is given by equation 2.67 (with the same reasoning):

$$\log p(\mathbf{y}_k|x_i, \theta) = \log \prod_{j=1}^{m} q_{x_i}(y^j)^{r_k \cdot \mathbf{y}_k[j]} \tag{A.19}$$

$$= \sum_{j=1}^{m} r_k \cdot \mathbf{y}_k[j] \log q_{x_i}(y^j) \tag{A.20}$$

We define $\mathbb{D}(\mathcal{X})$ as per equation A.7. Let $\mathbb{G} \in \mathbb{D}(\mathcal{X})$. For all $g \in \mathbb{G}$, w_g is constant, so we denote it $w_{\mathbb{G}}$ when highlighting that it does not depend on any specific g in $\mathbb{G}$. Summing A.18 over all $g \in \mathbb{G}$ and negating:

$$-\sum_{g \in \mathbb{G}} H(p_{x_g}, q_{x_g}) = \sum_{g \in \mathbb{G}} \left(\frac{1}{w_g} \sum_{j=1}^{m} \left[\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_g\} \mathbf{y}_k[j] \log p(y^j|x_g, \theta) \right] \right) \tag{A.21}$$

$$-w_{\mathbb{G}} \sum_{g \in \mathbb{G}} H(p_{x_g}, q_{x_g}) = \sum_{g \in \mathbb{G}} \left(\sum_{j=1}^{m} \left[\sum_{k \in \mathbb{G}} r_k \cdot \mathbf{y}_k[j] \log p(y^j|x_g, \theta) \right] \right) \tag{A.22}$$

$$-\sum_{g \in \mathbb{G}} w_g \cdot H(p_{x_g}, q_{x_g}) = \sum_{g \in \mathbb{G}} \left(\sum_{k \in \mathbb{G}} \left[\sum_{j=1}^{m} r_k \cdot \mathbf{y}_k[j] \log p(y^j|x_g, \theta) \right] \right) \tag{A.23}$$

$$= \sum_{g \in \mathbb{G}} \left(\sum_{k \in \mathcal{G}} \log p(\mathbf{y}_k|x_g, \theta) \right) \tag{A.24}$$

$$= |\mathbb{G}| \cdot \sum_{k \in \mathbb{G}} \log p(\mathbf{y}_k|x_k, \theta) \tag{A.25}$$

$$-\sum_{g \in \mathbb{G}} m_g \cdot H(p_{x_g}, q_{x_g}) = \sum_{k \in \mathbb{G}} \log p(\mathbf{y}_k|x_k, \theta) \tag{A.26}$$

where

$$m_g = \frac{\sum_{k=1}^{n} r_k \cdot \mathbb{1}\{x_k = x_g\}}{\sum_{k=1}^{n} \mathbb{1}\{x_k = x_g\}} \tag{A.27}$$

Summing both sides of equation A.26 over all $\mathbb{G} \in \mathbb{D}(\mathcal{X})$,

$$-\sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\sum_{g\in\mathbb{G}} m_g \cdot H(p_{x_g}, q_{x_g}) = \sum_{\mathbb{G}\in\mathbb{D}(\mathcal{X})}\sum_{k\in\mathbb{G}} \log p(\mathbf{y}_k|x_k,\theta) \tag{A.28}$$

$$\sum_{i=1}^{n} m_i \cdot H(p_{x_i}, q_{x_i}) = \sum_{i=1}^{n} \log p(\mathbf{y}_i|x_i,\theta) \tag{A.29}$$

We may then write equation 2.26 as

$$\theta^{MAP} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i}^{n} m_i \cdot H(p_{x_i}, q_{x_i}) - \log p(\theta) \right) \tag{A.30}$$

Equation 2.36 can be written as

$$\theta^{ML} = \underset{\theta}{\operatorname{argmin}} \left(\sum_{i}^{n} m_i \cdot H(p_{x_i}, q_{x_i}) \right) \tag{A.31}$$

If the samples $\mathbf{y}_k[j]$ are equally reliable then $r = r_k$ is constant, and $m = m_i$ reduces to the constant r. Thus if output samples $\mathbf{y}_k[j]$ are equi-reliable, equations A.30 and A.31 reduce to equations 2.73 and 2.74 respectively.

A.3 Proof of CDF symmetry

It can be easily shown that functions that are symmetric around the vertical line $y = 0.5$ satisfy the property $f(x) = f(1-x)$. We use this to prove that a function whose PDF is symmetric around $y = 0.5$ has a CDF satisfying the property $f(1-x) = 1 - f(x)$.

Suppose $\lambda \in [0, 1]$ is distributed such that its PDF is symmetric around $\lambda = 0.5$. Denote its PDF $f(\lambda)$. Then the CDF at $\lambda = \lambda_0$ is

$$F(\lambda_0) = \int_0^{\lambda_0} f(\lambda) d\lambda \tag{A.32}$$

$$= 1 - \int_{\lambda_0}^{1} f(\lambda) d\lambda \tag{A.33}$$

$$= 1 - \int_{0}^{1-\lambda_0} f(\lambda) d\lambda \tag{A.34}$$

$$= 1 - F(1-\lambda_0) \tag{A.35}$$

where the symmetry of $f(\lambda)$ is used to go from equation A.33 to equation A.34 (this may be done by using the geometric symmetry of the areas under the curve, or using variable substitution $u = 1 - \lambda$). Figure illustrates a PDF with the aforementioned symmetric property, where the area of the blue region is $F(\lambda_0)$ and the area of the green region is $1 - F(\lambda_0) = F(1 - \lambda_0)$

Thus a PDF function possessing the property $f(x) = f(1-x)$ has a CDF which possesses the property $F(1-x) = 1 - F(x)$.

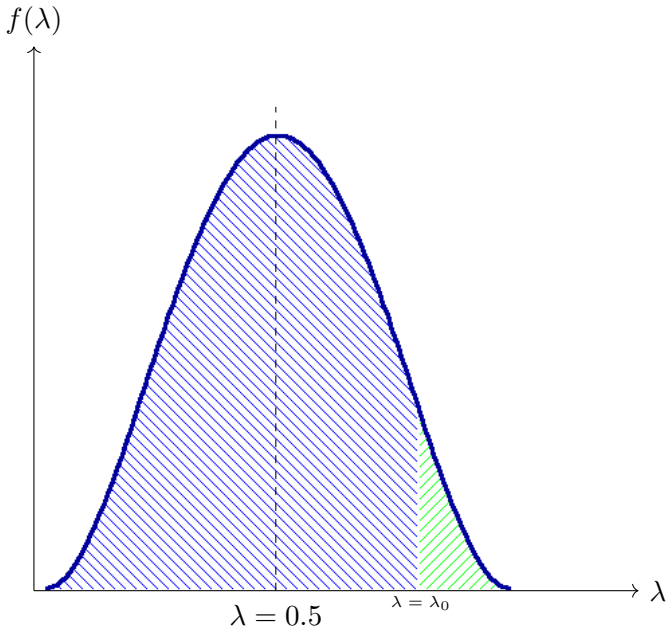


Figure A.1: Example of PDF function where $f(\lambda) = f(1-\lambda)$

A.4 Sufficiency of Symmetric Distributions

Suppose as before that $s = (\mathbf{x}, \mathbf{y})$ and $s' = (\mathbf{x}', \mathbf{y}')$ are two samples in $\mathbb{S}_D$, and Q is a symmetric distribution over $\mathbb{S}_D \times \mathbb{S}_D$. Then sample pair (s, s') is equally likely to be drawn as (s', s). Let the MixUp sample generated using (s, s') be $(\mathbf{x}_a, \mathbf{y}_a)$ and the sample generated using (s', s) be $(\mathbf{x}_b, \mathbf{y}_b)$, so that

$$\mathbf{x}_a = \lambda_a \mathbf{x} + (1-\lambda_a)\mathbf{x}' \tag{A.36}$$
$$\mathbf{y}_a = \lambda_a \mathbf{y} + (1-\lambda_a)\mathbf{y}' \tag{A.37}$$
$$\mathbf{x}_b = \lambda_b \mathbf{x}' + (1-\lambda_b)\mathbf{x} \tag{A.38}$$
$$\mathbf{y}_b = \lambda_b \mathbf{y}' + (1-\lambda_b)\mathbf{y} \tag{A.39}$$

where λ_a is the mixing ratio drawn when mixing (s, s') and λ_b is the mixing ratio drawn when mixing (s', s). Now the following generated pair

$$\mathbf{x}_g = \lambda \mathbf{x} + (1-\lambda)\mathbf{x}' \tag{A.40}$$
$$\mathbf{y}_g = \lambda \mathbf{y} + (1-\lambda)\mathbf{y}' \tag{A.41}$$

can occur in one of two ways: the pair (s, s') is drawn and $\lambda_a = \lambda$, or (s', s) is drawn and $\lambda_b = 1-\lambda$. Since (s, s') and (s', s) have equal probability of occurring, then the probability of $(\mathbf{x}_g, \mathbf{y}_g)$ is $0.5(p(\lambda) + p(1-\lambda))$, which is symmetric around $\lambda = 0.5$.

A.5 Desirable properties of UMixUp using $\mathcal{U}(B(\alpha, \beta))$ with $\alpha > 1$, $\beta \leq 1$

Formally, the PDF of the $B(\alpha, \beta)$ distribution is

$$f(x; \alpha, \beta) := \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\mathcal{B}(\alpha, \beta)} \tag{A.42}$$

where

$$\mathcal{B}(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)} \tag{A.43}$$

where Γ is the Gamma function. Thus using the transform $\mathcal{U}(B(\alpha, \beta))$ gives the mixing policy $p(\lambda)$ and the weighting function $\gamma(\lambda)$ as

$$p(\lambda) := \frac{1}{2}\left(f(\lambda; \alpha, \beta) + f(1-\lambda; \alpha, \beta)\right) \text{ and } \gamma(\lambda) := \frac{f(\lambda; \alpha, \beta)}{f(\lambda; \alpha, \beta) + f(1-\lambda; \alpha, \beta)} \tag{A.44}$$

We prove that given $\alpha > 1$ and $\beta \leq 1$, $\gamma(\lambda)$ possesses the desirable properties described in section 3.2.3.

1. $\gamma(0) = 0$ and $\gamma(1) = 1$: $\gamma(0) = 0$ is an obvious consequence of $\alpha > 1$. $\gamma(1)$, however, is undefined for $\beta < 1$, but $lim_{\lambda \to 1}\gamma(\lambda) = 1$. Thus for practical purposes the property is achieved, since the probability of drawing $\lambda = 1$ exactly is zero.

2. $\gamma(\lambda)$ is a monotonically increasing function. Let $\lambda_1, \lambda_2 \in [0, 1]$ such that $\lambda_1 > \lambda_2$. To prove that $\gamma(\lambda)$ is a monotonically increasing function over $[0, 1]$, we show that $\gamma(\lambda_1) \geq \gamma(\lambda_2)$. Since $\gamma(\lambda) > 0$, we may equivalently prove that $\frac{1}{\gamma(\lambda_1)} \leq \frac{1}{\gamma(\lambda_2)}$.

$$\frac{1}{\gamma(\lambda_1)} = \frac{f(\lambda_1; \alpha, \beta) + f(1-\lambda_1; \alpha, \beta)}{f(\lambda_1; \alpha, \beta)} \tag{A.45}$$

$$= 1 + \frac{f(1-\lambda_1; \alpha, \beta)}{f(\lambda_1; \alpha, \beta)} \tag{A.46}$$

$$= 1 + \frac{(1-\lambda_1)^{\alpha-1}(\lambda_1)^{\beta-1}}{\lambda_1^{\alpha-1}(1-\lambda_1)^{\beta-1}} \tag{A.47}$$

$$= 1 + \left(\frac{1-\lambda_1}{\lambda_1}\right)^{\alpha-1}\left(\frac{\lambda_1}{1-\lambda_1}\right)^{\beta-1} \tag{A.48}$$

   and similarly

$$\frac{1}{\gamma(\lambda_2)} = 1 + \left(\frac{1-\lambda_2}{\lambda_2}\right)^{\alpha-1}\left(\frac{\lambda_2}{1-\lambda_2}\right)^{\beta-1} \tag{A.49}$$

   We now inspect the terms

$$\left(\frac{1-\lambda_1}{\lambda_1}\right)^{\alpha-1} \qquad \text{and} \qquad \left(\frac{1-\lambda_2}{\lambda_2}\right)^{\alpha-1} \tag{A.50}$$

Since the exponent $\alpha - 1 > 0$, and since the numerator $1 - \lambda_1 < 1 - \lambda_2$ and the denominator $\lambda_1 > \lambda_2$, then

$$\left(\frac{1-\lambda_1}{\lambda_1}\right)^{\alpha-1} < \left(\frac{1-\lambda_2}{\lambda_2}\right)^{\alpha-1} \tag{A.51}$$

Similarly since $\beta - 1 \leq 0$,

$$\left(\frac{\lambda_1}{1-\lambda_1}\right)^{\beta-1} <= \left(\frac{\lambda_2}{1-\lambda_2}\right)^{\beta-1} \tag{A.52}$$

Given that all fractions are positive we have

$$\left(\frac{1-\lambda_1}{\lambda_1}\right)^{\alpha-1}\left(\frac{\lambda_1}{1-\lambda_1}\right)^{\beta-1} < \left(\frac{1-\lambda_2}{\lambda_2}\right)^{\alpha-1}\left(\frac{\lambda_2}{1-\lambda_2}\right)^{\beta-1} \tag{A.53}$$

proving that $\frac{1}{\gamma(\lambda_1)} \leq \frac{1}{\gamma(\lambda_2)}$ and thus that $\gamma(\lambda)$ is a monotonically increasing function.

3. $\gamma(1-\lambda) = 1 - \gamma(\lambda)$. Proof:

$$1 - \gamma(\lambda) = \frac{f(\lambda;\alpha,\beta) + f(1-\lambda;\alpha,\beta)}{f(\lambda;\alpha,\beta) + f(1-\lambda;\alpha,\beta)} - \frac{f(\lambda;\alpha,\beta)}{f(\lambda;\alpha,\beta) + f(1-\lambda;\alpha,\beta)} \tag{A.54}$$

$$= \frac{f(1-\lambda;\alpha,\beta)}{f(1-\lambda;\alpha,\beta) + f(\lambda;\alpha,\beta)} \tag{A.55}$$

$$= \gamma(1-\lambda) \tag{A.56}$$

A.6 Proof of Lemma 4

For any fixed infinite sequence $(\nu,\nu')_1^\infty$ of samples drawn i.i.d. from Q and any infinite sequence of i.i.d. random variables Λ_1^∞ drawn from P^{DAT}, let $\mathcal{L}^{\text{DAT}}\left((\nu,\nu')_1^r, \Lambda_1^r\right)$ be defined according to (3.25), with the first r elements of $(\nu,\nu')_1^\infty$ and the first r elements of Λ_1^∞ as input. Define

$$\delta_{\text{DAT}} := \max_{\substack{(\nu,\nu')\in \\ \mathcal{D}\times\mathcal{D}}} \sup_{\substack{(\lambda,\lambda')\in \\ [0,1]\times[0,1]}} |\ell^{\text{DAT}}(\nu,\nu',\lambda;\theta) - \ell^{\text{DAT}}(\nu,\nu',\lambda';\theta)|$$

For any given $\lambda_1^r \in [0,1]^r$ and any of its modified version $u_1^r \in [0,1]^r$ which differs from λ_1^r in exactly one location, it can be verified, following the definition of δ_{DAT}, that

$$\left|\mathcal{L}^{\text{DAT}}\left((\nu,\nu')_1^r, \lambda_1^r;\theta\right) - \mathcal{L}^{\text{DAT}}\left((\nu,\nu')_1^r, u_1^r;\theta\right)\right| \leq \delta_{\text{DAT}}/r$$

Since $\Lambda_1, \Lambda_2, \ldots \Lambda_K$ are independent and by McDiarmid Inequality [28], it follows that for any $\epsilon > 0$,

$$\Pr\left[\mathcal{L}^{\text{DAT}}\left((\nu,\nu')_1^r, \Lambda_1^r;\theta\right) - \mathcal{L}_E^{\text{DAT}}\left((\nu,\nu';\theta)_1^r\right) \geq \epsilon\right] < 2\exp\left(-\frac{2\epsilon^2}{r\cdot(\delta_{\text{DAT}}/r)^2}\right)$$

which proves the lemma □

A.7 Proof of Lemma 6

$$
\begin{aligned}
\mathcal{L}_E^{\text{uMix}}\left((\nu,\nu')_1^r,\gamma;\theta\right) :=& \frac{1}{r}\sum_{k=1}^{r}\mathbb{E}_{\lambda\sim P^{\text{Mix}}}\left\{\gamma(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\overline{\gamma(\lambda)}\ell^{\text{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)\right\}\\
=&\frac{1}{r}\sum_{k=1}^{r}\int\left(\gamma(\lambda)P^{\text{Mix}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\ +\overline{\gamma(\lambda)}P^{\text{Mix}}(\lambda)\ell^{\text{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)\right)d\lambda\\
=&\frac{1}{r}\sum_{k=1}^{r}\int\left(\frac{1}{2}P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\ +\frac{1}{2}P^{\text{DAT}}(\overline{\lambda})\ell^{\text{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)\right)d\lambda\\
=&\frac{1}{2K}\left(\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)d\lambda+\sum_{k=1}^{r}\int P^{\text{DAT}}(\overline{\lambda})\ell^{\text{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)d\lambda\right)\\
\overset{(a)}{=}&\frac{1}{2K}\left(\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)d\lambda+\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu'_k,\nu_k,\lambda;\theta)d\lambda\right)\\
\overset{(b)}{=}&\frac{1}{2K}\left(\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)d\lambda+\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)d\lambda\right)\\
=&\frac{1}{r}\sum_{k=1}^{r}\int P^{\text{DAT}}(\lambda)\ell^{\text{DAT}}(\nu_k,\nu'_k,\lambda;\theta)d\lambda\\
=&\mathcal{L}_E^{\text{DAT}}\left((\nu,\nu')_1^r;\theta\right)
\end{aligned}
$$

where (a) is due to a change of variable in the integration, (b) is due to the symmetry of $(\nu,\nu')_1^r$. Note that in equation 3.33 $g(\lambda)$ is undefined at values of λ for which the denominator is zero. But the lemma holds true because the denominator is only zero when $p(\lambda)=0$, so those λ for which $g(\lambda)$ is undefined never get drawn in the DAT scheme. □

A.8 Proof of Lemma 7

$$
\begin{aligned}
\mathcal{L}_E^{\mathrm{uMix}}\left((\nu,\nu')_1^r,\gamma;\theta\right) &= \frac{1}{r}\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\left(\gamma(\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\overline{\gamma(\lambda)}\ell^{\mathrm{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)\right)\\
&= \frac{1}{r}\left(\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\gamma(\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\overline{\gamma(\lambda)}\ell^{\mathrm{DAT}}(\nu'_k,\nu_k,\overline{\lambda};\theta)\right)\\
&\overset{(a)}{=} \frac{1}{r}\left(\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\gamma(\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\overline{\gamma(\lambda)}\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\overline{\lambda};\theta)\right)\\
&\overset{(b)}{=} \frac{1}{r}\left(\mathbb{E}_{\lambda\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\gamma(\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\mathbb{E}_{\overline{\lambda}\sim P^{\mathrm{Mix}}}\sum_{k=1}^{r}\overline{\gamma(\overline{\lambda})}\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\right)\\
&= \frac{1}{r}\sum_{k=1}^{r}\int\left(\gamma(\lambda)P^{\mathrm{Mix}}(\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)+\overline{\gamma(\overline{\lambda})}P^{\mathrm{Mix}}(1-\lambda)\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\right)d\lambda\\
&= \frac{1}{r}\sum_{k=1}^{r}\int\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\underbrace{\left(\gamma(\lambda)P^{\mathrm{Mix}}(\lambda)+\overline{\gamma(\overline{\lambda})}P^{\mathrm{Mix}}(1-\lambda)\right)}_{\mathcal{D}_{\mathrm{u}}(P^{\mathrm{Mix}},\gamma)}d\lambda\\
&= \frac{1}{r}\sum_{k=1}^{r}\mathbb{E}_{\lambda\sim P^{\mathrm{DAT}}}\ell^{\mathrm{DAT}}(\nu_k,\nu'_k,\lambda;\theta)\\
&= \mathcal{L}_E^{\mathrm{DAT}}\left((\nu,\nu')_1^r;\theta\right).
\end{aligned}
$$

where (a) is due to the symmetry of $(\nu,\nu')_1^r$, and (b) is by a change of variable in the second term (renaming $1-\lambda$ as λ). □

A.9 Curves of Reported UMixUp Results

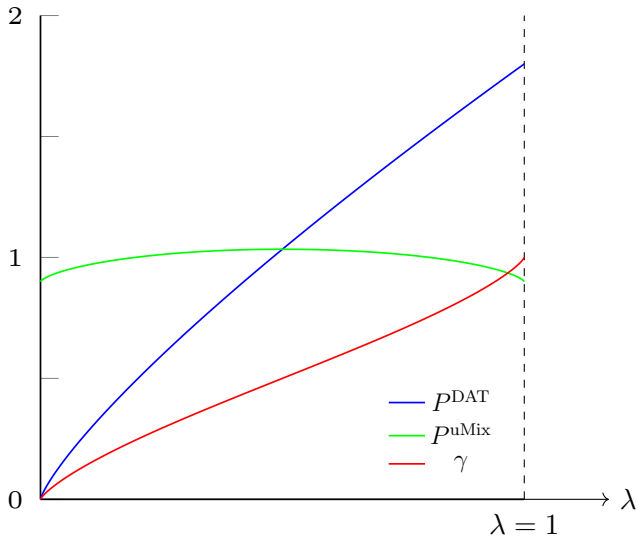


Figure A.2: Curves of $\{P^{\mathrm{uMix}}, \gamma\} = \mathcal{U}(P^{\mathrm{DAT}})$ where $P^{\mathrm{DAT}} = B(1.8, 1.0)$

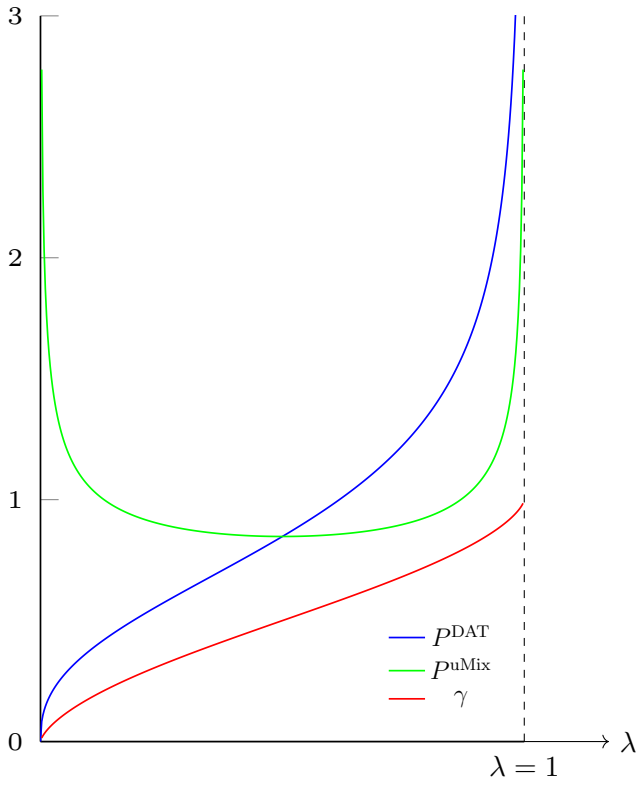


Figure A.3: Curves of $\{P^{\mathrm{uMix}}, \gamma\} = \mathcal{U}(P^{\mathrm{DAT}})$ where $P^{\mathrm{DAT}} = B(1.4, 0.7)$

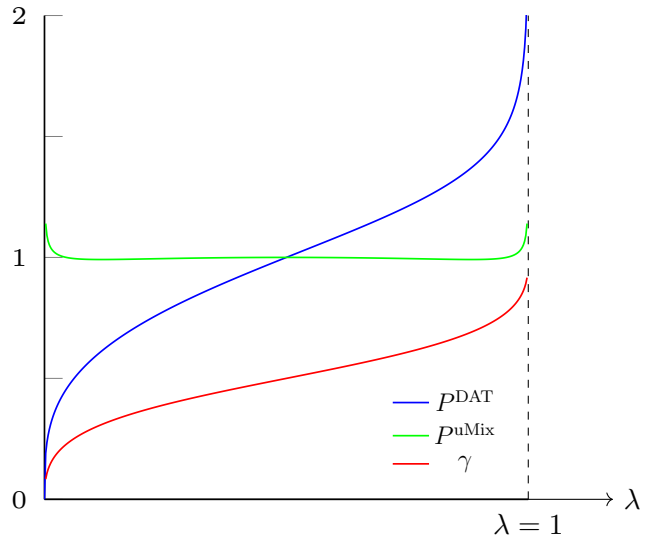


Figure A.4: Curves of $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$ where $P^{\text{DAT}} = B(1.3, 0.9)$

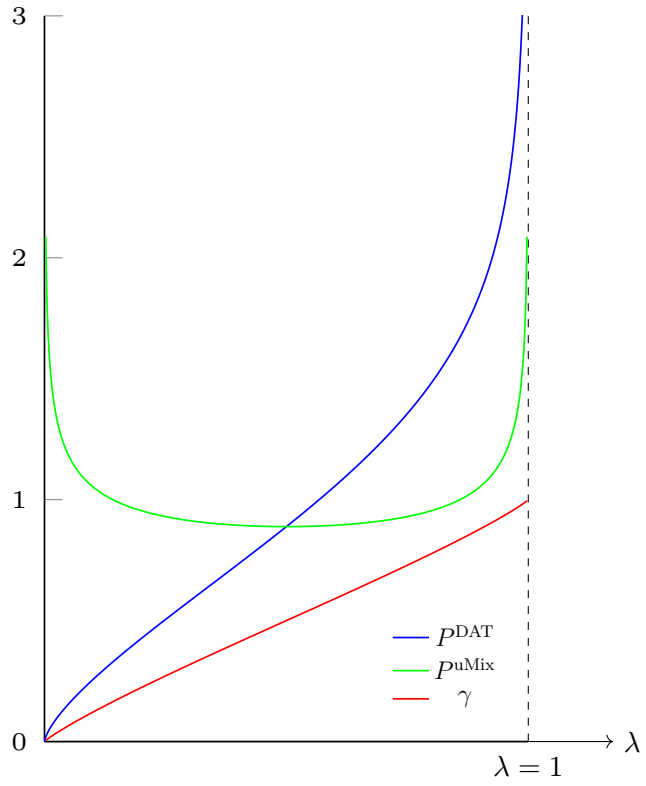


Figure A.5: Curves of $\{P^{\text{uMix}}, \gamma\} = \mathcal{U}(P^{\text{DAT}})$ where $P^{\text{DAT}} = B(1.7, 0.8)$

References

[1] Grant Sanderson (aka Youtuber 3Blue1Brown). Personal communication.

[2] N. S. Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992. doi: 10.1080/00031305.1992.10475879. URL https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879.

[3] Devansh Arpit, Stanislaw K. Jastrzebski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S. Kanwal, Tegan Maharaj, Asja Fischer, Aaron C. Courville, Yoshua Bengio, and Simon Lacoste-Julien. A closer look at memorization in deep networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, pages 233–242, 2017. URL http://proceedings.mlr.press/v70/arpit17a.html.

[4] Anish Athalye, Nicholas Carlini, and David Wagner. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420*, 2018.

[5] Hagan Demuth Beale, Howard B Demuth, and MT Hagan. Neural network design. *Pws, Boston*, 1996.

[6] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, 2007. ISBN 0387310738. URL http://www.amazon.com/Pattern-Recognition-Learning-Information-Statistics/dp/0387310738%3FSubscriptionId%3D13CT5CVB80YFWJEPWS02%26tag%3Dws%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D0387310738.

[7] Léon Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2(4):303–314, December 1989. ISSN 0932-4194. doi: 10.1007/BF02551274. URL http://dx.doi.org/10.1007/BF02551274.

[9] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

[10] Jeffrey L. Elman. Finding structure in time. *COGNITIVE SCIENCE*, 14(2):179–211, 1990.

[11] J. Geweke. *Contemporary Bayesian Econometrics and Statistics*. Wiley Series in Probability and Statistics. Wiley, 2005. ISBN 9780471744726. URL https://books.google.ca/books?id=0cWkYlBb8DwC.

[12] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[13] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR 2015*, 2015.

[14] Hongyu Guo, Yongyi Mao, and Richong Zhang. Mixup as locally linear out-of-manifold regularization. *arXiv preprint arXiv:1809.02499*, 2018.

[15] Douglas Hawkins. The problem of overfitting. *Journal of chemical information and computer sciences*, 44:1–12, 05 2004. doi: 10.1021/ci0342472.

[16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[17] Warren He, Bo Li, and Dawn Song. Decision boundary analysis of adversarial examples. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018. URL https://openreview.net/forum?id=BkpiPMbA-.

[18] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[19] Wolfgang Härdle. *Applied Nonparametric Regression*. Econometric Society Monographs. Cambridge University Press, 1990. doi: 10.1017/CCOL0521382483.

[20] So S (https://stats.stackexchange.com/users/133063/so s). What exactly is a hypothesis space in machine learning? Cross Validated. URL https://stats.stackexchange.com/q/304702. URL:https://stats.stackexchange.com/q/304702 (version: 2019-02-05).

[21] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems 4, NIPS Conference, Denver, Colorado, USA, December 2-5, 1991*, pages 950–957, 1991. URL http://papers.nips.cc/paper/563-a-simple-weight-decay-can-improve-generalization.

[22] Jan Kukacka, Vladimir Golkov, and Daniel Cremers. Regularization for deep learning: A taxonomy. *CoRR*, abs/1710.10686, 2017.

[23] Alex Lamb, Vikas Verma, Juho Kannala, and Yoshua Bengio. Interpolated Adversarial Training: Achieving Robust Neural Networks without Sacrificing Too Much Accuracy. *arXiv e-prints*, art. arXiv:1906.06784, Jun 2019.

[24] Moshe Leshno, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867, 1993. URL http://dblp.uni-trier.de/db/journals/nn/nn6.html#LeshnoLPS93.

[25] Kuang Liu. URL https://github.com/kuangliu/pytorch-cifar, 2017.

[26] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6232–6240, Red Hook, NY, USA, 2017. Curran Associates Inc. ISBN 9781510860964.

[27] Yongyi Mao. Lecture notes in deep learning and reinforcement learning, October 2018.

[28] Colin McDiarmid. On the method of bounded differences. *Surveys in combinatorics*, 141(1):148–188, 1989.

[29] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997. ISBN 978-0-07-042807-2.

[30] Takeru Miyato, Andrew M Dai, and Ian Goodfellow. Adversarial training methods for semi-supervised text classification. *arXiv preprint arXiv:1605.07725*, 2016.

[31] In Jae Myung. Tutorial on maximum likelihood estimation. *Journal of Mathematical Psychology*, 47(1):90 – 100, 2003. ISSN 0022-2496. doi: https://doi.org/10.1016/S0022-2496(02)00028-7. URL http://www.sciencedirect.com/science/article/pii/S0022249602000287.

[32] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.

[33] Richard R Picard and R Dennis Cook. Cross-validation of regression models. *Journal of the American Statistical Association*, 79(387):575–583, 1984.

[34] Maithra Raghu, Ben Poole, Jon Kleinberg, Surya Ganguli, and Jascha Sohl-Dickstein. On the expressive power of deep neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 2847–2854, International Convention Centre, Sydney, Australia, 06–11 Aug 2017. PMLR. URL http://proceedings.mlr.press/v70/raghu17a.html.

[35] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[36] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3 edition, 2010.

[37] Uri Shaham, Yutaro Yamada, and Sahand Negahban. Understanding adversarial training: Increasing local stability of supervised models through robust optimization. *Neurocomputing*, 307:195–204, 2018.

[38] H.T. Siegelmann and E.D. Sontag. On the computational power of neural nets. *J. Comput. Syst. Sci.*, 50(1):132–150, February 1995. ISSN 0022-0000. doi: 10.1006/jcss.1995.1013. URL https://doi.org/10.1006/jcss.1995.1013.

[39] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014. URL http://dl.acm.org/citation.cfm?id=2670313.

[40] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

[41] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[42] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996. ISSN 00359246. URL http://www.jstor.org/stable/2346178.

[43] Dimitris Tsipras, Shibani Santurkar, Logan Engstrom, Alexander Turner, and Aleksander Madry. Robustness may be at odds with accuracy. 2018. URL http://arxiv.org/abs/1805.12152. cite arxiv:1805.12152.

[44] V. Vapnik. Principles of risk minimization for learning theory. In *Proceedings of the 4th International Conference on Neural Information Processing Systems*, NIPS'91, page 831–838, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc. ISBN 1558602224.

[45] Vikas Verma, Alex Lamb, Christopher Beckham, Amir Najafi, Aaron Courville, Ioannis Mitliagkas, and Yoshua Bengio. Manifold mixup: Learning better representations by interpolating hidden states. 2018.

[46] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*, 2016. URL http://www.bmva.org/bmvc/2016/papers/paper087/index.html.

[47] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017. URL https://openreview.net/forum?id=Sy8gdB9xx.

[48] Hongyang Zhang, Yaodong Yu, Jiantao Jiao, Eric P. Xing, Laurent El Ghaoui, and Michael I. Jordan. Theoretically principled trade-off between robustness and accuracy, 2019.

[49] Hongyi Zhang, Moustapha Cisse, Yann N Dauphin, and David Lopez-Paz. mixup: Beyond empirical risk minimization. *arXiv preprint arXiv:1710.09412*, 2017. URL https://github.com/facebookresearch/mixup-cifar10.