A Responsible Softmax Layer in Deep Learning

by

Ryan Coatney

A Dissertation Submitted to the Faculty of the

Department of Mathematics

In Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy

In the Graduate College

The University of Arizona

2 0 2 0

Table of Contents

List of Figures

LIST OF TABLES

Abstract

Clustering algorithms are an important part of modern data analysis. The K-means and EM clustering algorithm both use an iterative process to find latent (or hidden) variables in a mixture distribution. These Hidden variables may be interpreted as class label for the data points of a sample. In connection with these algorithms, we consider a family of nonlinear mappings called *responsibility* maps. The responsibility map is obtained as a gradient of the log likelihood of N independent samples, drawn from a mixture of K distributions. We look at the discrete dynamics of this family of maps and give a proof that iteration of responsibility converges to an estimate of the mixing coefficients. We also show that the convergence is consistent in the sense that the fixed point acts as a maximizer of the log likelihood.

We call the process of determining class weight by iteration *dynamic responsibility* and show that it converges to a unique set of weights under mild assumptions. Dynamic responsibility (DR) is inspired by the expectation step of the expectation maximization (EM) algorithm and has a useful association with Bayesian methods. Like EM, dynamic responsibility is an iterative algorithm, but DR will converge to a unique maximum under reasonable conditions. The weights determined by DR can also be found using gradient descent but DR guarantees non-negative weights and gradient descent does not.

We present a new algorithm which we call *responsible softmax* for doing classification with Neural networks. This algorithm is intended to handle imbalanced training sets and is accomplished via multiplication by per class weights. These weights may be interpreted as class probabilities for a generalized mixture model, and are determined through DR rather than by empirical observation of the training set and heuristically selecting the underlying probability distributions.

We compare the performance of responsible softmax with other standard techniques, including standard softmax, and weighted softmax using empirical class probabilities. We use generated Gaussian mixture model data and the MNIST data set for proof of concept. We show that in general, responsible softmax produces more useful classifiers than softmax when presented with imbalanced training data. It will also be seen that responsible softmax approximates the performance of empirically weighted softmax, and in some cases may do better.

Chapter 1

INTRODUCTION

1.1   Introduction

This dissertation introduces a new type of neural network layer for classification problems which I call responsible softmax. I will show that both in theory and practice, responsible softmax may be a useful tool for dealing with imbalanced data of many types, especially when the data can be modeled with a mixture model.

Working with imbalanced data is a common problem in machine learning. There are many reasons for this, though a common one is that some classes of data are difficult to obtain. It may also be that the underlying process creating the data is imbalanced. Regardless of the reason, imbalanced data tends to bias classifiers towards the majority classes. For this reason and others, it is important to address imbalanced data when choosing an algorithm.

There have been several techniques developed over the years to handle the problem of data imbalance. Each of these algorithms has several pros and cons. Trade offs between accuracy, precision, computation time, and generalization are not easy to balance. For example, Batista *et al.* [BPM04] use data balancing to level the per class instances either by undersampling the majority classes, or oversampling the minority classes. While this can fix the inherent bias against minority classes in typical classifiers, it also can hurt generalization by either severely reducing the data available for training or memorizing (overfitting) the minority class.

Another example is prior re-weighting (or scaling). This refers to the idea that the output of a neural network may be modified via Bayes' Rule to appropriately adjust the model of the class mixtures to match what is found empirically. While I cover this idea in more detail in section 5.2 there are many tricks and techniques that fall into this category. See Lawrence e.t. al. [LBB+12] for a partial review.

A final example I give for now is mixture of experts (MOE) [JJNH91]. The idea here is to train several learners so that they can differentiate between only a few classes. The idea here is that each learner could be an 'expert' in identifying one or two classes. Then each expert learner reports their

confidence on classification to a gating network. This gating network is trained to choose the correct expert for each data point. Nets that work as MOE are very adaptable, but they can be expensive to train. Further, many of the training methods for MOE can get stuck in suboptimal local minima as per Makkuva *et al.* [MVKO18].

Responsible Softmax (RS) addresses some of the problems of imbalance. RS resembles both MOE and prior scaling. It is similar to prior scaling in that it can be viewed as a re-weighting or regularization of standard softmax layer and cross-entropy loss. It resembles MOE in that it uses a type of gating function to establish weights for the loss. These weights can be trained separately or concurrently with the standard neural network weights.

Responsible Softmax derives inspiration from the notion of cluster responsibility from the soft K-means [Mac02, ch.20-22] and expectation maximization [DLR77, NH98] clustering algorithms. Much of the work in this dissertation assumes an underlying generalized mixture model for the data. Cluster responsibility is closely related to the mixing coefficients of such models.

In general, a mixture model combines K different probability distributions by a convex combination of those models. In more specific terms, if $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$ $k = 1, \ldots, K$ are different distribution functions, and $\{\pi_1, \ldots, \pi_K\}$ are positive reals such that $\sum_k \pi_k = 1$, then the distribution function of the mixture model is

$$\phi(\boldsymbol{x}|\boldsymbol{\pi}, \boldsymbol{\theta}_1, \ldots, \boldsymbol{\theta}_K) = \sum_{k=1}^{K} \pi_k f_k(\boldsymbol{x}, \boldsymbol{\theta}_k). \tag{1.1}$$

The parameters $\{\pi_1, \ldots, \pi_K\}$ are interchangeably called mixing coefficients and class probabilities. Responsible softmax directly estimates these class probabilities.

This dissertation defines and explores Responsible Softmax (RS) and dynamic responsibility (DR), including a proof of convergence for dynamic responsibility in theorem 3.8 and comparison of responsible softmax to some standard algorithms. I first show that dynamic responsibility requires simple hypotheses for convergence to a MLE for mixing coefficients of a mixture model. This applies also to responsible softmax which uses dynamic responsibility to weight a softmx layer of a neural network. Then the paper examines the performance of responsible softmax when compared to the standard softmax and a softmax weighted with an empirical prior derived from the data labels. I use data sets that highlight the advantages of each algorithm.

Chapter 2 covers some basic background required for the dissertation. Chapter 3 covers mathematical analysis of dynamic responsibility. I show that fixed points of dynamic responsibility act as maximum likelihood estimators for per class probabilities. Chapter 4 covers the basics needed for

using responsible weighting in back propagation and gives a basic outline of RS. Chapter 5 will cover empirical analysis of networks using RS on imbalanced data sets and compare RS to the performance of other methods, including standard softmax.

Chapter 2

Background

2.1 Clustering and Classification

Classification and clustering are two closely related statistical tasks. Clustering is more exploratory in nature, while classification is predictive. Clustering looks to find ways of associating data points with each other to maximize some objective. Classification seeks to put new data into predefined groups.

In some sense, Clustering may be considered an example of distribution inference given data. If we have several samples, and a good idea that each sample comes form a different distribution, we can ask separate, but related questions. First, what are the distributions that were sampled from? Second, how can we infer which data points came from which distribution? In general, the first question may be called clustering, and the second question classification. For now we will focus mostly on clustering and return to classification later.

2.1.1 Clustering

To make the question of clustering more precise, consider the problem of sampling from $K < \infty$ probability distributions given by distribution functions $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$, $1 \leq k \leq K$. Each distribution $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$ is chosen at random with proportion π_k^*, $\sum_k \pi_k^* = 1$. This is a situation that is mimicked easily enough in Monte Carlo simulations, and is common in applications.

Experiment 2.1. As a two stage experiment, we work as follows to select a point in $\boldsymbol{x}_n \in \mathbb{R}^I$:

1. Stage 1: From the K possible distributions, select a label k_n with probability $P(k_n = k) = \pi_k^*$.

2. Stage 2: Sample $\boldsymbol{x}_n$ from $f_{k_n}(\boldsymbol{x}, \boldsymbol{\theta}_k)$

Given N such data points, $\boldsymbol{D} = \{\boldsymbol{x}_n\}_{n=1}^N$, clustering then is the problem of estimating the parameters $\boldsymbol{\Theta} = \{\boldsymbol{\pi}^*, \boldsymbol{\theta}_1, \boldsymbol{\theta}_2, \ldots, \boldsymbol{\theta}_K\}$, where $\boldsymbol{\pi}^* := \{\pi_k^*\}_{k=1}^K$. This essentially assigns each of the data points $\boldsymbol{x}_n$ as a sample from a particular distribution in $X_k \sim f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$, $1 \leq k \leq K$. We have in this case

$$P(\boldsymbol{x}_n|\boldsymbol{\Theta}) = \sum_k P(\boldsymbol{x}_n|k_n = k, \boldsymbol{\theta}_k)P(k_n = k, \boldsymbol{\theta}_k) = \sum_k \pi_k^* f_k(\boldsymbol{x}_n, \boldsymbol{\theta}_k). \qquad (2.1)$$

Here we make the implication that $P(\boldsymbol{x}_n|k_n = k, \boldsymbol{\theta}_k) = f_k(\boldsymbol{x}_n, \boldsymbol{\theta}_k)$ and $P(k_n = k, \boldsymbol{\theta}_k) = \pi_k^*$.

We often choose to estimate the parameters $\{\pi_k^*\}$ first as often our other estimates for the remaining parameters $\boldsymbol{\Theta}$ are not independent of our choices for the labels. In example of this, consider some situation where an algorithm has found a local maximum likelihood estimate for the parameters $\hat{\boldsymbol{\Theta}} = \{\hat{\boldsymbol{\pi}}, \hat{\boldsymbol{\theta}}_1, \ldots, \hat{\boldsymbol{\theta}}_K\}$. Supposing further that all of the pdfs $f_k(x, \theta_k)$ are similar (*e.g.* gaussian) then we know that the given local estimate is not unique.

To be precise, if σ is any permutation of $1, \ldots, K$, then the estimate given by

$$\sigma(\hat{\boldsymbol{\Theta}}) := \{\hat{\pi}_{\sigma(1)}, \hat{\pi}_{\sigma(2)}, \ldots, \hat{\pi}_{\sigma(K)}, \hat{\boldsymbol{\theta}}_{\sigma(1)}, \hat{\boldsymbol{\theta}}_{\sigma(2)}, \ldots, \hat{\boldsymbol{\theta}}_{\sigma(K)}\}$$

gives the exact same likelihood as $\hat{\boldsymbol{\Theta}}$. This means that our likelihood function is not convex, and that we have no guarantees that any algorithm will give us a 'correct' estimate. Because of this it is a common practice to use several different initializations for any clustering algorithm used, and compare the results.

2.1.2 Classification

Classification does not generally share the non-convexity problem associated with clustering. Instead of trying to estimate parameters for the distribution of the data, clustering attempts to find the best label for a data point from a given set of prescribed labels. This is often presented as a maximum likelihood problem, in the sense that we are trying to maximize the probability of class labels given the data, *e.g.* find k such that $P(\boldsymbol{x}_n|k_n = k, \boldsymbol{\Theta})$ is maximized.

Classification is often given as a type of supervised learning as discussed in section 2.2. Often one is interested in the class k' of a new data point $\boldsymbol{x}'$ which an algorithm may infer from calculating $P(\boldsymbol{x}'|k' = k, \boldsymbol{D}) \ \forall k \le K$. A maximum likelihood estimate could then compare these probabilities and make a decision on the label. Such a process is also called maximum likelihood classification. Another option would be to use Bayes rule and calculate $P(k' = k|\boldsymbol{x}', \boldsymbol{D}) \ \forall k \le K$, and then choose the class with the greatest probability. This is called a maximum *a posteriori* (MAP) classifier.

A connection between clustering and classification appears through some analysis via Bayes rule,

$$P(k_n = k | \boldsymbol{x}_n, \boldsymbol{\Theta}) = \frac{P(\boldsymbol{x}_n | k_n = k, \boldsymbol{\Theta}) P(k_n = k, \boldsymbol{\theta}_k)}{P(\boldsymbol{x}_n | \boldsymbol{\Theta})} \tag{2.2}$$

in that the goal of clustering is proportional to the goal of classification. Indeed, it is possible to use clustering and establish a model for use in classification, as done with the software package *AutoClass* [SC96, CS96]. It is also possible to use labeled data sets and classification to perform nearest neighbors clustering as discussed in *The Elements of Statistical Learning* chapter 13 [HTF09].

2.2 Unsupervised and Supervised Machine Learning

Within the realm of machine learning there are two broad collections of algorithms known as supervised and unsupervised learning. While each set of algorithms has their own uses and drawbacks, they are often compared as if they were the two extremes of a spectrum. The practice of employing algorithms in these two categories is often more nuanced.

Supervised machine learning requires large amounts of labeled data $\boldsymbol{D} = \{\boldsymbol{X}, \boldsymbol{T}\}$. Here the data points $\boldsymbol{X} := \{\boldsymbol{x}^n\}$ are drawn from some underlying probability space $\mathcal{X}$. The data also has an extra feature $\boldsymbol{T} := \{\boldsymbol{t}^n\}$, drawn from a set $\mathcal{T}$, that acts as labels for individual data points. Many practical situations suppress direct discussion of the sets $\mathcal{X}, \mathcal{T}$ in favor of talking about what the data implies about the underlying sets.

The labels $\{\boldsymbol{t}^n\}$ may be categorical, as when we are trying to classify data points. $\boldsymbol{T}$ may also be output of some unknown function on which we wish to perform regression. For the remainder of this paper, we will consider the classification problem but regression remains a good source of inspiration.

In either case, the goal of supervised learning is to develop a program that correctly predicts the label t' of a given data point x' which the algorithm has not seen before. In the case of classification problems, the algorithm gives a set of probabilities $P(t' = \ell | x')$ as ℓ ranges over the finite set of classification categories which we will call $\mathcal{C}$. A reasonable constraint in this situation is to require that

$$\sum_{\ell \in \mathcal{C}} P(t' = \ell | x') = 1.$$

In light of the above discussion it is effective when considering supervised learning to view the problem as an estimation of the conditional probability $P(\boldsymbol{X} | \boldsymbol{T})$. We may then use Bayes' Rule to

find

$$P(\boldsymbol{T}|\boldsymbol{X}) \propto P(\boldsymbol{X}|\boldsymbol{T}) \cdot P(\boldsymbol{X}).$$

As part of this process, it is typical to choose a loss (or cost) function $L : \mathcal{X} \times \mathcal{T} \rightarrow \mathbb{R}$. The probability $P(\boldsymbol{X}|\boldsymbol{T})$ is then estimated by minimization of the chosen loss function. Use of the known labels $\boldsymbol{T}$ and a given loss function are key features of supervised learning. Common supervised learning algorithms are support vector machines, naive Bayes, logistic regression, and neural networks such as the multilayer perceptron.

The basic ideas behind supervised learning can be more fully explored through the example of the multilayer perceptron. This discussion follows the explanation given in Bishop [Bis06]. This model is discussed in chapter 5 of Bishop, and there it is also called the feed-forward neural network. It is closely related to, and simpler than, the 'deep' learning in common use today.

For this specific example, suppose that $\mathcal{X} = \{\boldsymbol{x}^{(n)}\}$, with $\boldsymbol{x}^{(n)} \in \mathbb{R}^d$ for $n = 1 \ldots N$. Recall that the goal of supervised classification is to make an appropriate approximation of the distribution $P(\mathcal{T}|\mathcal{X})$.

The way a multilayer perceptron does this is through composing two or more layers to perform inference. Each layer can be viewed as a many logistic regression algorithms working together to pass appropriate information on to the next layer. The final layer is called the loss layer, and it has the responsibility of measuring how far the outputs of the neural network are from the ideal distribution.

Unsupervised learning, on the other hand, seeks to find patterns in the data without the requirement of labels. One set of unsupervised learning algorithms are clustering algorithms. These algorithms seek to find patterns among the data and group the data points according to these patterns.

Among clustering algorithms, we wish to pay most attention to mixture modeling. While mixture modeling is useful for more than just clustering, it is worthwhile to think of them as a clustering algorithms to begin with. Two algorithms for mixture models on which we will focus are the K-means algorithm and the Expectation Maximization (EM) algorithm. While we will focus on each of these algorithms in detail in sections 2.5.1 and 2.5.2, at this point we will discuss some of the common details.

First, as in the introduction, all mixture models suppose that the data is sampled from $K < \infty$ different distributions modeled by the distributions $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$, $k = 1 \ldots K$. Here the $\boldsymbol{\theta}_k$ are distribution specific parameters. We then form a model $p(\boldsymbol{x})$ by taking a convex combination of the given

distributions,

$$P(\boldsymbol{x}; \boldsymbol{\pi}, \boldsymbol{\Theta}) = \sum_{k=1}^{K} \pi_k f_k(\boldsymbol{x}, \boldsymbol{\theta}_k). \tag{2.3}$$

Where we require that $\sum_k \pi_k = 1$ and $\boldsymbol{\Theta} = \{\boldsymbol{\theta}_1 \dots \boldsymbol{\theta}_K\}$. The goal then of mixture models is to determine $\{\boldsymbol{\pi}, \boldsymbol{\Theta}\}$ from the given data. In this case, equation 2.3 is the same as equation 1.1. One important example of a mixture model is the Gaussian Mixture Model (GMM), which assumes that the distributions $f_k(\boldsymbol{x}, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ ar all normal distributions with possibly unknown means and covariance matrices.

We recall from section 2.1 that clustering and classification are two closely related but different problems. Clustering seeks to infer a distribution for the various clusters in the data. Classification looks to label the data points according to membership in various clusters. Both the K-means and EM algorithms have a semi-classification step which we will refer to as responsibility assignment. [Bis06, DFO20]

In the EM algorithm, these responsibility assignments are often referred to as latent variables. The mixing constants $\boldsymbol{\pi}$, may also be considered latent variables, but as will be seen, responsibility is closely related to the mixing constants.

We first give a definition of responsibility. In its simplest form, responsibility is the cluster assignment for a point in one iteration of the K-means or EM algorithm. If $N = |\mathcal{D}|$ is the number of data points, and K is the number of clusters, then $r_k^{(n)}$, $1 \leq n \leq N$, $1 \leq k \leq K$ is the responsibility of the K-th cluster for the data point $\boldsymbol{x}^{(n)}$.

In the most basic implementation, $r_k^{(n)} \in \{0, 1\}$. Explicitly, we have $r_k^{(n)} = 1$ if $x^{(n)}$ is assigned to cluster k and $r_k^{(n)} = 0$ otherwise. We will call this *hard responsibility*. As a slight modification, we may also consider the case where $r_k^{(n)} \in [0, 1]$. In this case we require that $\sum_k r_k^{(n)} = 1$. We will call this *soft responsibility*.

The total responsibility for the cluster k is the value $N_k = \sum_n r_k^{(n)}$. Whether we are working with hard or soft responsibility, the relative responsibility of cluster k is $\frac{N_k}{N}$. The mixing probability is approximated by the relative responsibility as will be discussed further in section 3.5.

As a brief discussion of the connection between responsibility and mixing constants, recall experiment 2.1. As the number of samples generated by this Monte Carlo sampling grows, the following holds.

$$\lim_{N \to \infty} \frac{N_k}{N} = \pi_k.$$

2.3 Softmax and Logistic Regression

Logistic regression is a method of modeling a binary dependent variable. For example, we may wish to classify data into one of two categories, and then logistic regression may be a good tool for this.

Let us suppose we are in the case where we are trying to classify our data into either class 0 or 1. We wish to model

$$p(c = 1|X) = 1 - p(c = 0|X).$$

This can be achieved by performing linear regression on the log odds of the probabilities we wish to model.

If we let $\pi = p(c = 1|X)$ then the log odds of π is the value

$$\ell(\pi) = \log\left(\frac{\pi}{1 - \pi}\right).$$

Logistic regression supposes a linear relationship between the log odds and the data. That is,

$$\ell(\pi) = \beta_0 + \sum_{i=1}^{d} \beta_i x_i.$$

If we then sove for π, we get

$$\pi = \frac{1}{1 + \exp\left(\beta_0 + \sum_{i=1}^{d} \beta_i x_i\right)}.$$

This is particularly significant because the function on the right is known in machine learning as the *sigmoid activation function*,

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

Thus we do logistic regression by performing linear regression on the log odds of a binary dependent variable. The end result is a map of $\pi = p(c = 1|x)$ as $\pi = \sigma \circ a(x)$ for some linear function $a(x)$. In this sense, we may say that a perceptron which uses a sigmoidal activation is performing logistic regression.

The softmax function is a map $\sigma : \mathbb{R}^D \to S_D$. As a reminder, the probability simplex $S_D = \{x \in \mathbb{R}^D | \sum_i x_i = 1, \ x_i \geq 0 \ \forall i\}$. The softmax function is given by

$$\sigma_i(\boldsymbol{x}) = \frac{e^{x_i}}{\sum_{j=1}^{D} e^{x_j}}. \tag{2.4}$$

In the case that $D = 2$, the softmax function is simply the sigmoidal activation function. In this sense, the softmax function may be considered as a multivariate extension of the logistic regression model.

The softmax function is so named because it smoothly approximates the arg max function. Given a vector $\boldsymbol{x} \in \mathbb{R}^D$, we may represent the arg max function in the following manner.

$$\arg\max_i(\boldsymbol{x}) = \begin{cases} 1 & \text{if } x_i = \max_i(\boldsymbol{x}) \\ 0 & \text{otherwise} \end{cases}$$

In this form, it is clear that $\arg\max : \mathbb{R}^D \to \{0, 1\}^D$ is a locally constant function. We also see that softmax smoothly approximates argmax, in the following sense. If for a given $\boldsymbol{x}$ some coordinate x_i satisfies $x_i \gg x_j \; \forall j \neq i$, then $\sigma_i(\boldsymbol{x}) \approx 1$ and $\sigma_j(\boldsymbol{x}) \approx 0$ for all $j \neq i$.

However, if for some i, j, $x_i = x_j \gg x_k \; \forall k \neq i, j$, then $\sigma(\boldsymbol{x}) \approx \frac{1}{2} \arg\max(\boldsymbol{x})$. In a similar way, $\sigma(\boldsymbol{x})$ varies continuously over all of $\mathbb{R}^D$. When argmax indicates more than one index for the maximum, then softmax will distribute the max assignment equally to each of the indices.

The advantage of this comparison is that some of the properties of the softmax function become immediately apparent. First, softmax is projective, so for any $\lambda \in \mathbb{R}$, $\sigma(\lambda \boldsymbol{x}) = \sigma(\boldsymbol{x})$. Second, if we define $\boldsymbol{c} = c \cdot \mathbf{1}_d = (c, c, \dots, c)^\mathsf{T}$ then $\sigma(\boldsymbol{x} + \boldsymbol{c}) = \sigma(\boldsymbol{x})$, which is a type of translation invariance.

In regard to both of these properties, we mention the log-sum-exp trick used frequently in computation of the softmax function. The point is that often in machine learning applications one may be required to use data types that will easily cause underflow and overflow errors. For example, if x represents a single precision floating point number and $x < -103$, then $\mathrm{fl}(\log(\mathrm{fl}(e^x))) = \mathrm{fl}(\log 0) = -\infty$, even though it should be the case that $\log(e^x) = x$. Such a situation might be encountered often in neural network applications.

The log-sum-exp function $\mathrm{lse} : \mathbb{R}^D \to \mathbb{R}$ is defined by

$$\mathrm{lse}(\boldsymbol{x}) = \log\left(\sum_i e^{x_i}\right).$$

It is worth noting that $\nabla \mathrm{lse}(\boldsymbol{x}) = \sigma(\boldsymbol{x})$, so that softmax represents the gradient of the log-sum-exp function. A property of the lse function is that

$$y = \log\left(\sum_{i=1}^D e^{x_i}\right) = \log\left(\sum_{i=1}^D e^a e^{x_i - a}\right) = \log\left(e^a \sum_{i=1}^D e^{x_i - a}\right).$$

If $a \in \mathbb{R}$

$$y = a + \log \left(\sum_{i=1}^{D} e^{x_i - a} \right).$$

For the softmax function, this amounts to translation invariance as mentioned above. In other words,

$$\sigma_i(\boldsymbol{x}) = \frac{e^{x_i - a}}{\sum_j e^{x_j - a}}.$$

A common value to use to avoid overflow is $a = \max_i x_i$. This also tends to avoid loss of precision due to underflow.

Finally, in connection to the lse trick, it is noted that

$$\log(\sigma_i(\boldsymbol{x})) = x_i - \log \left(\sum_j e^{x_j} \right).$$

So that one may implement the shift via the translation property of the lse function. However, this tends to exaggerate numerical inaccuracies we are looking to avoid. For further details one may refer to [BHH19].

2.4 Basic Neural Nets

Neural networks are a class of supervised machine learning algorithms used for various learning and inferential tasks. The first practical neural net was the perceptron proposed by Rosenblatt in 1958 [Ros58] who was inspired by earlier work of McCulloch and Pitts [MP43]. More recently we have seen many exciting advances in the use of neural networks, including deep learning, reinforcement learning, and the use of Neural nets for transfer learning. Here we will briefly discuss a basic neural network architecture, the multilayer perceptron, and the typical method for training neural nets, back propagation. Because of the techniques use to make inference, neural networks that use the processes described here are called *feedforward* neural networks.

2.4.1 Single Layer Perceptron

As first conceived by Rosenblatt, the perceptron was intended to be a machine, rather than a program. To this end, the inputs are intended to be binary vectors, and the output is also binary. Shortly after Rosenblatt described the algorithm for the perceptron, Minsky and Papert [MP90] showed that a

single perceptron is unable to calculate the XOR function. While this proved that a single perceptron is not good for general computation, it turns out that a more general form, the multi layer perceptron, is a provably universal function approximator, as shown by Hornik [Hor91] and expanded on by Lesho *et al.* [LLPS93]. We start with the regular perceptron as a simplified example.

The perceptron is a type of linear discriminant. Inspired by one model of human neurons, it is 'on' if the output is large enough, and 'off' otherwise. The output is determined as a linear combination of data features. A figure of a simple perceptron is seen in figure 2.1
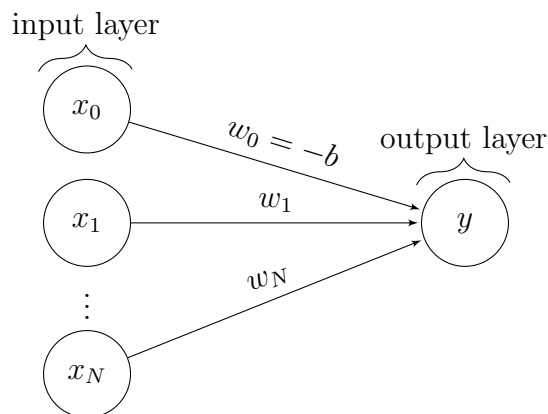
Figure 2.1: The perceptron consists of N input units and at least one output unit. All units are labeled according to their output: $y = H(z)$ in the case of the output unit; x_n in the case of input units. The input values x_n are propagated to the output unit using a weighted sum, in other words $z = \boldsymbol{w}^\mathsf{T}\boldsymbol{x}$. An additional input value $x_0 := 1$ is used to include the biases as weights.

To describe this more precisely, let $\boldsymbol{X} = \{\boldsymbol{x}_n\}_{n=1}^N$ be the data, and $\boldsymbol{T} = \{t_n\}_{n=1}^N$, $t_n \in \{0,1\} \ \forall n$ be the labels. We will refer to the points $\{(x^n, t^n)\} \subset \boldsymbol{X} \times \boldsymbol{T}$ as the training set. Then the perceptron algorithm seeks to determine the class of a new data point $\boldsymbol{x}$ by using a function of the form

$$y = H(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}). \tag{2.5}$$

Here $\boldsymbol{w}$ is a vector of real weights determined through training and $H(v)$ is the Heaviside step function

$$H(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{if } v < 0 \end{cases}$$

In practice, a bias term b is added to equation 2.5, so that it becomes $y = H(\boldsymbol{w}^\mathsf{T}\boldsymbol{x} + b)$. This is most usually done by adding an extra feature to all of our data points while training. This extra feature is always set to 1. An additional weight is also appended to the weight vector $\boldsymbol{w}$ so that the

model still looks like 2.5. The effect of bias is to change where the perceptron activates, and makes it harder to 'fire' (*i.e.* give a positive output for a given input). This also has the effect of allowing the model to center the data, as would happen with a bias in the regression setting.

The act of training for a perceptron is choosing weights $\boldsymbol{w}$ so that $H(\boldsymbol{w}^\mathsf{T}\boldsymbol{x}_n) = t_n \ \forall (x_n, t_n) \in \mathcal{X} \times \mathcal{T}$. The hope is that this will generalize well to data points not seen in the training set. The point emphasized in the Minsky book [MP90] is that training will only work if the two classes in the training set are linearly separable. This only applies to single layer perceptrons, as one can create a NAND gate from a single layer perceptron, so using multiple perceptrons one could conceivably approximate any function to arbitrary precision.

The perceptron algorithm also includes methods for training, but these are not in common use today. More modern architecture use the backpropagation algorithm, which requires gradient descent. The problem with perceptrons in that context is that the derivative of the Heaviside step function is zero.

The use of the Heaviside function in the perceptron is called an activation function. In more modern neural networks, activation functions with non-zero derivative allow use of backpropagation and gradient descent. The discussion in section 2.3 mentions the sigmoidal activation function. As discussed there, if we replace the Heaviside step function with the sigmoid function, then we are actually performing logistic regression. We will refer to perceptrons using the sigmoidal activation function as sigmoidal neurons.

2.4.2 Multilayer Perceptron as a Multi-Class Classifier

The multilayer perceptron (MLP) is actually built out of several sigmoidal neurons. The MLP we describe has a single hidden layer, which means that the output of one sigmoidal neuron becomes the input for the next neuron. Figure 2.2 below gives an example of a single hidden layer MLP.

Neural networks such as the MLP are often called feedforward neural networks. This is because at each stage of computation information is only passed forward toward the output nodes. The general pattern of feedforward nets is that each node passes a weighted sum of previous nodes through an activation function. We have already mentioned two activation functions, Heaviside and sigmoidal. Other common activation functions are hyperbolic tangent, rectified linear unit (RELU), and sigmoid linear unit [EUD17].

In the case of the MLP, each of the hidden layer nodes work as a sigmoidal neuron *i.e.* $z_l = \sigma(\boldsymbol{w}_{1l}^\mathsf{T}\boldsymbol{x})$.
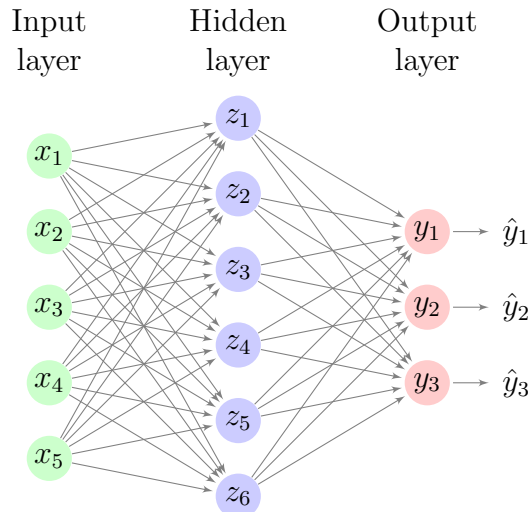
Figure 2.2: A graph model of a single hidden layer MLP with 5 inputs, 3 output and 6 hidden layer nodes. The output of each layer is determined by composing a linear map with a nonlinear map. The general form of this nonlinear map is determined at the outset, though it may have trained parameters. The linear map is determined through a set of learned weights.

Likewise, each of the output layers can be expressed as a sigmoidal neuron, $y_k = \sigma(\boldsymbol{w}_{2k}^{\mathsf{T}} \boldsymbol{z})$. This view is not consistent with a desire to express the output in terms of a probability. This is particularly important if we wish to use the MLP as a multi-class classifier, as we wish to interpret the outputs as $P(k_n = k | \boldsymbol{x}_n)$.

While each individual output is a log odds in the sense of logistic regression, together we cannot interpret each individual y_k as a probability. In particular, we have no guarantee that $\sum_k y_k = 1$. To fix this, we may pass the outputs y_k through the softmax function *i.e.*

$$\hat{y}_k = \frac{e^{y_k}}{\sum_{j=1}^{K} e^{y_j}} \tag{2.6}$$

Passing the output through a softmax function has the effect of guaranteeing that $\sum_k y_k = 1$, but if we are to interpret $\hat{y}_k$ as a probability, then it must be regarded as a Gibbs distribution. From a mathematical standpoint this changes the interpretation of y_k to an approximation of an energy function. To be consistent with this interpretation, we must be careful in our choice of cost function. This becomes more apparent when we consider the role of backpropagation in training.

2.4.3   Backpropagation

Backpropagation (BP) was modernly popularized as a method for training neural networks by Rumelhart *et al.* [RHW86] in 1986. Similar training methods had been used earlier, *e.g.* Linnainmaa [Lin76], in the context of optimization. Backpropagation for use in neural network training was first used by Paul Werbos in his 1974 thesis [Wer94]. Brief histories of the development of BP can be found in Griewank *et al.* [GW08], Schmidhuber [Sch15], and Goodfellow *et al.*,[GBC16, see ch6.6]. The practical development and popularization of BP led to a very active period of research in multilayer neural networks.

While BP is often described as 'just gradient descent', it is the case that BP actually refers to a specific method for calculating the gradient of the loss function with respect to the weights. Backpropagation relies heavily on the chain rule, and the architecture of perceptron inspired neural networks. We will use a simple neural network model to help describe BP. We draw most of the inspiration for this example from the book by Goodfellow *et al.* [GBC16, ch6.5]

The goal of BP is to adjust the weights by subtracting off the gradient of the cost function with respect to the weights. The key realization of BP is that with the correct network setup, this may be done by passing information backwards through the graph.

Recall that the typical role of a layer in a feedforward neural net is to pass forward predictions based on information from the previous layer. The role of layer ℓ during training with backpropagation is to pass forward the predictions as usual. Then during the backward pass layer ℓ updates its own weights W_ℓ using gradient descent and the chain rule. Finally the layer passes back the new gradient of the loss with respect to the output $\boldsymbol{z}_{\ell-1}$of the previous layer. Figure 2.3 gives a high level model of this process.

While the diagram in figure 2.3 implies that the backpropagation algorithm only uses chain rule, in reality it may be a bit more complicated. Since $\sigma_\ell : \mathbb{R}^{d_{\ell-1}} \to \mathbb{R}^{d_\ell}$ is a non linear multivariate function, taking derivatives can be complicated. Further, the parameters $\boldsymbol{W}_\ell$ might be part of a higher dimensional linear map (e.g matrix, tensor). For such situations we need to be more careful in calculating gradients. While many references address this problem, [The05] offers a good treatment of the chain rule in such situations. Section 2.4.4 covers this in more detail.

Finally it is worth noting that in many situations, backpropagation can be simply calculated in a coordinate manner. This is because much of the structure of neural nets gives an implied coordinate system to each layer. It is also a good reason that one must be careful in the choice of cost function.

Layer ℓ

$$\boldsymbol{z}_{\ell-1}$$

$$\sigma_\ell = \boldsymbol{f}_\ell(\langle \boldsymbol{W}_\ell, \cdot \rangle)$$

$$\boldsymbol{z}_\ell = \boldsymbol{\sigma}_\ell(\boldsymbol{z}_{\ell-1})$$

$$\frac{\partial L}{\partial \boldsymbol{z}_{\ell-1}} = \frac{\partial L}{\partial \boldsymbol{z}_\ell}\frac{\partial \boldsymbol{z}_\ell}{\partial \boldsymbol{z}_{\ell-1}}$$

$$\frac{\partial L}{\partial \boldsymbol{z}_\ell}$$

$$\nabla_{\boldsymbol{W}_\ell}\sigma_\ell = \frac{\partial L}{\partial \boldsymbol{z}_\ell}\frac{\partial \boldsymbol{z}_\ell}{\partial \boldsymbol{W}_\ell}$$
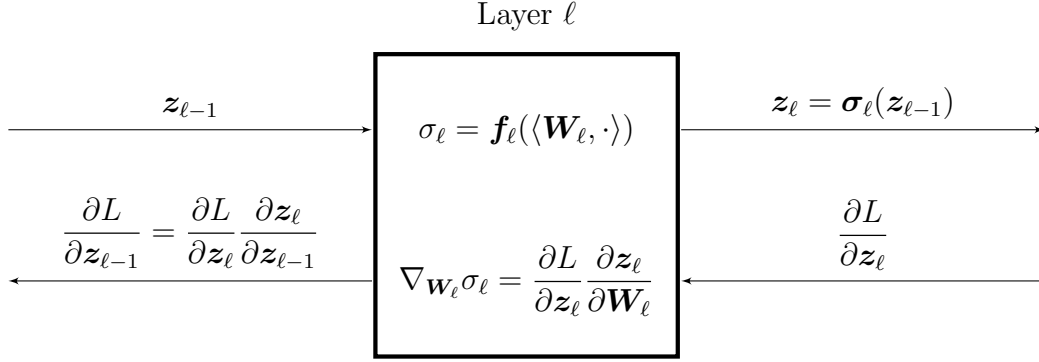
Figure 2.3: A model of a single layer in a neural network. For the feedforward pass it calculates $\boldsymbol{z}_\ell = \boldsymbol{\sigma}_\ell(\boldsymbol{z}_{\ell-1}) = \boldsymbol{f}_\ell(\langle \boldsymbol{W}_\ell, \boldsymbol{z}_{\ell-1}\rangle)$. On the backward pass it calculates $\frac{\partial L}{\partial \boldsymbol{W}_\ell}$ and $\frac{\partial L}{\partial \boldsymbol{z}_{\ell-1}}$. The layer uses $\frac{\partial L}{\partial \boldsymbol{W}_\ell}$ to update its own weights with gradient descent. The layer passes $\frac{\partial L}{\partial \boldsymbol{z}_{\ell-1}}$ to the previous layer to use.

When an appropriate cost function is chosen, backpropagation becomes a quick operation. This helps explain why backpropagation is so popular in recent neural net training.

2.4.4 Derivative Notation

Before backpropagation resurfaces in chapter 4, this section establishes important notation standards. In the literature [AR67, Man12, MN85, The05] there are several different notations used for differentiation of functions $f : \mathbb{R}^n \to \mathbb{R}^m$. Each author seems to prefer their own notation, and while these notations often overlap, reading various papers quickly becomes confusing without precise communication. This section seeks to establish a reference for derivative notation to be used through the remainder of chapter 4.

In [Ryc19], care is taken to distinguish the Frechét (or contravariant) derivative of a function from the gradient (or covariant derivative) of the same function. Given a vector valued function $f : \mathbb{R}^n \to \mathbb{R}^m$, the Frechét derivative of f at x is the linear map $A_{f(x)} : \mathbb{R}^n \to \mathbb{R}^m$ defined by

$$\lim_{\|\boldsymbol{h}\|\to 0} \frac{\left\| f(\boldsymbol{x}+\boldsymbol{h}) - f(\boldsymbol{x}) - A_{f(x)}(\boldsymbol{h}) \right\|}{\|\boldsymbol{h}\|} = 0. \tag{2.7}$$

Provided such a map exists, it is unique. In particular if $n, m < \infty$, and $\frac{\partial f}{\partial \boldsymbol{x}} = \left(\frac{\partial f_i}{\partial x_j}\right)_i^j$ is the matrix of partial derivatives (or Jacobian matrix) of f, then $A_{f(x_0)}(h) = \frac{\partial f}{\partial \boldsymbol{x}}\big|\, x_0 \cdot h$. It is worth noting that the existence of a continuous Jacobian matrix for f guarantees f has a Frechét derivative. The reverse implication is not true, f can have a Frechét derivative but not have continuous partials everywhere.

The Frechét derivative of f at the point $\boldsymbol{x} \in \mathbb{R}^n$ is often denoted by $Df(\boldsymbol{x})$, a convention which this dissertation will follow. The notation $Df(\boldsymbol{x})[\boldsymbol{h}]$ works when necessary to discuss both the point $\boldsymbol{x}$ at which the derivative is being taken, and the direction $\boldsymbol{h}$ on which it is acting. In this sense it may be said that Df is a map $Df : \mathbb{R}^n \to L(\mathbb{R}^n, \mathbb{R}^m)$. Here, $L(V, W)$ is the collection of all linear maps from one real vector space V to another real vector space W.

Given this notation for the Frechét derivative, denote by $\nabla f(\boldsymbol{x})$ the gradient of f. A function $f : \mathbb{R}^n \to \mathbb{R}^m$ only has a gradient if $m = 1$. In this case the gradient is a map $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ such that for all $\boldsymbol{x}, \boldsymbol{h} \in \mathbb{R}^n$

$$Df(\boldsymbol{x})[\boldsymbol{h}] = \langle \boldsymbol{h}, \nabla f(\boldsymbol{x}) \rangle. \tag{2.8}$$

Here the angle brackets denote the standard euclidean inner product on $\mathbb{R}^n$. Because the gradient of a scalar valued function is a vector valued map, it is possible for ∇f to be differentiable. In this case the resulting derivative is called the *Hessian* of f and will be denoted by $\nabla^2 f$.

An analog of the gradient for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \geq 1$ is the adjoint operator of Df. The adjoint operator of any linear map $A \in L(\mathbb{R}^n, \mathbb{R}^m)$ is the map $A^* \in L(\mathbb{R}^m, \mathbb{R}^n)$ such that $\forall \, x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$,

$$\langle A(x), y \rangle_{\mathbb{R}^m} = \langle x, A^*(y) \rangle_{\mathbb{R}^n}.$$

Since $Df : \mathbb{R}^n \to L(\mathbb{R}^n, \mathbb{R}^m)$, the definition of an adjoint operator $D^*f : \mathbb{R}^n \to L(\mathbb{R}^m, R^n)$ depends on the point $x \in \mathbb{R}^n$ at which it is evaluated. Thus D^*f is defined by

$$\langle Df(x)[h], u \rangle_{\mathbb{R}^m} = \langle h, D^*f(x)[u] \rangle_{\mathbb{R}^n}$$

holding $\forall \, x, h \in \mathbb{R}^n, y \in \mathbb{R}^m$. Aside from the reliance of the definition on the inner product, the adjoint derivative

When dealing with linear maps between $\mathbb{R}^n$ and $\mathbb{R}^m$, all the maps can be recognized as matrix maps; in this case the adjoint is the transpose of the matrix, *i.e.* $A^* = A^\intercal$. This is not true for general vector spaces V, W over $\mathbb{R}$ as there may be linear maps which have more structure than $n \times m$ matrix maps between real vector spaces. Important examples are when V, W are matrix algebras over $\mathbb{R}$, and tensor algebras of such vector spaces.

Thus it will not be assumed *a priori* that the derivative maps Df, D^*f are matrix maps. In fact, for some of the functions used in chapter 4, Df, D^*f will not just be linear maps, but multilinear maps, or tensors. In this case there are still analogs of adjoint operators but more care must be taken

in describing them. Discussion of such details will come when necessary.

Since backpropagation does gradient descent, it must calculate the gradient of the loss L with respect to weights W. In figure 2.3, this is shown to be done via the chain rule, but in the general case more care must be applied. The following lemma makes this much easier.

Lemma 2.1. *Let U, V be real Riemannian Manifolds and $f : V \to \mathbb{R}$ and $g : U \to V$ be smooth maps. Then if $h = f \circ g$, we have that*

$$\nabla h = D^* g[\nabla f \circ g]$$

where $\nabla h, \nabla f$ are the gradients of h and f respectively, and $D^* g$ represents the adjoint linear operator of Dg with respect to the metrics on TV and TU.

Proof. This is lemma 4.1 of Theis [The05]. This proof adapts it for use in this dissertation.

First, let $u \in U$ and $x \in T_u U$ be arbitrary. Because f, g are smooth, they induce maps $Dg(u) : T_u U \to T_{g(u)} V$ and $Df(g(u)) : T_{g(u)} V \to \mathbb{R}$. It is given (by definition) that $Dh(u)[x] = \langle \nabla h(u), x \rangle_{T_u U}$. Further, it is clear that $Dh(u) : T_u U \to \mathbb{R}$ is given by $Dh(u)[x] = D(f \circ g)(u)[x] = Df(g(u))[Dg(u)[x]]$.

Now $Df(g(u))[Dg(u)[x]] = \langle \nabla f(g(u)), Dg(u)[x] \rangle_{T_{g(u)} V}$. Then for the linear operator $Dg(u) : T_u U \to T_{g(u)} V$, the adjoint linear operator $D^* g$ is defined by the equation $\langle y, Dg(u)[x] \rangle_{T_{g(u)} V} = \langle D^* g(u)[y], x \rangle_{T_u U}$ for $x \in T_u U$ and $y \in T_{g(u)} U$. This gives

$$Df(g(u))[Dg(u)[x]] = \langle \nabla f(g(u)), Dg(u)[x] \rangle_{T_{g(u)} V} = \langle D^* g(u)[\nabla f(g(u))], x \rangle_{T_u U}.$$

So that $\langle \nabla h(u), x \rangle_{T_u U} = \langle D^* g(u)[\nabla f(g(u))], x \rangle_{T_u U}$, and as u, x were arbitrary, the theorem is proved.

$\square$

A key aspect of the proof above is the use of the metric on U and V. This allows identification of tangent spaces to their duals, $TU \cong TU^*$ and $TV \cong TV^*$, to define Dg^* appropriately. this is not surprising as the definition for the gradient of f in equation 2.8 is closely tied to the inner product on the domain of f. It follows that wherever derivatives will be used in this dissertation, the appropriate choice of metric (and thus inner product) on the tangent space will be essential.

Because all the spaces involved can be embedded in $\mathbb{R}^n$ for some n, many calculations in chapters 3 and 4 use the inner product on $\boldsymbol{M} = L(\mathbb{R}^N, \mathbb{R}^K)$, defined by the Frobenius inner product $\langle A, B \rangle = \text{tr}(A^\intercal \cdot B)$. The map vec : $\boldsymbol{M} \to \mathbb{R}^{KN}$ given by stacking the columns of M is a diffeomorphism, and the Frobenius inner product on $\boldsymbol{M}$ is equivalent to the standard euclidean inner product on $\mathbb{R}^{KN}$. In

short, the following diagram commutes.

$$
\begin{array}{ccc}
\boldsymbol{M} \times \boldsymbol{M} & \xrightarrow{\text{vec}} & \mathbb{R}^{KN} \times \mathbb{R}^{KN} \\
& \searrow{\scriptstyle\text{Frob}} & \downarrow{\scriptstyle\text{euc}} \\
& & \mathbb{R}
\end{array}
\tag{2.9}
$$

Here Frob and euc represent the inner product (metric) on each of the spaces.

Finally, it is worth mentioning that when necessary, such as in the proof of lemma 2.1, discussions reference the tangent space TU of a Riemannian manifold U. Since all the spaces here are generally euclidean, such notational references help distinguish the space U from $T_u U$, vectors tangent to some point $u \in U$. The full strength of considering these spaces as manifolds will not be used.

2.5 Responsible Clustering Algorithms

2.5.1 K-means algorithm

The K-means algorithm has been in use for several decades. Though the first mention of the algorithm by name was given by MacQueen in 1967 [Mac67] the idea had been around for some time(insert steinhaus ref). The standard algorithm was used at Bell Labs in 1957 [Llo82] for pulse code modulation. Pollard [Pol81, Pol82] showed that the K-means algorithm is consistent in a very precise sense. Today the algorithm is used in many applications [SC96, CS96], and can be found in many good books on machine learning. [Bis95, Mac02, Bis06, HTF09, DFO20]

The outline below is primarily compiled from chapters 20 and 22 of MacKay [Mac02], though the Bishop and Deisenroth books [Bis06, DFO20] are also a good reference.

The K-means algorithm is used for vector quantization and for data clustering. It is so named because it separates data points into K distinct groups, each characterized by a 'mean' $\boldsymbol{m}_k$, $k = 1, \ldots, K$. In the case that the K-means algorithm is being used for clustering, these means are the cluster centers and each data point is assigned to the closest mean. In this situation it is the case that

$$
\boldsymbol{m}_k = \frac{\sum_{n=1}^{N} r_k^n \boldsymbol{x}^{(n)}}{N_k}
$$

where

$$r_k^n = \begin{cases} 1 & \text{if } \boldsymbol{x}^{(n)} \text{ is closest to } \boldsymbol{m}_k \\ 0 & \text{otherwise} \end{cases}$$

and $N_k = \sum_{n=1}^{N} r_k^n$. In other words, the means $\boldsymbol{m}_k$ are literally the means of the assignment clusters.

Of course we cannot understand 'closest' without first defining a distance function on the underlying data space. For the original K-means algorithm, the distance was the manhattan distance, though we will use a scaled square of the euclidean distance:

$$d(\boldsymbol{x}, \boldsymbol{y}) = \frac{1}{2} \sum_i (x_i - y_i)^2.$$

It is worth noting that the choice of distance in this sense is somewhat arbitrary. In most descriptions of the algorithm, euclidean distance is used to aid visualization.

To implement the K-means algorithm, start with K distinct means. A common practice is to use randomly sampled data points $\{\boldsymbol{m}_1 = \boldsymbol{x}^{(n_1)}, \dots, \boldsymbol{m}_K = \boldsymbol{x}^{(n_K)}\}$. Then iteratively do the following:

1. For each data point, $\boldsymbol{x}^{(n)}$, set $\hat{k}_n = \arg\min_k d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_k)$.

2. Set $r_{\hat{k}_n}^n = 1$, for each n. Set all other $r_k^n = 0$

3. Calculate $N_k = \sum_{n=1}^{N} r_k^n$ and
$$\boldsymbol{m}_k^{new} = \frac{\sum_{n=1}^{N} r_k^n \boldsymbol{x}^{(n)}}{N_k}.$$
   If $N_k = 0$, $\boldsymbol{m}_k^{new} = \boldsymbol{m}_k$.

4. If $d(\boldsymbol{m}_k, \boldsymbol{m}_k^{new})$ is within a predefined tolerance for every k, stop. Otherwise set $\boldsymbol{m}_k = \boldsymbol{m}_k^{new}$ and repeat at step 1. Equivalently, convergence happens when no assignments $\hat{k}_n$ change.

In the literature, it is common to see steps one and two listed as the assignment step, and steps three and four as the update step.

The easiest way to see that this algorithm terminates is to recognize that the function

$$L := \sum_{n=1}^{N} d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_{\hat{k}_n})$$

either stays the same or decreases at each update step. In this sense, L acts as a Lyapunov function for the K-means algorithm.

One problem with k-means clustering that is particularly relevant to this dissertation comes when the clusters do not have equal representation in the data. As an example, [XWC09] give examples of data sets where kmeans produces equal sized clusters, even though their true sizes relative to the number of data points is different.

One way to address the problem is with *soft responsibility*

$$r_k^{(n)} = \frac{\exp\left(-\beta d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_k)\right)}{\sum_i \exp(-\beta d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_i))}. \tag{2.10}$$

The idea behind soft responsibility is that each cluster center is partially responsible for each data point. The amount of responsibility $r_k^{(n)}$ of $\boldsymbol{m}_k$ for the data point $\boldsymbol{x}^{(n)}$ ought to be inversly proportional to $d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_k)$. That is, cluster centers closer to data points have greater responsibility for those data points.

The factor β included here is an inverse temperature, or stiffness parameter, and it can be set at the beginning or through training. In further refinements of the soft K-means algorithm, each cluster center has its own β_k, and at each iteration $\beta_k = \dfrac{1}{\sigma_k^2}$ where σ_k^2 is the weighted sample variance of the data points assigned to cluster k.

$$\sigma_k^2 = \frac{\sum_n r_k^{(n)} d(\boldsymbol{x}^{(n)}, \boldsymbol{m}_k)^2}{N_k}. \tag{2.11}$$

Further refinements to this algorithm can be found in MacKay's book, and were also developed in the software *AutoClass*. [Mac02, SC96, CS96]

2.5.2 Expectation Maximization

Upon close inspection, it can be seen that the soft K means algorithm is very similar to the Expectation Maximization (or EM) algorithm for Gaussian Mixture Models. We give a brief overview of expectation maximization and the relation of this algorithm to responsibility as discussed in section 2.5.1. The discussion below roughly follows discussions available in Bishop and other sources [DFO20, Bis06, HTF09].

EM was first described for the general case in a paper by Dempster *et al.* [DLR77]. Though the paper mentions that Hartley [Har58], Baum [BPSW70] and others [Woo70, Sun74, Sun76] had already used similar techniques in special circumstances.

The basic idea behind EM is to add hidden or latent variables to a modeling problem in such a

way that maximum likelihood estimation is made easier. The heuristic of this approach is that the latent variables are simply unobserved features of the data.

To be more precise, suppose we are given data x and we want to fit a model with parameters θ for the pdf $p(x|\theta)$ using maximum likelihood estimation. In many cases this is an intractable problem that can be simplified by considering the conditional pdf

$$p(x|\theta, z). \tag{2.12}$$

Now as z are latent variables, we must place a prior $p(z)$ on the distribution of z. Using 2.12, and the law of total probability we may write

$$p(x|\theta) = \int_{\mathcal{Z}} p(x|\theta, z)p(z)\, dz. \tag{2.13}$$

Where the integral is taken over the space of possible latent variables.

In practice, the integral in 2.13 can easily become intractable. The trick is to choose z and $p(z)$ in a manner that avoids this difficulty. The EM algorithm is an iterative procedure that relies on repeating two steps until convergence. It is the use of these two steps which help us choose z and $p(z)$. These steps are also from which the algorithm receives its name.

While EM is more broadly defined and used than what we will discuss, we will consider the case where we are using EM to fit a Gaussian mixture model for simplicity. In this case, let

$$\mathcal{D} = \{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(N)}\}$$

$$f_k(\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k) \quad k = 1, \ldots, K \tag{2.14}$$

$$p(z = k) = \pi_k, \quad \sum_k \pi_k = 1 \tag{2.15}$$

Then to use the EM algorithm, set parameters to some initial values: $\pi_k = \pi_k^0$, $\mu_k = \mu_k^0$, $\Sigma_k = \Sigma_k^0$. Then apply the following steps:

1. *Expectation* step: Set
$$r_k^n = \frac{\pi_k f_k(\boldsymbol{x}^{(n)})}{\sum_{j=1}^{K} \pi_j f_j(\boldsymbol{x}^{(n)})} \tag{2.16}$$

2. *Maximization* step: Set

$$N_k = \sum_{n=1}^{N} r_k^n \tag{2.17}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{2.18}$$

$$\boldsymbol{\mu}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_k^n \boldsymbol{x}^{(n)} \tag{2.19}$$

$$\boldsymbol{\Sigma}_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_k^n (\boldsymbol{x}^{(n)} - \boldsymbol{\mu}_k^{new})(\boldsymbol{x}^{(n)} - \boldsymbol{\mu}_k^{new})^\intercal \tag{2.20}$$

3. If all of π_k, $\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$ are close enough to their new counterparts, stop. Otherwise, set $\pi_k = \pi_k^{new}$, $\boldsymbol{\mu}_k = \boldsymbol{\mu}_k^{new}$, $\boldsymbol{\Sigma}_k = \boldsymbol{\Sigma}_k^{new}$, then repeat steps 1 and 2.

We note that this is a specific implementation of the EM algorithm in the case we suspect that a Gaussian mixture model works well for the underlying data. Other changes that may be made are choosing different parameterized distributions $f_k(\boldsymbol{x})$, or even adapting the algorithm for completely different models. For example the Baum-Welch algorithm [BPSW70], is tailored specifically to using EM on Hidden Markov Models. More information about generalizations of EM may be found in Hastie *et al.* [HTF09, p.276].

Other than ease of explanation, we use the preceding description of EM to illustrate the similarity of EM and K-means algorithms. For example, fix $\boldsymbol{\Sigma}_k$ to be isotropic with variance as described in 2.11, then equations 2.10 and 2.16 agree on the definition of responsibility.

2.6 Clustering with Mixture Models

Recalling the discussion in section 2.1, this section will consider one way to recover the mixing components given a reasonable approximation of the underlying cluster distributions. While such approximations are frequently incalculable, it is instructive to explore situations where calculation is possible. Consider the following situation inspired by *Information Theory, Inference, and Learning Algorithms* [Mac02], and covered in notes from a course taken in 2016 [Ryc16].

Using Bayes' rule, this section proposes an optimization strategy for recovering the parameters $\{\pi_k^*\}_{k=1}^{K}$. Given some data D, this strategy considers the likelihood that the labels chosen were $\{k_n\}_{n=1}^{N}$, given some prior distribution of the labels as $\{\pi_k\}_{k=1}^{K}$. In other words, the prior is given by $P(k_n =$

$k, \boldsymbol{\theta}_k) = \pi_k$, with or without considering the data. A potentially good uninformative prior would be that $\pi_k = \frac{1}{K}$ for every k.

Bayes rule gives:

$$
\begin{aligned}
P(k_n = k | \{x_n\}, \{\pi_k\}) &= \frac{P(x_n | k_n = k) P(k_n = k)}{\sum_{k'} P(x_n | k_n = k') P(k_n = k')} \\
&= \frac{\pi_k f_k(x_n)}{\sum_{k'} \pi_{k'} f_{k'}(x_n)}.
\end{aligned}
\tag{2.21}
$$

Assuming independence of the samples in $D = \{x_n\}$, the joint distribution of $\boldsymbol{x} = \{x_n\}$ and possible labels $\boldsymbol{k} = \{k_n\}$ is

$$
P(\boldsymbol{x}, \boldsymbol{k} | \{\pi_k\}) = \prod_{n=1}^{N} P(k_n = k) P(x_n | k_n = k) \sim \prod_{n=1}^{N} \pi_{k_n} f_{k_n}(x_n).
\tag{2.22}
$$

Note that dependence of equation (2.22) on the parameters $\boldsymbol{\theta}_k$, $k = 1, \ldots, K$ is suppressed.

Since practical methods of knowing the true labels of points x_n are difficult to obtain, perform marginalization to get

$$
\begin{aligned}
P(\{x_n\} | \{\pi_k\}) &= \sum_{(k_1, k_2, \ldots, k_N)} \prod_n \pi_k f_{k_n}(x_n) \\
&= \prod_n \left(\sum_k \pi_k f_k(x_n) \right) \\
&= \prod_n P(x_n | \{\pi_k\})
\end{aligned}
$$

where $P(x_n | \{\pi_k\}) = \sum_k \pi_k f_k(x_n)$. It is informative to compare equation (2.21) with the formula (2.16) for responsibility in section 2.5.2.

The goal of this strategy is to find the most likely values of $\{\pi_k\}$, given data $\{x_n\}$. This strategy assumes complete lack of knowledge, *i.e.* the prior distribution of $\{\pi_k\}$ is uniform on the standard probability simplex

$$
S_K := \left\{ \{\pi_k\}_{k=1}^{K} : 0 \leq \pi_k \leq 1; \sum_{k=1}^{K} \pi_k = 1 \right\}.
\tag{2.23}
$$

Another way of stating this idea is to say that the prior is the Dirichlet distribution with parameters $\alpha_i = 1$ for $i = 1, \ldots, K$, *i.e.* the flat Dirichlet distribution.

In this situation Bayes' rule gives

$$P(\{\pi_k\}|\{x_n\}) = \frac{P(\{x_n\}|\{\pi_k\})P(\{\pi_k\})}{\int \int_{...} \int P(\{x_n\}|\{\pi_k\})P(\{\pi_k\})\, d\boldsymbol{\pi}}.$$

Since the prior $P(\{\pi_k\})$ is uniform and does not depend on $\boldsymbol{\pi}$, it cancels out to get

$$P(\{\pi_k\}|\{x_n\}) = \frac{P(\{x_n\}|\{\pi_k\})}{\int \int_{...} \int_{S_K} P(\{x_n\}|\{\pi_k\})\, d\boldsymbol{\pi}}. \tag{2.24}$$

Note here that if performing a maximum *a posteriori* (MAP) estimate of $\boldsymbol{\pi}^*$ at this point, it would be equivalent to finding a maximum likelihood estimator for $\boldsymbol{\pi}^*$.

While the marginal probability in the denominator of 2.24 is difficult to compute, looking at the log likelihood creates numerous opportunities. The strategy is to define

$$\ell := \ln P(\{x_n\}|\{\pi_k\}) = \sum_n \ln P(x_n|\{\pi_k\}) \tag{2.25}$$

and then maximize the likelihood ℓ on the simplex S_K. In other words, define an estimator $\hat{\boldsymbol{\pi}}$ of $\boldsymbol{\pi}^*$ by

$$\{\hat{\pi}_k\} = \arg\max_{S_K} \ell.$$

The problem of finding the 'correct' discrete probability distribution $P(\{\pi_k\})$ then may be viewed as equivalent to the problem of finding the maximum likelihood estimator of ℓ on S_K.

One way to optimize a function subject to constraints is to use the method of Lagrange multipliers. The objective function

$$\mathcal{L} = \ell - \lambda G$$

where $G(\{\pi_k\}) = \sum_k \pi_k - 1$, gives the system of equations:

$$\frac{\partial \mathcal{L}}{\partial \pi_k} = \frac{\partial \ell}{\partial \pi_k} - \lambda \frac{\partial G}{\partial \pi_k} = 0 \qquad k = 1, \ldots, K$$

$$\frac{\partial \mathcal{L}}{\partial \lambda} = -G(\{\pi_k\}) = 0.$$

Taking partial derivatives with respect to π_k, $k = 1, \ldots, K$ and λ gives

$$\frac{\partial \ell}{\partial \pi_k} = \sum_n \frac{\partial}{\partial \pi_k} \ln P(x_n | \{\pi_k\})$$

$$= \sum_n \frac{f_k(x_n)}{P(x_n | \{\pi_k\})}.$$

The above equations become the system of algebraic equations

$$\sum_n \frac{f_k(x_n)}{\sum_{k'} \pi_{k'} f_{k'}(x_n)} = \lambda \qquad k = 1, 2, \ldots, K \qquad (2.26)$$

$$\sum_k \pi_k = 1. \qquad (2.27)$$

This problem, namely of finding the posterior estimates $\{\hat{\pi}_k\}$ with Lagrangian multipliers is well-posed. The formation of this problem hides some conditions of the simplex S_K. Namely, some of the $\hat{\pi}_k$ that act as a joint solution to (2.26) and (2.27) could be negative. In practicality this means any strategy must check the boundary conditions $\{\pi_k \geq 0 \,| k = 1, \ldots, K\}$.

To help understand the complexity of solving equations (2.26) and (2.27), consider the following CAS generated set of solutions for $K = N = 2$.

$$\pi_1 = \frac{(2 f_2(x_1) - f_1(x_1)) f_2(x_2) - f_2(x_1) f_1(x_2)}{(2 f_2(x_1) - 2 f_1(x_1)) f_2(x_2) + (2 f_1(x_1) - 2 f_2(x_1)) f_1(x_2)} \qquad (2.28)$$

$$\pi_2 = -\frac{f_1(x_1) f_2(x_2) + (f_2(x_1) - 2 f_1(x_1)) f_1(x_2)}{(2 f_2(x_1) - 2 f_1(x_1)) f_2(x_2) + (2 f_1(x_1) - 2 f_2(x_1)) f_1(x_2)} \qquad (2.29)$$

$$\lambda = 2 \qquad (2.30)$$

These solutions suppose that the values $f_k(x_n)$ are known. As the number of parameters increases, and especially as the number of samples N increases, such exact solutions quickly become intractable. Part of the difficulty is that any exact solution requires finding solutions to a polynomial of degree N in $K + 1$ variables. Clearly, other methods are required to compute a good point estimate of $\boldsymbol{\pi}^*$.

2.7   A Brief Introduction to Discrete Dynamical Systems

This section presents some definitions from the field of dynamical systems that are relevant to the dissertation. Many of these definitions can be found in an undergraduate text like [Dev89]. The

definitions in this section are mostly adapted from the review paper by Mei and Bullo [MB17]. Their paper in turn is a summarized version of the contents of a book by LaSalle [LS76].

To begin, let $\mathbb{N}$ represent the natural numbers, and $\mathbb{R}$ the real numbers. Then $\mathbb{R}^m$ is m dimensional euclidean space, and the vectors $\mathbb{1}_m$, $\mathbf{0}$ denote the vectors composed entirely of 1's and 0's respectively. The notation $\mathbf{e}_i$ $1 \le i \le m$ will denote the standard basis vectors for $\mathbb{R}^m$.

For any sequence of points $\{x_k\}_{k\in\mathbb{N}} \in \mathbb{R}^m$ use $x_k \to y$ to mean that $\|x_k - y\| \to 0$ as $k \to \infty$. For a set $S \subset \mathbb{R}^m$, let $\operatorname{Int} S$ denote the interior of S. If S is a bounded set, let ∂S denote the boundary of S, and $\overline{S} = \operatorname{Int} S \cup \partial S$ be the closure of S. The symbol $\varnothing$ will denote the empty set.

Given a map $T : \mathbb{R}^m \to \mathbb{R}^m$, for $n \in \mathbb{N}$, define $T^n = T \circ T \circ \ldots \circ T$ to be the n-fold composition of T with itself. The study of *discrete dynamical systems* is, broadly speaking, the study of such continuous maps and their compositions.

Definition 2.2 (Discrete Dynamical System). For $M \subset \mathbb{R}^m$ the map $\boldsymbol{\tau} : \mathbb{Z} \times M \to M$ describes a discrete dynamical system on M if for all $n, k \in \mathbb{Z}$ and any $x \in M$,

1. $\boldsymbol{\tau}(0, x) = x$;

2. $\boldsymbol{\tau}(n, \boldsymbol{\tau}(k, x)) = \boldsymbol{\tau}(n + k, x)$;

3. $\boldsymbol{\tau}$ is continuous.

If requirement 2 holds for only $n, k \ge 0$, then $\boldsymbol{\tau}$ describes a discrete *semi-dynamical* system on M. Thus this definition of a discrete dynamical system requires $\boldsymbol{\tau}(1, x)$ to be a continuous bijection with continuous inverse (*i.e.* a homeomorphism).

Given a continuous map $T : M \to M$ and some initial point $x_0 \in M$, one of the important goals of studying discrete dynamical systems is deciding whether sequences $\{x_n\} := T^n(x_0)$ have any limit points.

Definition 2.3 (Orbits). Sequences of the form $x_n = T^n(x_0)$ for some $x_0 \in M$ are called *orbits* of T.(also motions or trajectories).

Property 2 guarantees that for any $x \in M$, there is exactly one orbit of T such that $x_0 = x$. By abuse of notation this will be called the orbit of x.

Definition 2.4 (Limit points). Given a specific point $x \in M$ the set $\Omega(x) \subset \mathbb{R}^m$ is the set of all limit points of x. The point $y \in \mathbb{R}^m$ is a limit point of x if there is a subsequence x_{n_k} with $|n_k| \to \infty$ of the orbit $T^n(x)$ such that $x_{n_k} \to y$. This is the case for both dynamical and semi-dynamical systems.

For some set $H \subset M$, the set $\Omega(H)$ is the set of all limit points of for $x \in H$, *i.e.* $\Omega(H) = \bigcup_{x \in H} \Omega(x)$.

Definition 2.5 (Invariant Sets). The set H is called *positively invariant* if $T^n(H) \subset H \ \forall n \in \mathbb{N}$. A set H is called *negatively invariant* if $T^n(H) \supset H \ \forall n \in \mathbb{N}$. If $T(H) = H$ then H is called *invariant*.

A compact invariant set satisfies $\Omega(H) \subset H$. For discrete semi-dynamical systems, the set H needs only to be positively invariant and compact.

Property 2 is also required to guarantee uniqueness of *periodic points*.

Definition 2.6 (Periodic points, Fixed points). Periodic points are those $x \in M$ which satisfy $T^n(x) = x$ for some n. For a periodic point x, the smallest $k \in \mathbb{N}$ such that $T^k(x) = x$ is called the period of x. *Fixed points* are periodic points of period 1, *i.e.* $x \in M$ such that $T(x) = x$.

All periodic points of period k are fixed points of the map T^k. If they exist, periodic points are limit points of a discrete dynamical system.

Definition 2.7 (Stable set). For a given periodic point $p \in M$ of period k, the *stable set* of p is the set of all points $x \in M$ that eventually arrive at p, *i.e.* $W^s(T, p) := \{x \in M | T^{k+n}(x) \to p \text{ as } n \to \infty\}$. The stable set of a fixed periodic point is always non-empty as it contains p.

Definition 2.8 (Asymptotically Stable). A periodic point p is called *asymptotically stable* if $p \in \text{Int } W^s(T, p)$. In other words, p is asymptotically stable if there is some $\delta > 0$ such that $\|x - p\| < \delta$ implies that $T^{k+n}(x) \to p$, where k is the period of p.

Definition 2.9 (Lyapunov stable). A point $x \in M$ is *Lyapunov stable* if points that start sufficiently near x have orbits close to x. More precisely, x is Lyapunov stable if for any $\epsilon > 0$ there is some $\delta > 0$ such that $\|x - y\| < \delta$ implies $\|T^n(x) - T^n(y)\| < \epsilon$ for all $n \in \mathbb{N}$.

The last definition of this section is briefly covered in the review paper [MB17], but a more thorough treatment is given by LaSalle in chapter 1 section 6 of [LS76]. The definition that follows is adapted from LaSalle's work.

Definition 2.10 (Lyapunov Function). Given a discrete (semi-)dynamical system described by iterating the continuous map $T : M \to M$ Let $G \subset \mathbb{R}^m$, $G \cap M \neq \varnothing$ then a continuous map $V : G \to \mathbb{R}$ is a *Lyapunov function* for T on G if $V(T(x)) - V(x) \leq 0$ for all $x \in T(M) \cap G$.

Generally speaking, the map V is difficult to find for a given discrete dynamical system. However, as the following theorem shows, very powerful results come from finding a Lyapunov function.

Theorem 2.2 (Invariance Principle)**.**

If V is a Lyapunov function for T on G, define $E := \{x \in \overline{G} \,|\, V(T(x)) - V(x) = 0\}$ and let H denote the largest invariant set in E. Then if $T^n(x) \subset G$ is a bounded orbit of $x \in G$, there exists a number $c \in \mathbb{R}$ such that $T^n(x) \to H \cap V^{-1}(c)$.

Proof. This is theorem 3.1 of LaSalle chapter 4, section 3 [LS76]. The same idea is explored in a more general setting in chapter 4 of the same book, and is what allows passage to $M \subset \mathbb{R}^m$. An accessible, self contained version of the proof can be found in the review paper by Mei and Bullo [MB17]. $\qquad\square$

Chapter 3

DYNAMIC RESPONSIBILITY AND CLUSTER PROPORTIONS

3.1 An Iterative Algorithm

Exact solutions to the Lagrangian example 2.26 can be difficult and costly numerically. The general case of the Lagrangian requires solving a system of $K+1$ polynomials in K variables of degree at most N, with $K \cdot N$ parameters. If, for example, one uses a Gröbner basis to solve such a problem, it can take doubly exponential time in K and N. In most cases a quicker approach is preferred.

Consider instead an iterative approach by looking at the rational maps

$$r_i(\boldsymbol{\pi}) = \frac{1}{N} \sum_n \frac{\pi_i f_i(x_n)}{\sum_k \pi_k f_k(x_n)} \quad i = 1, \ldots, K$$

and defining a map

$$R : S_K \to S_K : R(\pi_1, \pi_2, \ldots, \pi_K) = (r_1(\boldsymbol{\pi}), r_2(\boldsymbol{\pi}), \ldots, r_K(\boldsymbol{\pi})). \tag{3.1}$$

An estimate of the parameters $\boldsymbol{\pi}^*$ may be obtained by calculating the fixed points of the discrete semi-dynamical system given by iteration of R. This dissertation refers to such an approach as *dynamic responsibility*. While R is differentiable on S_K this dissertation does not explore whether R is 1-1 or onto. The map $R(\boldsymbol{\pi})$ is homogeneous of degree zero as will be discussed in further detail later.

Note that this approach relies on the parameters $f_k(x_n)$, which represent the entries of a $K \times N$ matrix F. The matrix F acts as a set of parameters for the map $R(\boldsymbol{\pi})$. Writing the map as $R_F(\boldsymbol{\pi})$ will emphasize this relationship when necessary. An implementation of this map can be found in appendix A, section A.1.

Dynamic responsibility (DR) describes an alternative algorithm (see algorithm 1) to the Lagrange multipliers method shown above. It is based on the idea of iterating the map $R(\boldsymbol{\pi})$. This is likely to be a good idea because $R(\boldsymbol{\pi})$ is partially derived from Bayes' rule, which means each iteration could

---

Algorithm 1 Dynamic Responsibility Algorithm

Require: F a $K \times N$ matrix
Require: $\boldsymbol{\pi}_0$, ϵ
  procedure ITERATION$(F, \boldsymbol{\pi}_0, \epsilon)$ $\triangleright$ ϵ serves as a stopping tolerance
      $n \leftarrow 1$
      $\boldsymbol{\pi}_n \leftarrow R(\boldsymbol{\pi}_0)$
      $orbit \leftarrow \boldsymbol{\pi}_0, \boldsymbol{\pi}_1$
      while $|\boldsymbol{\pi}_n - \boldsymbol{\pi}_{n-1}| > \epsilon|\boldsymbol{\pi}_n|$ **do**
         $\boldsymbol{\pi}_{n+1} \leftarrow R(\boldsymbol{\pi}_n)$
         $orbit \leftarrow \boldsymbol{\pi}_0, \ldots, \boldsymbol{\pi}_{n+1}$
         $n \leftarrow n + 1$
      end while
      return $orbit$ $\triangleright$ at this point $\boldsymbol{\pi}_{n-1}$ is approximately $\hat{\boldsymbol{\pi}}$
  end procedure

Table 3.1: The main algorithm: Iteration of the $R_F(\boldsymbol{\pi})$ map. Note that the entire orbit is kept for gradient descent purposes.

be thought of as updating the probabilities defined by $\boldsymbol{\pi} \in S_K$.

While it is not known *a priori* that algorithm 1 is correct, the main theorem of this chapter, theorem 3.8, shows that dynamic responsibility converges to a unique solution under reasonable conditions. Section 3.5 discusses the correct hypotheses under which the fixed point $\hat{\boldsymbol{\pi}}$ satisfying $R(\hat{\boldsymbol{\pi}}) = \hat{\boldsymbol{\pi}}$ is a maximum likelihood estimator. A version of this algorithm is implemented in appendix A, section A.2.

3.2 Further Examination of the $K = 2$ Case for Arbitrary N

For now we will limit our discussion to the case $K = 2$. The advantage in this is that $R(\boldsymbol{\pi})$ is a homogenous map so we may consider the ratios of the two coordinates as a map on the projective line. The advantage is that this reduces the number of dimensions in consideration to 1.

To be precise, start with the $2 \times N$ matrix

$$F = \begin{pmatrix} a_1 & a_2 & a_3 & \ldots & a_N \\ b_1 & b_2 & b_3 & \ldots & b_N \end{pmatrix} \tag{3.2}$$

We should think of this F as being defined by taking N i.i.d. samples from the distribution $X_i \sim P(x) = \pi_1 f_1(x) + \pi_2 f_2(x)$ and define $a_i = f_1(x_i)$, $b_i = f_2(x_i)$. In particular, we have that $a_i \geq 0$

and $b_i \geq 0$.

In this context we have the map

$$R_F(\pi_1, \pi_2) = \frac{1}{N} \left(\sum_{i \leq N} \frac{\pi_1 a_i}{\pi_1 a_i + \pi_2 b_i}, \sum_{i \leq N} \frac{\pi_2 b_i}{\pi_1 a_i + \pi_2 b_i} \right) \tag{3.3}$$

It is clear from the definition that for any $\lambda \in \mathbb{R}$, we have $R(\lambda \pi_1, \lambda \pi_2) = R(\pi_1, \pi_2)$, and so R is homogeneous of degree zero as stated above. If we then define $t = \frac{\pi_1}{\pi_2}$ and

$$T(t) = \frac{\sum_{i \leq N} \frac{t a_i}{t a_i + b_i}}{\sum_{i \leq N} \frac{b_i}{t a_i + b_i}} \tag{3.4}$$

then $T(t) : \mathbb{P}^1_{\mathbb{R}} \to \mathbb{P}^1_{\mathbb{R}}$ is a map from the projective line to itself. If we write $R(\pi_1, \pi_2) = \frac{1}{N}(r_1(\pi_1, \pi_2), r_2(\pi_1, \pi_2))$, then the equation

$$T\left(\frac{\pi_1}{\pi_2}\right) = \frac{r_1(\pi_1, \pi_2)}{r_2(\pi_1, \pi_2)} \tag{3.5}$$

describes the relationship between $T(t)$ and $R(\pi_1, \pi_2)$.

The following theorem gives another characterization of the map $T(t)$.

Theorem 3.1. The function $f(t) = \prod_{i=1}^{N} t a_i + b_i$, is a solution to the differential equation

$$T(t) = \frac{t f'(t)}{N f(t) - t f'(t)}$$

Proof. We will show that $f(t)$ satisfies the given ODE. Let the numerator and denominator of $T(t)$ be $t A(t)$ and $B(t)$ respectively. Then note from equation (3.4) that we have

$$A(t) = \sum_{i=1}^{N} \frac{a_i}{t a_i + b_i},$$

$$B(t) = \sum_{i=1}^{N} \frac{b_i}{t a_i + b_i},$$

and

$$T(t) = \frac{t A(t)}{B(t)}.$$

We now calculate as follows:

$$T(t) = \frac{tf'(t)}{Nf(t) - tf'(t)}$$

$$\frac{NT(t)}{t(1 + T(t))} = \frac{f'(t)}{f(t)}$$

$$\int \frac{NT(t)}{t(1 + T(t))} \, dt = \log(f(t)) \tag{3.6}$$

Since $tA(t) + B(t) = N$, we have

$$1 + T(t) = \frac{1}{B(t)}(B(t) + tA(t)) = \frac{N}{B}$$

and so

$$\frac{NT(t)}{t(1 + T(t))} = t^{-1}B(t)T(t) = A(t)$$

substituting the expression $A(t) = \sum_{i=1}^{N} \frac{a_i}{ta_i + b_i}$ into

$$\int A(t) \, dt = \log(f(t))$$

and integrating gives the desired result. $\qquad\square$

At this point it is important to see that with $f(t)$ defined as in theorem 3.1

$$\pi_2^N \cdot f(t) = \prod_{i=1}^{N}(\pi_1 a_i + \pi_2 b_i) \tag{3.7}$$

and the RHS above is the evaluation of the sample x_1, x_2, ... , x_N on the joint distribution, with i.i.d. $X_i \sim P(x) := \pi_1 f_1(x) + \pi_2 f_2(x)$. In connection with this observation we have the following corollary.

Corollary 3.2. *If $L_F(\pi_1, \pi_2) = \prod_{i=1}^{N}(\pi_1 a_i + \pi_2 b_i)$ and $\mu_i = \log(\pi_i)$ for $i = 1, 2$, then $R_F(\pi_1, \pi_2)$ as described in equation (3.3) satisfies*

$$R_F(\pi_1, \pi_2) = \frac{1}{N}\nabla_{\boldsymbol{\mu}} \log(L_F(\pi_1, \pi_2)) \tag{3.8}$$

Proof. We begin by observing that equation (3.6) relates the log derivative of $f(t)$ to $T(t)$. Given the relationship between $T(t)$ and R_F as described in equation (3.5), the result is unsurprising.

We continue by direct derivation. If $\ell_F = \log(L_F)$ then

$$
\begin{aligned}
\frac{\partial \ell_F}{\partial \mu_1} &= \sum_{i=1}^{N} \frac{\partial \log(\pi_1 a_i + \pi_2 b_i)}{\partial \mu_1} \\
&= \sum_{i=1}^{N} \frac{a_i}{\pi_1 a_i + \pi_2 b_i} \frac{\partial \pi_1}{\partial \mu_1} \\
&= \sum_{i=1}^{N} \frac{\pi_1 a_i}{\pi_1 a_i + \pi_2 b_i} = r_1(\pi_1, \pi_2).
\end{aligned}
\tag{3.9}
$$

A similar computation gives

$$
\frac{\partial \ell_F}{\partial \mu_2} = r_2(\pi_1, \pi_2)
$$

Thus we have

$$
R_F(\pi_1, \pi_2) = \frac{1}{N}(r_1(\pi_1, \pi_2), r_2(\pi_1, \pi_2)) = \frac{1}{N}\left(\frac{\partial \ell_F}{\partial \mu_1}, \frac{\partial \ell_F}{\partial \mu_2}\right)
\tag{3.10}
$$

as required. ◻

One consequence of corollary 3.2 is that fixed points of R_F are constrained maxima of ℓ_F on S_2. This dissertation explores convergence further in theorem 3.5, but the relationship between fixed points and constrained maxima shows that iteration of R_F as in algorithm 1 will converge in some cases.

3.3 Some Basic Examples of the $K = 2$ Case

This section begins by exploring homogeneity of the functions $L_F(\pi_1, \pi_2) = \prod_{i=1}^{N}(\pi_1 a_i + \pi_2 b_i)$ and $R_F(\pi_1, \pi_2) = \frac{1}{N}\left(\pi_1 \frac{\partial \log(L_F)}{\partial \pi_1}, \pi_2 \frac{\partial \log(L_F)}{\partial \pi_2}\right)$. It has already been remarked near equation (3.3) that $R_F(\pi_1, \pi_2)$ is homogeneous of degree 0 in its arguments. Similarly, $L_F(\pi_1, \pi_2)$ is homogeneous of degree N in its arguments *i.e.* $L_F(\lambda \pi_1, \lambda \pi_2) = \lambda^N L_F(\pi_1, \pi_2)$.

It is also true that both of these functions are homogeneous in their parameters. In fact, take the matrix F as in equation 3.2 and set

$$
G = \begin{pmatrix} \lambda_1 a_1 & \lambda_2 a_2 & \lambda_3 a_3 & \dots & \lambda_N a_N \\ \lambda_1 b_1 & \lambda_2 b_2 & \lambda_3 b_3 & \dots & \lambda_N b_N \end{pmatrix}
\tag{3.11}
$$

then a quick calculation shows that $R_F(\pi_1, \pi_2) = R_G(\pi_1, \pi_2)$. In other words, R_F is homogeneous of degree zero in the columns of F. However this is not the case for L_F, for it is the case that $L_G(\pi_1, \pi_2) = \left(\prod_{i=1}^{N} \lambda_i\right) L_F(\pi_1, \pi_2)$, *i.e.* L_F is homogeneous of degree 1 in each column of F. These

facts also imply that $L_{\lambda F} = \lambda^N L_F$ and $R_{\lambda F} = R_F$.

I will use the observation of homogeneity to generalize some of my calculations. For example, calculations might assume that $a_i + b_i = 1 \; \forall i = 1 \ldots N$ by dividing each column of an original matrix by its sum. This ability will be important in chapter 4. Many of the following examples use projective coordinates for the columns, e.g. $a_i = 1 \; \forall i = 1 \ldots N$. A more practical way to do this with a matrix of real data would be to divide each column by any of its entries, for example dividing each column by the maximum entry. This is what the given examples do; exactly one entry in each column will have the value 1.

3.3.1 An Example for a simple family of matrices F_α

Example 3.1. As a first example consider the matrix $F = \left(\begin{smallmatrix} 1 & \alpha \\ 2 & 1 \end{smallmatrix}\right)$ for varying $\alpha > 0$. Figure 3.1 gives a graph of the fixed points $\hat{\boldsymbol{\pi}}_\alpha$ for varying α. Note that as might be expected, $\hat{\pi}_1$ increases with α. However, this increase is not strict or smooth. In fact, it is constant until $\alpha = 1.5$, after which it grows approximately like $1 - e^{-\alpha}$. Since $\hat{\pi}_1 + \hat{\pi}_2 = 1$, similar but opposing statements hold for $\hat{\pi}_2$.

A bit of algebra makes this example more precise. From figure 3.1d, note that the function

$$\ell_\alpha(\pi_1, 1 - \pi_1) = \frac{1}{2}\left[\ln(2 - \pi_1) + \ln(1 + (\alpha - 1)\pi_1)\right] \tag{3.12}$$

always goes through the point $\left(0, \frac{1}{2}\ln(2)\right)$. For some of the curves, this point is the maximum and for others it is not. The place where it changes is again at $\alpha = 1.5$. To see why this is, consider the partial derivative

$$\left.\frac{\partial \ell_\alpha}{\partial \pi_1}\right|_{\pi_1 = 0} = \frac{1}{2}\left(-\frac{1}{2} + \alpha - 1\right).$$

This is clearly zero exactly when $\alpha = \frac{3}{2}$.

I leave this example with one final remark. Solving $\frac{\partial \ell_\alpha}{\partial \pi_1} = 0$ for π_1 gives

$$\hat{\pi}_1 = 1 - \frac{1}{2(\alpha - 1)}.$$

This shows that, when $\alpha \geq \frac{3}{2}$, $\hat{\pi}_1 = 1 - e^{-\ln(2(\alpha-1))}$, giving $\hat{\pi}_2 = e^{-\ln(2(\alpha-1))}$ as mentioned earlier.

It is worth noting that in example 3.1 that increasing N gives a smaller effect to a change in the parameter α. In one example, consider appending to F_α the matrix $A = \left(\begin{smallmatrix} 1 & 2 \\ 2 & 1 \end{smallmatrix}\right)$ some finite number of times. Taking $N = 12$ by appending A three times to F_α changed the maximum distance separating
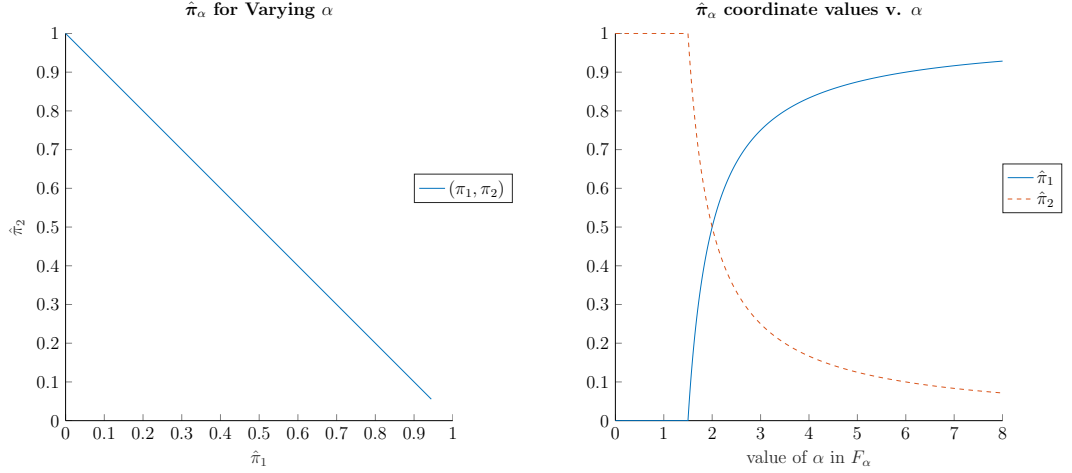
(a) A plot of stablepoints on S_2 for different F_α. Here $N = 2$

(b) A plot of point coordinates on S_2 for different F_α.
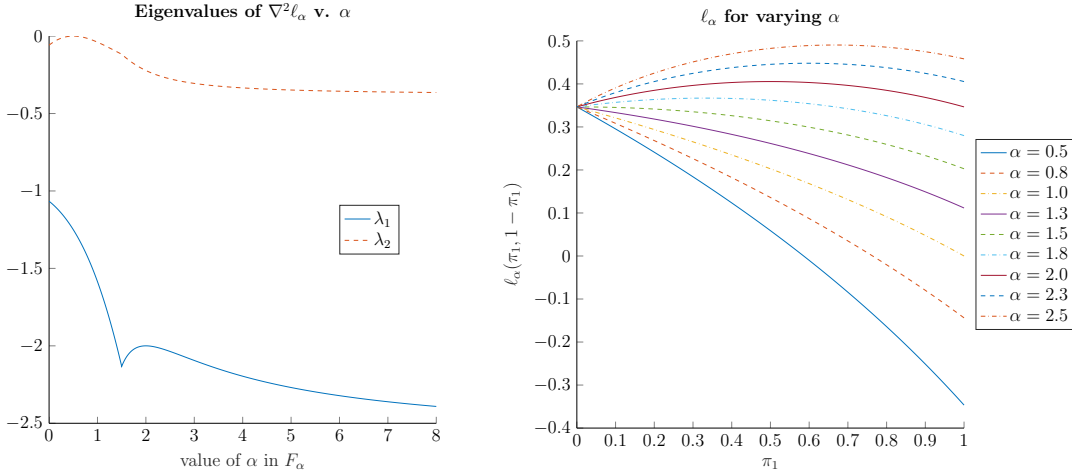
(c) A plot of eigenvalues for $\nabla^2 \ell_\alpha(\hat{\boldsymbol{\pi}}_\alpha)$ on S_2 for different F_α.

(d) A plot of $\ell_\alpha(\pi_1, 1 - \pi_1)$ on S_2 for different F_α.

Figure 3.1: A plot of fixed points for F_α as described in example 3.1. Plot a represents points on S_2 which are stable point for a different F_α. Each point on the curves in plot b represents a coordinate for the stable point of a different F_α. The x-axis represents α.

π_1 and π_2 to less than .9 as seen in figure 3.2. This corresponds with the intuition that as N increases the importance of a single sample decreases.

3.3.2 A GMM Example with an Unknown Mean

For this example, begin by considering the Gaussian mixture model defined by the distribution

$$\boldsymbol{X} \sim .8\mathcal{N}(0, 3) + .2\mathcal{N}(2.5, .04) \tag{3.13}$$

We may generate samples $\{X_1, X_2, \ldots, X_N\}$ of the random variable $\boldsymbol{X}$ via the process described

$\hat{\pi}_\alpha$ **coordinate values v.** $\log_2(\alpha)$
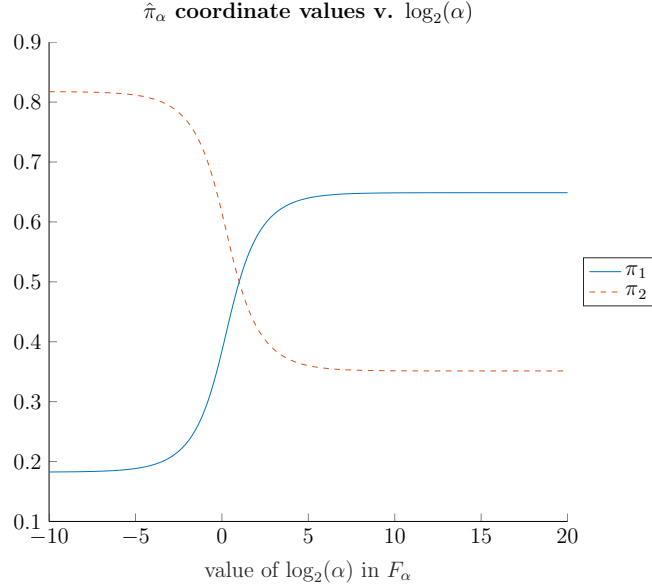
Figure 3.2: Coordinate plots of $\hat{\boldsymbol{\pi}}_\alpha$ for varying α. In this case, $N = 12$, so a single entry of F_α has a smaller effect.

in experiment 2.1. Figure 3.3 shows a histogram of $N = 10^5$ samples of $\boldsymbol{X}$, along with a graph of the pdf from 3.13.

Example 3.2. For a sample $\boldsymbol{X} = \{x_n\}_{n=1}^N$, let F_α be the $2 \times N$ matrix given by $(F_\alpha)_{ij} = f_i(x_j, \alpha)$ $i = 1, 2$ $j = 1, \ldots, N$ where

$$f_1(x, \alpha) = \frac{1}{\sqrt{6\pi}} \exp\left(-\frac{x^2}{3}\right) \tag{3.14}$$

and

$$f_2(x, \alpha) = \sqrt{\frac{5}{2\pi}} \exp\left(-5(x - \alpha)^2\right) \tag{3.15}$$

Thus the 'unknown mean' part of the example refers to the fact that equation 3.15 depends on α as the mean, μ_2, for the second cluster. Note that $f_1(x, \alpha)$ in equation 3.14 does not actually depend on α.

In figure 3.4 we see the behavior of $\hat{\boldsymbol{\pi}}_\alpha$ for several different sample sizes. We note that the non differentiable behavior of $\hat{\boldsymbol{\pi}}_\alpha$ is still captured in these examples. As expected, the curves for the coordinates settles down as N increases. It is also important to point out that the maximum occurs when $\alpha = \mu_2 = 2.5$. Notably, $\hat{\boldsymbol{\pi}}_{2.5}$ approximates π^*, which is the ratio of the given sample clusters.

Important takeaways from examples 3.1 and 3.2 are the behavior as N increases and the non-smooth
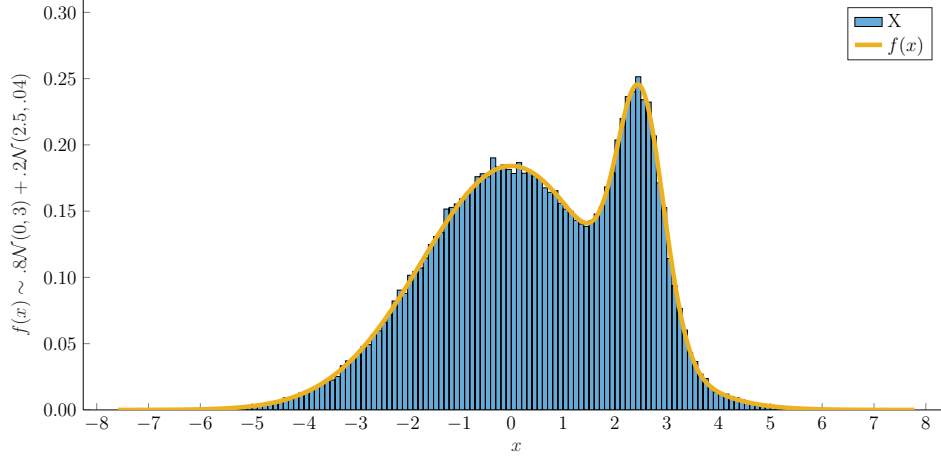
Figure 3.3: Histogram and p.d.f. of a 1 dimensional GMM with $K = 2$ clusters
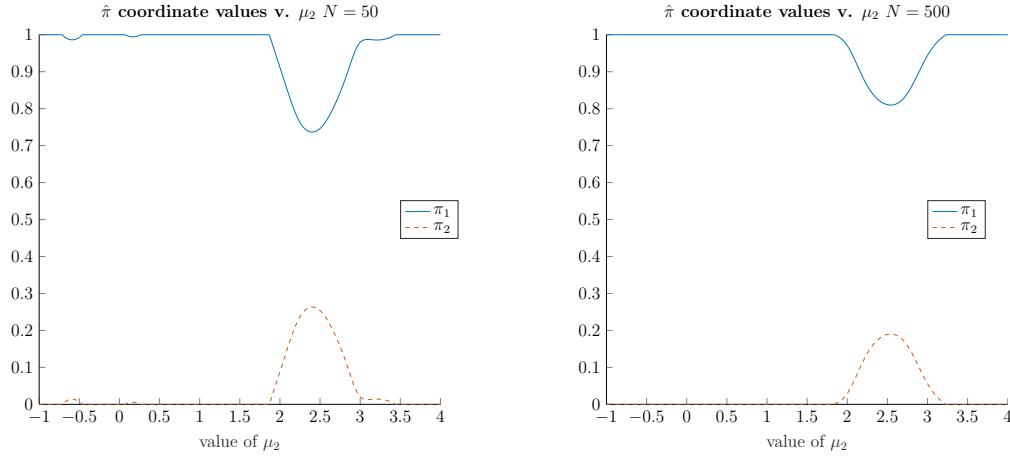
(a) Here $N = 50$. Ratios of the samples of the clusters are $\pi_1^* = 0.8$ and $\pi_2^* = 0.2$.

(b) Here $N = 500$. Ratios of the samples of the clusters are $\pi_1^* = 0.8$ and $\pi_2^* = 0.2$.

Figure 3.4: These graphs display coordinates of the fixed point $\hat{\boldsymbol{\pi}}_\alpha$ for varying F_α. The rows of F_α are given by evaluating equations 3.14 and 3.15 on a sample $\boldsymbol{X}$ of varying sizes. The parameter α represents a guess for μ_2. The graphs shown here are for sample sizes $N = 50$ and $N = 500$.

behaviors of the fixed point as parameters change. Later sections discuss each of these. Section 3.5, discusses the relationship of fixed points with the maximum likelihood estimate for $\hat{\boldsymbol{\pi}}$. Example 3.9 and theorem 3.9 discuss one reason that this fixed point process might be non-smooth, and chapter 4 addresses further difficulties.

3.4   Convergence of R_F for Arbitrary K

For the rest of this section let $F = (f_{ij})$ be a $K \times N$ matrix with positive entries. For $\boldsymbol{\pi} \in \mathbb{R}^K$, let $L_F(\boldsymbol{\pi}) = \prod_{j=1}^{N} \left(\sum_{i=1}^{K} f_{ij} \pi_i \right)$. If additionally, $\boldsymbol{\pi} \in \mathbb{R}_+^K := \{ \boldsymbol{x} \in \mathbb{R}^K | x_i \geq 0 \, \forall i \}$ is in the positive orthant,

let $\ell_F(\boldsymbol{\pi}) = \dfrac{1}{N}\log(L(\boldsymbol{\pi}))$. Occasionally, where the use of F is clear from context, the notation will omit it. Theorem 3.1 implies the corollary below.

Corollary 3.3. *The map $R_F(\boldsymbol{\pi})$ as defined in equation (3.1) satisfies*

$$R_F(\boldsymbol{\pi}) = \left(\pi_i \cdot \left.\frac{\partial \ell_F}{\partial \pi_i}\right|_{\boldsymbol{\pi}}\right)_{1 \le i \le K}$$

If $\mu_i = \log \pi_i$, $i = 1, \ldots, K$ then this also means that

$$R_F(\boldsymbol{\pi}) = \left(\left.\frac{\partial \ell_F}{\partial \mu_i}\right|_{\boldsymbol{\pi}=e^{\boldsymbol{\mu}}}\right)_{1 \le i \le K} = \nabla_{\boldsymbol{\mu}} \ell_F(e^{\boldsymbol{\mu}}) \tag{3.16}$$

In short, R_F is the gradient of ℓ_F with respect to the coordinates μ_i, $i = 1, \ldots, K$.

Proof. This follows from the fact that

$$\frac{\partial \ell}{\partial \pi_i} = \frac{1}{N}\sum_{j=1}^{N} \frac{f_{ij}}{\sum_{k=1}^{K} \pi_k \cdot f_{kj}}.$$

Defining F by $F_i^j = f_{ij} = f_i(x^{(j)})$ for $1 \le i \le K$ different p.d.f.s f_i, then this definition corresponds with the definition in equation (3.1).

Now given that $\mu_i = \log \pi_i$ then $\pi_i = e^{\mu_i}$ and $\frac{\partial \boldsymbol{\pi}}{\partial \boldsymbol{\mu}} = \operatorname{diag}(\boldsymbol{\pi})$ so long as $\pi_i \neq 0$ (*i.e.* $\boldsymbol{\pi} \in S_k^{\circ}$ the interior of S_K). Here $\operatorname{diag}(\boldsymbol{v})$ is the diagonal matrix with entries given by $\boldsymbol{v}$. This implies that

$$\frac{\partial}{\partial \boldsymbol{\mu}} = \frac{\partial \boldsymbol{\pi}}{\partial \boldsymbol{\mu}} \cdot \frac{\partial}{\partial \boldsymbol{\pi}} = \operatorname{diag}(\boldsymbol{\pi}) \cdot \frac{\partial}{\partial \boldsymbol{\pi}}.$$

In other words $\nabla_{\boldsymbol{\mu}} \ell_F(e^{\boldsymbol{\mu}}) = \operatorname{diag}(\boldsymbol{\pi}) \cdot \nabla_{\boldsymbol{\pi}} \ell_F(\boldsymbol{\pi}) = R_F(\boldsymbol{\pi})$. $\qquad\square$

Corollary 3.3 demonstrates a very close relationship between the maps $R_F(\boldsymbol{\pi}) : S_K \to S_K$ and $\ell_F(\boldsymbol{\pi}) : \mathbb{R}^K \to \mathbb{R}$. This is an important part of the proof of theorem 3.8, which shows algorithm 1 converges.

Also, the beginning of section 3.3 discussed the homogeneity of L_F and R_F. Corollary 3.3, sheds light on the relationship of these two functions. Namely, the homogeneity of L_F induces this same property in R_F, as R_F is essentially the gradient of $\log(L_F(\boldsymbol{\pi}))$.

Note here that the function $\ell_F(\boldsymbol{\pi}) = \dfrac{1}{N}\ln(L_F(\boldsymbol{\pi}))$ is not homogeneous, but satisfies the equation

$$\ell_F(\lambda \boldsymbol{\pi}) = \ell_F(\boldsymbol{\pi}) + \ln(\lambda). \tag{3.17}$$

This means that over $\mathbb{R}_+^K$, ℓ_F has no maximum. However, it will still have a maximum on S_K as the simplex is compact. A similar statement about homogeneity is also true for the columns of F. If $G = (\lambda_n F_n)$ then $\ell_G(\boldsymbol{\pi}) = \ell_F(\boldsymbol{\pi}) + \frac{1}{N} \sum_n \log(\lambda_n)$. As $\sum_n \log(\lambda_n)$ has no dependence on $\boldsymbol{\pi}$, this does not change the critical points of $\ell_F(\boldsymbol{\pi})$.

To show that algorithm 1 converges, first note that for fixed F, $-\ell(\boldsymbol{\pi})$ is a convex function on $\mathbb{R}^K$. This is obvious as $\ell(\boldsymbol{\pi})$ is the sum of logs of linear functions, though $-\ell(\boldsymbol{\pi})$ may be not strictly convex. Convexity is important here as if $-\ell(\boldsymbol{\pi})$ is strictly convex, then as inferred by corollary 3.2, iteration of $R(\boldsymbol{\pi})$ converges to a unique fixed point as shown in theorem 3.8. Lemma 3.4 explains when $-\ell(\boldsymbol{\pi})$ is *strictly* convex on S_K.

Lemma 3.4. *If $F = (f_{ij})$ has full rank, then $-\ell(\boldsymbol{\pi})$ is strictly convex.*

Proof. To begin, let F_i be the i-th column of F. In this notation $(F')_j$ would be the j-th row of F. F_i' is the transpose of the i-th column. To condense notation, we note that $\sum_{k=1}^K \pi_k \cdot f_{kj} = \langle F_j, \boldsymbol{\pi} \rangle$ With this notation, we calculate the Hessian of $\ell(\boldsymbol{\pi})$.

$$\frac{\partial^2 \ell}{\partial \pi_j \partial \pi_i} = \frac{\partial}{\partial \pi_j} \frac{1}{N} \sum_{n=1}^N \frac{f_{in}}{\sum_{k=1}^K \pi_k \cdot f_{kn}} \tag{3.18}$$

$$= -\frac{1}{N} \sum_{n=1}^N \frac{f_{in} f_{jn}}{\langle F_n, \boldsymbol{\pi} \rangle^2} \tag{3.19}$$

We note here that $\sum_{n=1}^N f_{in} f_{jn} = \langle (F')_i, (F')_j \rangle$ is the inner product of the i-th and j-th rows of F. Let $G(\boldsymbol{\pi}) = \left[\frac{F_1}{\langle F_1, \boldsymbol{\pi} \rangle}, \ldots, \frac{F_N}{\langle F_N, \boldsymbol{\pi} \rangle} \right]$. Then we have

$$\frac{\partial^2 \ell}{\partial \pi_j \partial \pi_i} = -\frac{1}{N} \langle (G(\boldsymbol{\pi})')_i, (G(\boldsymbol{\pi})')_j \rangle \tag{3.20}$$

so if $\nabla^2 \ell(\boldsymbol{\pi})$ is the Hessian of $\ell(\boldsymbol{\pi})$, we have

$$\nabla^2 \ell(\boldsymbol{\pi}) = -\frac{1}{N} G(\boldsymbol{\pi}) \cdot G(\boldsymbol{\pi})' \tag{3.21}$$

Since $G(\boldsymbol{\pi}) \cdot G(\boldsymbol{\pi})'$ is positive definite iff $G(\boldsymbol{\pi})'$ has linearly independent columns, we have the theorem. $\qquad \square$

Lemma 3.4 provides conditions under which $\ell_F(\boldsymbol{\pi})$ has a unique maximum on S_K. Note here that

$\ell_F(\boldsymbol{\pi})$ will achieve a maximum on S_K as it cannot have an asymptote unless $\boldsymbol{\pi} = \mathbf{0}$ or there is some column F_n of F such that $F_n = \mathbf{0}$. This is ruled out by hypotheses, so $\ell_F(\boldsymbol{\pi})$ is continuous, and will have a unique maximum on S_K. While the hypothesis that $f_{ij} > 0 \; \forall i \le K, j \le N$ may be changed slightly to require $\langle F_n, \boldsymbol{\pi} \rangle > 0 \; \forall n \le N \; \boldsymbol{\pi} \in S_K$; in practice the original hypothesis is easier to ensure.

In light of corollary 3.3, the function $g(\boldsymbol{\pi}) = \boldsymbol{\pi} - R_F(\boldsymbol{\pi})$ has a zero precisely when $\ell_F(\boldsymbol{\pi})$ has a maximum on S_K. This fact also seems to imply that iteration of $R_F(\boldsymbol{\pi})$ has a unique fixed point when $\ell_F(\boldsymbol{\pi})$ has a unique maximum on S_K. Theorem 3.5 addresses this idea, but the whole picture is slightly more complicated.

Theorem 3.5. If F has full rank, and $\hat{\boldsymbol{\pi}} \in \operatorname{Int} S_K$ maximizes $\ell_F(\boldsymbol{\pi})$ then the only fixed point for iteration of $R_F(\boldsymbol{\pi})$ on the interior of S_K is $\hat{\boldsymbol{\pi}}$.

Proof. Because the theorem discusses maximizing $\ell_F(\boldsymbol{\pi})$ subject to the linear constraint $\langle \boldsymbol{\pi}, \mathbb{1}_K \rangle = 1$, this theorem is strongly tied to corollary 1.2.1 of Nesterov [Nes03, chapter 1, p. 18]. The proof proceeds similarly.

Let $\mathbb{1}_K = (1, 1, \dots, 1) \in \mathbb{R}^K$ be the vector of all ones. If F has full rank, and $\hat{\boldsymbol{\pi}} = \arg\max_{\boldsymbol{\pi} \in S_K} \ell(\boldsymbol{\pi})$, we claim that $\hat{\boldsymbol{\pi}} = R(\hat{\boldsymbol{\pi}})$. Note here that if $\nabla \ell(\boldsymbol{\pi}) = \mathbb{1}_K$ for some $\boldsymbol{\pi} \in S_K$, corollary 3.3 gives that $\boldsymbol{\pi}$ satisfies $R(\boldsymbol{\pi}) = \boldsymbol{\pi}$. Thus it is sufficient to show that $\nabla \ell_F(\hat{\boldsymbol{\pi}}) = \mathbb{1}_K$

Now, for any $\boldsymbol{\pi} \in S_K$ we have

$$\left(\frac{\partial \ell}{\partial \pi_i} \right)_{\boldsymbol{\pi} \in S_K} = \left(\frac{\partial \ell}{\partial \pi_i} \right)_{\pi_K = 1 - \sum_{j<K} \pi_j}$$
$$= \frac{\partial \ell}{\partial \pi_i} + \frac{\partial \ell}{\partial \pi_K} \frac{\partial \pi_K}{\partial \pi_i} = \frac{\partial \ell}{\partial \pi_i} - \frac{\partial \ell}{\partial \pi_K} \tag{3.22}$$

Thus if $(\nabla \ell(\hat{\boldsymbol{\pi}}))_{\boldsymbol{\pi} \in S_K} = 0$, we have $\frac{\partial \ell}{\partial \pi_i} = \frac{\partial \ell}{\partial \pi_K} \; \forall i < K$. In other words $\nabla \ell(\hat{\boldsymbol{\pi}}) = \lambda \mathbb{1}_K$ for some $\lambda \ge 0$. Since $f_{ij} \ge 0 \; \forall j$, we have

$$\frac{\partial \ell}{\partial \pi_i} = \frac{1}{N} \sum_{j=1}^N \frac{f_{ij}}{\sum_{k=1}^K \pi_k f_{kj}} \ge 0. \tag{3.23}$$

We have equality here iff $f_{ij} = 0 \; \forall j$, but this cannot happen if F has full rank. Therefore $\lambda = \frac{\partial \ell}{\partial \pi_i} > 0$.

Then by corollary 3.3 we get

$$1 = \langle R(\boldsymbol{\pi}), \mathbb{1}_K \rangle = \langle \nabla \ell(\boldsymbol{\pi}), \boldsymbol{\pi} \rangle \qquad \forall \boldsymbol{\pi} \in S_K \tag{3.24}$$

but $\langle \nabla \ell(\hat{\boldsymbol{\pi}}), \hat{\boldsymbol{\pi}} \rangle = \lambda \langle \mathbb{1}_K, \hat{\boldsymbol{\pi}} \rangle = \lambda$. Thus $\lambda = 1$ and $R(\hat{\boldsymbol{\pi}}) = \hat{\boldsymbol{\pi}}$.

Since F has full rank by assumption, $\ell(\boldsymbol{\pi})$ is strictly convex, and therefore it has a unique maximizing point $\hat{\boldsymbol{\pi}} \in S_K$. As discussed above, this maximizer is a fixed point of the map $R(\boldsymbol{\pi})$. Since all fixed points of $R_F(\boldsymbol{\pi})$ on the interior of S_K must also be critical points of $\ell_F(\boldsymbol{\pi})$, there can be only one fixed point of $R_F(\boldsymbol{\pi})$ on Int S_K.

$\square$

Remark 3.3. It is important to note here that $R_F(\boldsymbol{\pi})$ behaves very differently on ∂S_K than it does on the interior. In fact, if $\pi_{i_m} = 0$ for some set of indicies $I = \{i_1, \ldots, i_m\}$, $m < K$, then those indicies will stay zero on every iteration of the map $R_F(\boldsymbol{\pi})$. On the other hand, if $\boldsymbol{\pi} \in$ Int S_K then $r_i(\boldsymbol{\pi}) > 0$ for $1 \leq i \leq K$ (given the hypothesis $f_{ij} > 0$). Thus if $\boldsymbol{\pi} \in$ Int S_K then $R_F(\boldsymbol{\pi}) \in$ Int S_K. Conversely, when $\boldsymbol{\pi} \in \partial S_K$, $R_F(\boldsymbol{\pi}) \in \partial S_K$. This implies that S_K has a partition by sets which are positively invariant under iteration by R_F, and motivates the following definition.

Definition 3.4 (Facet). A subset $U \subset S_K$ will be called a *facet* of S_K if there is a distinct set of indices $I_U = \{i_1, \ldots, i_m\}$, such that $\forall \boldsymbol{u} \in U$, $u_j = 0$ if $j \in I_U$ and $u_j > 0$ if $j \notin I_U$. For each increasing set of indices $I = \{i_1 < i_2 < \ldots < i_m\}$ $m < K$, there will be a unique facet $U \subset S_K$ such that $I_U = I$. Denote this facet by S_K^I. If $I = \varnothing$, define $S_K^I = \text{Int}(S_K)$. Because there are 2^K ways to choose such an index set I, there are 2^K facets of S_K.

Because facets of S_K are positively invariant, there will not necessarily be a single fixed point for iterating $R_F(\boldsymbol{\pi})$, but potentially one fixed point for each of the 2^K facets of S_K. Because $\partial S_K = \bigcup_{I \neq \varnothing} S_K^I$, at most one of the fixed points will be in $\text{Int}(S_K)$. Occasionally, some of the fixed points may coincide, but this will be addressed in remarks after theorem 3.8. Thus R_F has many fixed points, but only one fixed point will maximize ℓ_F.

Definition 3.5 (Critical Fixed Point,Maximizing Fixed Point). Call $\boldsymbol{\pi}_0 \in S_K$ a *critical fixed point* of (ℓ_F, R_F) if $\boldsymbol{\pi}_0 = R_F(\boldsymbol{\pi}_0)$ and $\boldsymbol{\pi}_0$ is a critical point of ℓ_F *i.e.* $\nabla \ell_F(\boldsymbol{\pi}_0)|_{S_K} = 0$. Call $\boldsymbol{\pi}_0$ a *maximizing fixed point*, if in addition to being a critical fixed point, $\max_{\boldsymbol{\pi} \in S_K} \ell_F(\boldsymbol{\pi}) = \ell_F(\boldsymbol{\pi}_0)$.

Facets each behave differently under iteration by R_F, therefore it is useful to have notation that allows discussion of each facet to proceed independently of the other facets. This is provided in part by the following definitions.

Definition 3.6 (Deletion map)**.** For a fixed k and for each $1 \leq i \leq k$ define the i-th *deletion map* $\phi_i^k : \mathbb{R}^k \to \mathbb{R}^{k-1}$ by

$$\phi_i^k(x_1, x_2, \ldots, x_k) = (x_1, \ldots, \hat{x}_i, \ldots, x_k),$$

where the notation $\hat{x}_i$ denotes deletion of the i-th coordinate.

Definition 3.7 (Insertion map)**.** For a fixed k and for each $1 \leq i \leq k$ define the i-th *insertion map* $\psi_i^{k-1} : \mathbb{R}^{k-1} \to \mathbb{R}^k$ by

$$\psi_i^{k-1}(x_1, x_2, \ldots, x_{k-1}) = (x_1, \ldots, x_{i-1}, 0, x_i, \ldots, x_{k-1}).$$

So ψ_i^{k-1} inserts zero into the i-th coordinate. Note that the image of the i-th deletion map is the plane perpendicular to the i-th standard basis vector $\boldsymbol{e}_i$ *i.e.* $\mathrm{Im}(\psi_i^{k-1}) = \{x \in \mathbb{R}^k | x_i = 0\}$

The insertion and deletion maps satisfy the following relations

$$\phi_i^k \circ \psi_i^{k-1} = \mathrm{id}_{\mathbb{R}^{k-1}} \tag{3.25}$$

$$\psi_j^{k-1} \circ \phi_j^k = P_{\mathrm{Im}(\psi_j^{k-1})} \tag{3.26}$$

where $\mathrm{id}_{\mathbb{R}^{k-1}}$ is the identity map on $\mathbb{R}^{k-1}$, and $P_{\mathrm{Im}(\psi_j^{k-1})}$ is projection onto the image of ψ_j^{k-1}. The notations id and P_ψ will be used where the use is clear in context.

The insertion and deletion maps relate to the facets via composition of the maps. In particular, let $I = \{i_1 < i_2 < \ldots < i_m\}$ be the index set for a facet $S_K^I \subset S_K$. Then define

$$\Psi_I = \psi_{i_1}^{K-1} \circ \psi_{i_2}^{K-2} \circ \ldots \circ \psi_{i_m}^{K-m},$$

and

$$\Phi_I = \phi_{i_m}^{K-m+1} \circ \phi_{i_{m-1}}^{K-m} \circ \ldots \circ \phi_{i_1}^{K}.$$

It is worth noting that both Ψ_I and Φ_I are linear maps, and may be therefore represented by matrices A_Ψ and B_Φ. For example the columns of B_Φ are the $K - m$ standard basis vectors $\{e_j | j \notin I\}$.

Given the definitions and relations (3.25) and (3.26), it follows that $\Phi_I(S_K^I) = S_{K-m}$, $S_K^I = \Psi_I(S_{K-m})$, and that Φ_I, Ψ_I restricted to S_K^I act as homeomorphisms between S_K^I and S_{K-m}. Given this identification, it is easy to see that $A_\Psi = B_\Phi^\intercal$ The insertion and deletion maps are inspired by the face and degeneracy maps of classical simplicial complexes. For further reference, see the reviews

[Fri12, nLa20a, nLa20b].

Theorem 3.5 gives uniqueness of a maximizing fixed point $\hat{\boldsymbol{\pi}}$ when $\hat{\boldsymbol{\pi}} \in \operatorname{Int} S_K$. Since $\ell_F(\boldsymbol{\pi})$ is guaranteed to have a maximum on S_K, the only other situation that requires consideration is the case where $\hat{\boldsymbol{\pi}} \in \partial S_K$. Fortunately, both equations 3.23 and 3.24 still apply on the facets of S_K. Thus theorem 3.5 still applies through the following lemma.

Lemma 3.6. *Suppose $\hat{\boldsymbol{\pi}} \in S_K^I \subsetneq \partial S_K$ is a fixed point of $\mathbb{R}_F(\boldsymbol{\pi})$. Then $\hat{\boldsymbol{\pi}}$ is a maximizing fixed point for R_F and ℓ_F restricted to S_K^I. Further, by regarding $\nabla \ell_F(\hat{\boldsymbol{\pi}})$ as a vector in $\mathbb{R}^K$, the gradient at $\hat{\boldsymbol{\pi}}$ satisfies $\Phi_I(\nabla \ell_F(\hat{\boldsymbol{\pi}})) = \mathbb{1}_{K-m}$.*

Proof. Let $m = |I|$ be the cardinality of I. Equipped with the maps Φ_I and Ψ_I, consider first the case $m = 1$ for purposes of illustration. So suppose without loss of generality $\hat{\boldsymbol{\pi}}$ has $\hat{\pi}_1 = 0$ i.e. $I = \{1\}$. Then let $F^{\phi_1} := [\phi_1^K(F_1), \ldots, \phi_1^K(F_N)]$. Now note that saying $\hat{\boldsymbol{\pi}}$ maximizes $\ell_F(\boldsymbol{\pi})$ is equivalent to the statement $\ell_{F^{\phi_1}}(\boldsymbol{\pi})$ is maximized on S_{K-1} by $\phi_1^K(\hat{\boldsymbol{\pi}})$. This is true because for $\boldsymbol{\pi} \in S_K^I$,
$$\sum_i f_{ij} \pi_i = \sum_{i \notin I} f_{ij} \pi_i$$

Now if $M > 1$, a similar construction gives the first result. Namely, given the index set I, let $F^\Phi := [\Phi_I(F_1), \ldots, \Phi_I(F_N)]$. Then the maps $\ell_{F^\Phi} \circ \Phi_I$ and ℓ_F are equal on S_K^I. Similarly, the maps ℓ_{F^Φ} and $\ell_F \circ \Psi_I$ are equal on S_{K-m}. In particular, ℓ_{F^Φ} is maximized at $\Phi_I(\hat{\boldsymbol{\pi}})$.

Since $\hat{\boldsymbol{\pi}} \in S_K^I$, the point $\Phi_I(\hat{\boldsymbol{\pi}}) \in \operatorname{Int}(S_{K-m})$, so theorem 3.5 applies and so $\nabla \ell_{F^\Phi}(\Phi_I(\hat{\boldsymbol{\pi}})) = \mathbb{1}_{K-m}$. Since $\ell_{F^\Phi} = \ell_F \circ \Psi_I$, applying lemma 2.1 gives

$$\nabla \ell_{F^\Phi}(\boldsymbol{\nu}) = D^* \Psi_I[\nabla \ell_F(\Psi_I(\boldsymbol{\nu}))]$$

for $\boldsymbol{\nu} \in \operatorname{Int}(S_{K-m})$. Since Ψ_I is linear, $D\Psi_I = \Psi_I$ which can be identified with the matrix A_Ψ. So

$$D^* \Psi_I = A_\Psi^\mathsf{T} = B_\Phi,$$

and so

$$D^* \Psi_I[\nabla \ell_F(\Psi_I(\boldsymbol{\nu}))] = \Phi_I(\nabla \ell_F(\Psi_I(\boldsymbol{\nu}))).$$

Since $\Psi_I(\Phi_I(\hat{\boldsymbol{\pi}})) = \hat{\boldsymbol{\pi}}$, combining the above equations gives the lemma. $\qquad\square$

Remark 3.8. If $\hat{\pi}_{i_j} = 0$ for $j = 1, \ldots, m < K$ then it is very likely that for $\boldsymbol{\pi} \in S_K^I$

$$\frac{\partial \ell_F(\boldsymbol{\pi})}{\partial \pi_{i_j}} \neq \frac{\partial \ell_{F^\Phi}(\Phi_I^K(\boldsymbol{\pi}))}{\partial \pi_{i_j}} \quad j = 1, \ldots, m.$$

In fact, unless $\hat{\boldsymbol{\pi}}$ gives the exact maximum of $\ell_F(\boldsymbol{\pi})$ on the affine plane $A_K \supset S_K$, then $\frac{\partial \ell_F}{\partial \pi_{i_m}} \neq 0$ for all m. More precisely the gradient $\nabla \ell_F$ will point 'out' of S_K in the sense that $\arg\max_{A_K} \ell_F(\boldsymbol{\pi})$ will have negative barycentric coordinates.

This idea can be expressed geometrically. If $\boldsymbol{n}$ is the normal vector to the facet S_K^I containing $\hat{\boldsymbol{\pi}}$, then $\langle \boldsymbol{n}, \nabla \ell_F(\boldsymbol{\pi}) \rangle > 0$. This happens precisely when $\ell_F(\boldsymbol{\pi})$ has no maximum on the interior of S_K.

Together theorem 3.5 and lemma 3.6 show that any fixed point of R_F on the simplex S_K will be a critical point of the constrained optimization problem given by equations (2.26) and (2.27). When ℓ_F is strictly convex, these points must be isolated. As a final lemma before showing that dynamic responsibility described in algorithm 1 converges, [Ryc20] gives the following powerful result.

Lemma 3.7. *The function* $-\ell_F(\boldsymbol{\pi}) : \mathbb{R}_+^K \to \mathbb{R}$ *is a Lyapunov function for* $R_F(\boldsymbol{\pi})$ *on* S_K. *Further,* $\ell_F(R_F(\boldsymbol{\pi})) - \ell_F(\boldsymbol{\pi}) = 0$ *iff* $R_F(\boldsymbol{\pi}) = \boldsymbol{\pi}$, *i.e.* $\boldsymbol{\pi}$ *is a fixed point of* R_F.

Proof. Before showing this, it is important to note that this proof does not require F to have full rank. Thus it holds for all $K \times N$ matrices F with strictly positive entries.

The proof is given by the following calculations. First, define $\dot{\ell}_F(\boldsymbol{\pi}) = \ell_F(R_F(\boldsymbol{\pi})) - \ell_F(\boldsymbol{\pi})$. Then

$$\dot{\ell}_F(\boldsymbol{\pi}) = \ell_F(R_F(\boldsymbol{\pi})) - \ell_F(\boldsymbol{\pi}) = \frac{1}{N}\sum_{n=1}^{N} \log\left\{\sum_{i=1}^{K} \pi_i f_{in} \frac{\partial \ell}{\partial \pi_i}\right\} - \log\left\{\sum_{k=1}^{K} \pi_k f_{kn}\right\} \tag{3.27}$$

$$= \frac{1}{N}\sum_{n=1}^{N} \log\left\{\frac{\sum_{i=1}^{K} \pi_i f_{in} \frac{\partial \ell}{\partial \pi_i}}{\sum_{k=1}^{K} \pi_k f_{kn}}\right\} \tag{3.28}$$

$$\geq \sum_{n=1}^{N}\sum_{i=1}^{K} \frac{1}{N}\frac{\pi_i f_{in}}{\sum_{k=1}^{K} \pi_k f_{kn}} \log\left(\frac{\partial \ell}{\partial \pi_i}\right). \tag{3.29}$$

Where equation (3.28) is greater than (3.29) because $\ell_F(\boldsymbol{\pi})$ is concave, and

$$\sum_{i} \frac{\pi_i f_{in}}{\sum_k \pi_k f_{kn}} = 1.$$

Swapping the order of the sums in (3.29), and recalling that $r_i(\boldsymbol{\pi}) = \pi_i \left. \frac{\partial \ell}{\partial \pi_i} \right|_{\boldsymbol{\pi}}$ gives

$$\sum_{i=1}^{K} \sum_{n=1}^{N} \frac{1}{N} \frac{\pi_i f_{in}}{\sum_{k=1}^{K} \pi_k f_{kn}} \log\left(\frac{\partial \ell}{\partial \pi_i}\right) = \sum_{i=1}^{K} r_i(\boldsymbol{\pi}) \log\left(\frac{r_i(\boldsymbol{\pi})}{\pi_i}\right). \tag{3.30}$$

Recall that $\sum_i r_i(\boldsymbol{\pi}) = \sum_k \pi_k = 1$, so they both can be considered discrete probability distributions. Thus Gibbs' inequality [Mac02, §2.6] gives

$$\sum_{i=1}^{K} r_i(\boldsymbol{\pi}) \log\left(\frac{r_i(\boldsymbol{\pi})}{\pi_i}\right) \geq 0. \tag{3.31}$$

Combining equations (3.27) through (3.31) gives the first result. The final inequality (3.31) is an equality iff $\pi_i = r_i(\boldsymbol{\pi})$ for all $1 \leq i \leq K$, which proves the last part of the lemma. $\qquad\square$

This lemma can best be summarized by the following quote from MacKay [Mac02, §2.6]: "Gibbs' inequality is probably the most important inequality in this book."

Describing a Lyapunov function for a given dynamical system is difficult endeavor, but provides powerful results. For example, lemma 3.7 is sufficient to show that the only limit points of $\boldsymbol{\pi} \in S_K$ are fixed points of R_F. In fact, combining lemmas 3.6, 3.7, and theorem 3.5 gives that the set $E = \{\boldsymbol{\pi} | \dot{\ell}_F(\boldsymbol{\pi}) = 0\}$ consists entirely of fixed points of R_F. The following theorem uses this fact to prove convergence.

Theorem 3.8. If F has full rank, and $\boldsymbol{\pi}_0 \in \text{Int } S_K$ then iteration of $R_F(\boldsymbol{\pi}_0)$ converges to $\hat{\boldsymbol{\pi}}$, the unique maximizing fixed point of $\ell_F(\boldsymbol{\pi})$ on S_K.

Proof. Let $\boldsymbol{\pi}_0 \in \text{Int}(S_K)$ Existence and uniqueness comes entirely from the fact that $-\ell_F(\boldsymbol{\pi})$ is strictly convex and acts as a Lyapunov function for $R_F(\boldsymbol{\pi})$. To show this precisely, theorem 3.1 in chapter 4 of LaSalle [LS76], as written in theorem 2.2, must be applied. To apply this theorem it is sufficient that the orbit $R_F^n(\boldsymbol{\pi}_0)$ stay in $\text{Int}(S_K)$. Since this is true, then it follows that $\Omega(\boldsymbol{\pi}_0) = H \cap \ell_F^{-1}(c)$ for some $c \in \mathbb{R}$. Here $\Omega(\boldsymbol{\pi}_0)$ is the limit point set of $\boldsymbol{\pi}_0$ as described in definition 2.4, and H is the largest invariant subset of the set $E := \{\boldsymbol{x} \in S_K | V(T(\boldsymbol{x})) - V(\boldsymbol{x}) = 0\}$ as in theorem 2.2.

The set $E = H$ consists of only fixed points of $R_F(\boldsymbol{\pi})$ by lemma 3.7. Because F has full rank, these points must be isolated. Let $a = \ell_F(\hat{\boldsymbol{\pi}})$, then the claim is that $\Omega(\boldsymbol{\pi}_0) = E \cap \ell_F^{-1}(a) = \{\hat{\boldsymbol{\pi}}\}$. To see this consider two cases, $\hat{\boldsymbol{\pi}} \in \text{Int}(S_K)$ and $\hat{\boldsymbol{\pi}} \in \partial S_K$.

For both cases it is helpful to consider the set $\omega(\boldsymbol{\pi}_0) = \{\boldsymbol{\pi} | \ell_F(\boldsymbol{\pi}) \geq \ell_F(\boldsymbol{\pi}_0)\}$. Clearly, this set is positively invariant, convex, and contains $\hat{\boldsymbol{\pi}}$. Moreover, the orbit $R_F^n(\boldsymbol{\pi}_0) \subset \omega(\boldsymbol{\pi}_0) \cap \text{Int}(S_K)$. If $\hat{\boldsymbol{\pi}} \in \text{Int}(S_K)$, then because fixed points of $R_F(\boldsymbol{\pi})$ are isolated when F has full rank, $\hat{\boldsymbol{\pi}}$ is the only fixed point in $\omega(\boldsymbol{\pi}_0) \cap \text{Int}(S_K)$. Thus it must be the case that $\Omega(\boldsymbol{\pi}_0) = \{\hat{\boldsymbol{\pi}}\}$ when $\hat{\boldsymbol{\pi}} \in \text{Int } S_K$.

Now if $\hat{\boldsymbol{\pi}} \in \partial S_K$, fix some $n \in \mathbb{N}$ and consider the set $R_F^n(\omega(\boldsymbol{\pi}_0))$. This set contains $\hat{\boldsymbol{\pi}}$, $\{R_F^m(\boldsymbol{\pi}_0) | m \geq n\}$ and $R_F^n(\omega(\boldsymbol{\pi}_0)) \cap \text{Int}(S_K) \neq \varnothing$. Thus there is some relatively open set $U_n \subset S_K$ containing both $\Omega(\boldsymbol{\pi}_0)$ and $\{\hat{\boldsymbol{\pi}}\}$. Since this is true for all $n \in \mathbb{N}$, the set $\bigcap_{n \in \mathbb{N}} U_n = \Omega(\boldsymbol{\pi}_0) = \{\hat{\boldsymbol{\pi}}\}$ $\square$

As a brief remark, corollary 4.2.1 of Michel *et al.* [MHL15] gives asymptotic stability of $\{\hat{\boldsymbol{\pi}}\}$ for any orbit $R_F^n(\boldsymbol{\pi}_0)$ starting at some $\boldsymbol{\pi}_0 \in \text{Int}(S_K)$. This will provide strong implications for the derivative DR in chapter 4, section 4.4.

Applying theorem 3.8 to each face of S_K shows that each facet $S_K^I = \Psi_I(S_{K-|I|})$ is a region of attraction for some point $\hat{\boldsymbol{\pi}}_I \in S_K^I$. For some full rank parameter matrices F it is the case that one or more of these $\hat{\boldsymbol{\pi}}_I$ coincide. This happens precisely when $\hat{\boldsymbol{\pi}} \in \partial S_K$. This sort of behavior is more common when F is not full rank, as shown by example 3.9.

Example 3.9. As an example of what can happen when F does not have full rank, let $K = 2$, and $\boldsymbol{F} = (f_{i,j})$ be a $K \times N$ matrix with positive entries and full rank. (Here $N \gg K$). Define $R_F(\boldsymbol{\pi})$ as the responsibility map on the simplex S_2. Also let $\hat{\boldsymbol{\pi}} = (\hat{\pi}_1, \hat{\pi}_2)^\mathsf{T}$ be the fixed point of $R_F(\boldsymbol{\pi})$.

Then for positive $\lambda \in \mathbb{R}$ and a fixed $a \in (0, .5)$, let

$$
\boldsymbol{A}_\lambda = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ \lambda a & \lambda(1 - a) \end{pmatrix}
$$

and $\boldsymbol{G}_\lambda = \boldsymbol{A}_\lambda \boldsymbol{F}$.

Then $R_{G_\lambda}(\boldsymbol{\pi})\colon S_3 \to S_3$ exhibits a bifurcation at $\lambda = 1$.

For $\lambda < 1$ there is a heteroclinic orbit (in S_3) going from $(0, c, 1 - c)^\mathsf{T}$ to $(\hat{\pi}_1, \hat{\pi}_2, 0)^\mathsf{T}$, where $c = \hat{\pi}_2 - \frac{a}{1-a}\hat{\pi}_1$ (note, this is if $\hat{\pi}_1 < \hat{\pi}_2$, otherwise the roles of $\hat{\pi}_1, \hat{\pi}_2$ reverse!)

For $\lambda > 1$, the direction of the heteroclinic orbit reverses.

At $\lambda = 1$, the entire line in S_3 between $(0, c, 1 - c)^\mathsf{T}$ and $(\hat{\pi}_1, \hat{\pi}_2, 0)^\mathsf{T}$ consists of fixed points of $R_{G_\lambda}(\boldsymbol{\pi})$.

As a partial generalization of example 3.9, we have the following theorem.

Theorem 3.9. If F does not have full rank, then the set of fixed points for the map $R(\boldsymbol{\pi})$ is the intersection of a linear subspace of $\mathbb{R}^K$ with S_K.

Proof. In light of theorem 3.5 and corollary 3.3, it is enough to show the set of maximal points for $\ell_F(\boldsymbol{\pi})$ satisfies the conclusion.

Since F is not full rank, it is possible that $\ell_F(\boldsymbol{\pi})$ is not strictly convex on S_K. Let $\mathcal{M} \subseteq S_K$ be the set of minimizers for $\ell(\boldsymbol{\pi})$, and let m be the minimum value achieved by $\ell(\boldsymbol{\pi})$ on $\mathcal{M}$. Without loss of generality we may suppose that $\ell_F(\boldsymbol{\pi})$ is not strictly convex, so that $\mathcal{M}$ is not a single point. In particular, $\mathcal{M}$ will be relatively open in S_K. This will mean that $\mathcal{M}$ will be compact and contain at most $K - 1$ linearly independent vectors.

The level set $\mathcal{C}_m := \left\{ \boldsymbol{\pi} \in \mathbb{R}^K \,|\, \ell_F(\boldsymbol{\pi}) = m \right\}$ is an algebraic subset of $\mathbb{R}^K$ cut out by the equation $L(\boldsymbol{\pi}) = \exp(Nm)$. Now by Bezout's theorem, we know that for a linear subvariety $\mathcal{L}$ of $\mathbb{R}^K$, either $\mathcal{C}_m \bigcap \mathcal{L}$ is finite, or $\mathcal{L} \subset \mathcal{C}_m$. Since $\mathcal{M} \subset \mathcal{C}_m$, we know that if $\mathcal{L} \subset \mathbb{R}^K$ is an affine linear subvariety that includes $\mathcal{M}$ as a relatively open subset, then $\mathcal{L} \subset \mathcal{C}_m$.

Now $\mathcal{M}$ is a convex set, as $\ell_F(\boldsymbol{\pi})$ is a convex function. In particular, $\mathcal{M}$ is contained in some linear subset $M \subset \mathbb{R}^K$. As $\mathcal{M}$ must be relatively open in M, C_m must contain all of M. Since $\mathcal{M}$ is precisely those elements of M which reside in S_K, $\mathcal{M} = S_K \bigcap M$. $\qquad \square$

Note that in the proof of theorem 3.9, $\mathcal{M}$ is a proper subset of S_K which has dimension less than or equal to $K - 1$. If the dimension of $\mathcal{M}$ is exactly $K - 1$, this precludes $R_F(\boldsymbol{\pi})$ from doing anything interesting, as then $\mathcal{M} = S_K$. The only example of a parameter matrix which will do some thing like this is the matrix $c \cdot \mathbb{1}_{K \times N}$, where c is any real constant and $\mathbb{1}_{K \times N}$ is the matrix of all ones.

3.5 Relationship to the MLE

Theorem 3.5 proves that looking for the fixed points of $R_F(\boldsymbol{\pi})$ when F has full rank is equivalent to finding a maximum likelihood estimator. This means that maximization algorithms work in such cases to find the fixed point of $R_F(\boldsymbol{\pi})$ on the interior of S_K. More importantly, with a little bit of work results from [Wal49, Pol81, Pol82] adapt to give many of the same properties for this algorithm as we would for the MLE. In particular, these things prove consistency of the fixed point $\hat{\boldsymbol{\pi}}$ as an estimator for the point $\boldsymbol{\pi}^* = \{\pi_k^*\}_{k=1}^K$ which describes the mixing proportions of a mixture model.

Algorithm 2 uses Newton's method to maximize $\ell(\boldsymbol{\pi})$ on S_K. This algorithm requires using Hessian of $\ell(\boldsymbol{\pi})$, but this is calculated in the proof of lemma 3.4, equation (3.21).

Algorithm 2 Maximization Algorithm

Require: F a $K \times N$ matrix
Require: $\boldsymbol{\pi}_0$, ϵ
  procedure Newton Maximization$(F, \boldsymbol{\pi}_0, \epsilon)$
      $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi}_0$
      $\Delta\boldsymbol{\pi} \leftarrow \mathbb{1}_K$
      while $\|\Delta\boldsymbol{\pi}\| > \epsilon\|\boldsymbol{\pi}\|$ **do**
$$\boldsymbol{D}_\ell(\boldsymbol{\pi}) \leftarrow \begin{bmatrix} H_\ell(\boldsymbol{\pi}) & \mathbb{1}_K \\ \mathbb{1}_K^\top & 0 \end{bmatrix}$$
         $\Delta\boldsymbol{\pi} \leftarrow \boldsymbol{D}_\ell(\boldsymbol{\pi})^{-1}(\nabla\ell(\boldsymbol{\pi}), 0)$
         $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} + \Delta\boldsymbol{\pi}$
      end while
      return $\boldsymbol{\pi}$ $\triangleright$ at this point $\boldsymbol{\pi}$ is approximately $\hat{\boldsymbol{\pi}}$
  end procedure

Table 3.2: A Newton Method version of the Main Algorithm

The column vector $\mathbb{1}_K$ as used in this algorithm is defined in the proof to theorem 3.5. A version of this algorithm is found in appendix A.3, file `stablepointNewton.m`.

Using the block matrix in this algorithm ensures that $\boldsymbol{\pi} + \Delta\boldsymbol{\pi}$ stays in S_K. This is equivalent to the geometric idea that $\Delta\boldsymbol{\pi}$ be restricted to the tangent plane of S_K. In light of theorem 3.5, it is also worth noting that this algorithm gives $\Delta\boldsymbol{\pi} = 0$ precisely when $\nabla\ell(\boldsymbol{\pi}) = \mathbb{1}_K$.

One set of issues this algorithm resolves is the problem discussed in remark 3.3. While algorithm 1 cannot leave ∂S_K when given an initial value in ∂S_K, algorithm 2 will not do this. However, in practice algorithm 2 tends to be more sensitive to the conditions of the matrix F than algorithm 1.

The biggest problem that 2 faces happens when $\arg\max_{S_K} \ell_F(\boldsymbol{\pi}) = \hat{\boldsymbol{\pi}} \in \partial S_K$. In this case remark 3.8 shows that following the gradient will lead to a maximum outside of S_K. Such behavior happens fairly often in practical situations, especially when working with imbalanced data.

The use of algorithm 2 might be recovered via use of KKT conditions as in section 2.6. Another common method is to add some sort of additional term such as $-H(\boldsymbol{\pi})$, the entropy of $\boldsymbol{\pi}$, to prevent $\ell_F(\boldsymbol{\pi})$ from having a maximum on the edge. This technique is sometimes called the 'free energy' approach, and comparing the two methods would be a good topic for future work.

I now turn to examining the behavior of $\hat{\boldsymbol{\pi}}$ as an estimator of $\boldsymbol{\pi}^*$. It is worthwhile to look at some experimental data. As N increases $\hat{\boldsymbol{\pi}}$, as described in theorem 3.8, approaches $\boldsymbol{\pi}^*$.

Table 3.3 summarizes some numerical experiments done with the scripts `GMMData.m` and `error_samples.m` in appendix A.4. The version of algorithm 1 in the file `stablepoint.m` was used. For each N, 10000

N	$\boldsymbol{\mu}$	$\boldsymbol{\sigma^2}$
10	0.372980	0.04218
100	0.133670	0.008056
1000	0.042706	0.000884
100000	0.004335	0.00000952

Table 3.3: Experimental Evidence of Consistency

trials were generated and the difference

$$\Delta\boldsymbol{\pi} = \boldsymbol{\pi}^* - \hat{\boldsymbol{\pi}}$$

was recorded for each trial.

The second column records the mean of $\|\Delta\boldsymbol{\pi}\|$ for the trials. The third column shows the variance of $\|\Delta\boldsymbol{\pi}\|$ for the trials. As expected for a consistent estimator, the error shrinks like $O\left(\dfrac{1}{\sqrt{N}}\right)$.

As $L_F(\boldsymbol{\pi})$ is formed from the likelihood function of a given sample, and iterating $R_F(\boldsymbol{\pi})$ gives a maximum for $\ell_F(\boldsymbol{\pi})$ on S_K, finding a fixed point as in algorithms 1 or 2 is calculating argmax of the Log likelihood. In other words, $\hat{\boldsymbol{\pi}}$ is a MLE for $\boldsymbol{\pi}^*$.

Chapter 4

RESPONSIBLE SOFTMAX LAYER

Given the method from chapter 3 for determining an approximation to $\boldsymbol{\pi}$ given $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$, this chapter focuses on a method for determination of the distributions f_k. A major drawback of the fixed point iteration method is that it requires the ability to evaluate $f_k(\boldsymbol{x}^{(n)}, \boldsymbol{\theta}_k)$ for all n, k. If these functions cannot be evaluated, it becomes a barrier to creating a reliable model. Algorithms such as the EM algorithm or the closely related variational auto-encoder [KW13, vdOV+17] solve this by making the assumption that the functions f_k have a specific form, usually Gaussian. This has the drawback that the model can be inflexible. Data that is not linearly separable such as the classic bulls-eye, or paired moons data sets, require difficult adjustments to the mentioned algorithms.

On the other hand, there are techniques such as Bayesian nonparametric models and hierarchical mixtures of experts that are more flexible in their restrictions. These models offer greater power, but at the expense of greater computational complexity. In both cases overfitting becomes a worry, in the sense that there are not always guarantees that the variance of the given model will be computationally bounded away from zero.

This chapter proposes an algorithm for modeling the distributions f_k that fits somewhere between these two extremes. It has the great advantage that it fits in existing deep network structures. This allows the use of high powered computational techniques such as stochastic gradient descent. It also gives the flexibility of more powerful models and still preserves the structure of a mixture model on the given data.

4.1 Neural Net Learning of the Parameters for Dynamic Responsibility

Let us suppose that $X \subset \mathcal{X}$ can be modeled by a mixture model of the form

$$p(\boldsymbol{x}; \boldsymbol{\pi}, \boldsymbol{\Theta}) = \sum_{k=1}^{K} \pi_k f_k(\boldsymbol{x}, \boldsymbol{\theta}_k),$$

as in section 2.2. If the functions $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$ are known, the mixing components $\boldsymbol{\pi}^*$ may be approximated by the fixed-point process $R(F, \hat{\boldsymbol{\pi}}) = \hat{\boldsymbol{\pi}}$ where $(F)_k^n = f_k(\boldsymbol{x}^{(n)}, \boldsymbol{\theta}_k)$. This requires that $\hat{\pi}_k \geq 0$ for all $k \leq K$, so set $\hat{\mu}_k = \log \hat{\pi}_k$, with the requirement that $\hat{\mu}_k = -\infty$ if $\hat{\pi}_k = 0$.

Let T be the training targets of the data, and Y the output of the neural network. This dissertation uses a feedforward neural network, as is typical with deep learning. This discussion will only focus on the final few layers of the network.

Now, in the case of supervised learning, targets T for some training set are given. Often it is the case that these targets are *one-hot encoded* as discussed in section 2.2. This means that $\boldsymbol{t}^{(n)} \in \{0, 1\}^K$, and $\sum_k t_k^{(n)} = 1$. This requires that the $t_k^{(n)}$ are all zero except the entry corresponding to the class of $\boldsymbol{x}^{(n)}$. One shorthand way of writing this is to say that $t_k^{(n)} = \delta_{\ell_n}^k$, where ℓ_n is the class label of $\boldsymbol{x}^{(n)}$, and $\delta_{\ell_n}^k$ is the Kronecker delta function.

Using targets in neural networks also allows $\boldsymbol{t}^{(n)} \in (0, 1)^K$, but enforces the requirement that $\sum_k t_k^{(n)} = 1$. This is slightly more flexible than one-hot encoding, and represents the situation where the labeling is less certain. This is closely related to the regularization technique of label smoothing as mentioned in Müller [MKH19]. Either one hot labels or smooth labels will be reasonable for training targets. The first situation is more typical of classification problems, the second may be treated more like regression. Feedforward neural nets are suitable for both activities.

It is worth looking briefly at the relationship of targets T and class labels $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_N\}$. In the case that T is one-hot encoded, say $t_k^{(n)} = \delta_{\ell_n}^k$, then the entries of T are expressing certainty concerning the the label ℓ_n. This situation may be viewed as expressing the idea that $P(\ell_n = k | \boldsymbol{x}^{(n)}) = \delta_k^{\ell_n}$. If T is instead a list of categorical distributions, *e.g.* $t_k^{(n)} \in [0, 1]$ and $\sum_k t_k^{(n)} = 1$, then the model expresses more uncertainty about the labels. It is still the case that $P(\ell_n = k | \boldsymbol{x}^{(n)}) = t_k^{(n)}$.

Given training data X, building the neural network requires a choice of loss function and activation function(s) to enforce the mixture model structure on $X \sim \sum_{k=1}^K \pi_k f_k(\boldsymbol{x}, \boldsymbol{\Theta})$, where $\boldsymbol{\Theta} = \boldsymbol{W}$ are the weights of the neural network. To save time, this notation leaves out the direct dependence of the f_k on the weights unless necessary. Since the goal is classification $y_k^{(n)}$, the outputs of the neural net on $x^{(n)}$, should approximate the conditional probability that label ℓ_n is k given $\boldsymbol{x}^{(n)}$, *i.e.* $y_k^{(n)} \approx P(\ell_n = k | \boldsymbol{x}^{(n)}) = t_k^{(n)}$. Of course, this requires that $\sum_k y_k^{(n)} = 1$. Denote outputs of the neural network on the n-sample by $(Y)_k^n := y_k^{(n)}$.

As dynamic responsibility assumes a mixture model for X, it also supposes that $P(x^{(n)} | \ell_n = k) \sim$

$f_k(x^{(n)})$. Then by Bayes' rule

$$P(\ell_n = k | \boldsymbol{x}^{(n)}) = \frac{P(\boldsymbol{x}^{(n)} | \ell_n = k) P(\ell_n = k)}{\sum_i P(\boldsymbol{x}^{(n)} | \ell_n = i) P(\ell_n = i)} \tag{4.1}$$
$$= \frac{f_k(x^{(n)}) P(\ell_n = k)}{\sum_i f_i(x^{(n)}) P(\ell_n = i)}.$$

which leads to the definition

$$y_k^{(n)} = \frac{\pi_k f_k(x^{(n)})}{\sum_{i=1}^K \pi_i f_i(x^{(n)})}. \tag{4.2}$$

Note here that this definition does not yet include a dynamic responsibility approximation of $\boldsymbol{\pi}^*$. Rather this is showing that for mixture models, Bayesian inference can provide guidance on network output. Also, if $\pi_i = \frac{1}{K}$ for all i, then equation (4.2) is equivalent to the standard softmax output in common use.

4.1.1 Choice of Loss Function

A good goal in a situation like this is to minimize the difference between Y and T. Since both Y and T are distributions, multiple candidates for loss functions work well to have Y approximate T. The method used here is that of cross entropy loss, though other common ones are mean squared error [Bis95] and connectionist temporal classification loss [GFG06].

Conflating the ideas of a probability distribution and its underlying measure, define cross-entropy of two probability distributions $p(x)$ and $q(x)$ on the same underlying probability space $\mathcal{X}$ by the quantity

$$H(p, q) := E_p[-\log(q)] = -\int_{\mathcal{X}} p \log(q) d\mu. \tag{4.3}$$

Where μ is any measure on $\mathcal{X}$ such that p and q are absolutely continuous with respect to μ.

The cross entropy is closely related to the Kullback-Leibler (KL) divergence from q to p. The KL divergence from q to p is defined as

$$D_{\mathrm{KL}}(p \parallel q) := \int_{\mathcal{X}} p(x) \log\left(\frac{p}{q}\right) d\mu. \tag{4.4}$$

then

$$H(p, q) = D_{\mathrm{KL}}(p \parallel q) + H(p), \tag{4.5}$$

where

$$H(p) = -\int_{\mathcal{X}} p \log(p) d\mu \tag{4.6}$$

is the Shannon Entropy of p.

In the case discussed here, $P(\mathcal{L}|X) = T$ is a fixed distribution, and so the Shannon Entropy $H(T)$ is constant. Thus finding a distribution Y such that $H(Y, T)$ is minimized also finds a distribution which minimizes $D_{\mathrm{KL}}(T \parallel Y)$, the Kullback-Leibler divergence of Y from T.

There are many reasons to choose cross-entropy loss, but one that motivates responsible softmax comes from Neal and Hinton [NH98]. It is the case, as Neal and Hinton show, that a form of cross entropy acts as a Lyapunov function for the EM algorithm. In other words, each step, either expectation or maximization, of the EM algorithm reduces the cross entropy between a target and predicted distribution. They further use this fact to show that any improvement in either step reduces the loss. This idea returns in section 4.2.

4.1.2 Method for determining F

As theorem 3.8 shows, if the matrix F given by $(F)_k^n = f_k(\boldsymbol{x}^{(n)}, \boldsymbol{\theta}_k)$ has linearly independent rows, iteration of $R_F(\boldsymbol{\pi})$ converges to a unique fixed point $\hat{\boldsymbol{\pi}}$ on the interior of S_K. Note here that for theorem 3.8 to work, the functions $f_k(\boldsymbol{x}^{(n)}, \boldsymbol{\theta}_k)$ must be positive. As the functions seek to approximate pdfs, this should not pose a problem.

Since the output of a neural network seeks to approximate these pdfs, exponentiation of the output is reasonable. This ensures positivity of the output, and almost surely guarantees that F will satisfy the linear independence requirement. Because the function $R(F, \boldsymbol{p})$ is homogeneous in both the rows and columns of F, it makes sense to use the typical softmax output of a neural network as discussed near equation 2.6.

With the task of approximating F in mind, this section revisits the structure of Neural Networks. To simplify things, consider the case with one fully connected hidden layer as in figure 2.2. This fully connected layer takes the data $\boldsymbol{X}$ as input and outputs an activation $\boldsymbol{A}$ by combination of a linear transformation and a non-linear activation function. To be precise, if $\boldsymbol{X} = (\boldsymbol{x}^{(n)})_{n \leq N}$ for $\boldsymbol{x}^{(n)} \in \mathbb{R}^D$ then $\boldsymbol{A} = (\boldsymbol{a}^{(n)})_{n \leq N}$ where $\boldsymbol{a}^{(n)} \in \mathbb{R}^K$ and

$$\boldsymbol{a}^{(n)} = \sigma\left(\boldsymbol{W}\boldsymbol{x}^{(n)}\right),$$

with $\boldsymbol{W} = (w_{ij})$ a $K \times D$ matrix of weights and σ a chosen non-linear activation function. For simplicity, let us suppose that $\sigma(x) = (1 + e^{-x})^{-1}$ is the sigmoidal activation function for now. A common identification of the vector space of such matrices and the space of linear transformations $W \in L(\mathbb{R}^D, \mathbb{R}^K)$, $W : \mathbb{R}^D \to \mathbb{R}^K$ is taken in this situation.

In typical neural networks used for classification the activations are then transformed by the softmax function. However, per the discussion in section 2.3, first make a different transformation

$$F = e^{\boldsymbol{A}}$$

so that $\boldsymbol{f}^{(n)} = e^{\boldsymbol{a}^{(n)}}$ is a column vector in $\mathbb{R}^K$ with non-zero entries. Doing this implies $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$ can be well approximated by an exponential family distribution. This also means that $F \in M_{K,N}$, and defines $\hat{\boldsymbol{\pi}}_F$ as in equation 4.17. Then for classification use the matrix

$$Y(F, \hat{\boldsymbol{\pi}}) = \left(\frac{\hat{\pi}_i f_{ij}}{\sum_{k=1}^K \hat{\pi}_k f_{kj}} \right)$$

as in equation 4.2.

In connection with the softmax function $\boldsymbol{\sigma} : \mathbb{R}^K \to \mathbb{R}^K$ and section 2.3, it is worth noting that by defining $\hat{\boldsymbol{\mu}}_F = \log(\hat{\boldsymbol{\pi}}_F)$, $Y(F, \boldsymbol{p})$ changes in the following manner. First note that because F depends on $\boldsymbol{A}$, then so does $\log \hat{\boldsymbol{\pi}}_F = \hat{\boldsymbol{\mu}}_F$. Then because $\hat{\pi}_i f_{ij} = \exp(\hat{\mu}_i + a_{ij})$ for $1 \leq i \leq K$ and $1 \leq j \leq N$,

$$Y(F, \hat{\boldsymbol{\pi}}) = Y(\boldsymbol{A}) := \sigma(\boldsymbol{A} + \hat{\boldsymbol{\mu}}(\boldsymbol{A})). \tag{4.7}$$

This recalls the fact that $R(F, \boldsymbol{\pi})$ is the gradient of $\ell(F, \boldsymbol{\pi})$ with respect to coordinates $\boldsymbol{\mu} = \log(\boldsymbol{\pi})$.

4.2 Proposed Neural Network Layer

This section describes the *responsible softmax layer* as a method to use dynamic responsibility in approximating class weights $\hat{\boldsymbol{\pi}}$ and class density functions $f_i(x)$. The responsible softmax layer uses a combination of dynamic responsibility and backpropagation to determine class weights $\hat{\boldsymbol{\pi}}$ for weighting a typical softmax layer. While weighted softmax layers are not new, the typical approach is to apply the same weight for every class. This weight is either set as a hyper parameter, or is occasionally determined through backpropagation. On the occasion that multiple class weights are used, they are usually set once to proportions determined *a priori* by class labels T and are not learned parameters.

Chapter 5 discusses when each approach might be reasonable.

To discuss responsible softmax (RS), understanding of weighted softmax is required. Recall from section 2.3 equation 2.4 that softmax is the multivariable extension of the logistic function. A weighted softmax adds multiplication weights to each exponential function in the usual softmax *i.e.*

$$\sigma_i(\boldsymbol{x}, \boldsymbol{\beta}) = \frac{e^{x_i \beta_i}}{\sum_j e^{x_j \beta_j}}. \tag{4.8}$$

Equation 4.8 for weighted softmax is very similar to equations 2.10 and 2.16 from chapter 2. One paper that uses a weighted softmax approach is that of Liu *et al.* [LWYY16]. This paper defines L-softmax, which gives a different weight to each class.

Another brief discussion for weighting the softmax loss is implied in Bishop [Bis95, sect. 6.5]. Bishop frames the discussion there in terms of compensating for different priors. If $\tilde{\boldsymbol{\pi}}$ is defined by $\tilde{\pi}_i = \#\{t_i^n \in T | t_i^n = 1\}/N$, *i.e.* the proportion of training targets with class i, then compensating for that prior is what this dissertation calls *empirically weighted softmax*. This name is chosen as the weights are determined empirically from the training targets T.

Responsible Softmax uses the predictor Y as described in equation 4.2. As seen in equation 4.7, this is similar to adding a bias element in the fully connected layer before the softmax layer. Responsible softmax differentiates itself from using a bias in that the wights $\mu_i = \log \pi_i$ are not determined by backpropagation, but through dynamic responsibility. This also distinguishes RS from other weighted softmax methods in that the multiplicative weight is not on the output of a fully connected layer, but on the exponentiation of that output.

To make the discussion of responsible softmax more precise, consider the figure 4.1. Like the usual softmax layer, the responsible softmax layer follows a fully connected layer and precedes the loss. This means that RS is responsible for the final estimates passed to the loss function. This is the typical role of a softmax layer. This role occurs so frequently that many papers collectively call the last three layers softmax loss. For the purposes of this discussion, X is a $D \times N$ matrix and $A = W \cdot X$ is a $K \times N$ matrix, where N is the sample size (or batch size for algorithms like stochastic gradient descent).

The responsible softmax layer consists of two pieces. The first piece is an exponentiation layer, which ensures entries of the matrix F are positive. Here the exponentiation works on each term in A, as A is not a square matrix. Because $Y(F, \boldsymbol{p})$ is homogeneous in its arguments, it is practical to use $F = \sigma(A)$ in responsible softmax implementations, where $\sigma(A)$ is the softmax function applied to each element of A. However, for analysis reasons, $F = e^A$ will be used in the rest of this chapter.

Fully Connected Layer
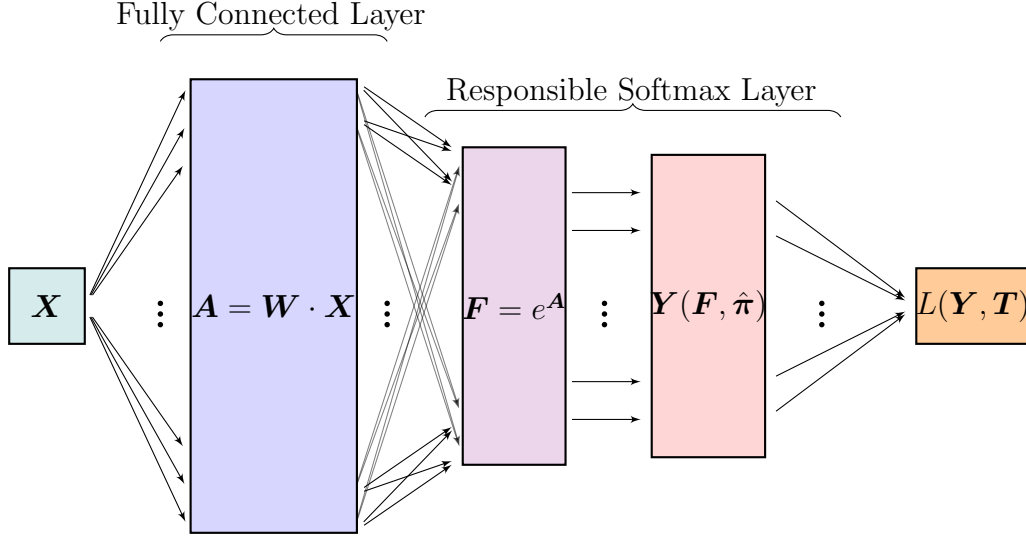
Figure 4.1: A responsible softmax layer consists of two pieces, an exponentiation layer to make sure that F has only positive entries, and an inference layer that uses dynamic responsibility to find an estimator of class probabilities $\hat{\pi}$. The responsible softmax layer follows a fully connected layer that takes the $D \times N$ dimensional input $\boldsymbol{X}$ and maps it to a $K \times N$ output $\boldsymbol{A}$. The responsible softmax layer precedes the loss layer.

The second piece of the responsible softmax layer is the inference layer, which uses dynamic responsibility to calculate $Y(F, \boldsymbol{p})$. Recall, from chapter 3 (sections 3.4 and 3.5) that algorithms 1 and 2 can have bad behavior for certain parameters F. If on any iteration there is some class i such that $\hat{\pi}_i = 0$, algorithm 1 will set $\hat{\pi}_i = 0$ for every iteration after that. On the other hand, algorithm 2 will often set some weight $\hat{\pi}_i < 0$ in similar situations. Of the two, it is easier to handle the problems with algorithm 1.

While it might seem that negative values for $\hat{\pi}_i$ might be less difficult to handle, it is often the case that a negative class weight for class i will cause $y_i^{(n)} < 0$ for some i. As $L(T, Y) = -\frac{1}{N} \sum_{i,n} t_i^{(n)} \log\left(y_i^{(n)}\right)$, having $y_i^{(n)} < 0$ for any i or n quickly causes trouble. On the other hand, the resolution for difficulties caused by dynamic responsibility algorithm 1, are much easier to solve.

Inspired by combination of the paper by Neal and Hinton [NH98], and the connection of dynamic responsibility to the EM-algorithm, consider the following modification to equation 4.2. For a given $C \in \mathbb{N}$, and an initial point $\boldsymbol{\pi}_0 \in S_K$, let $\boldsymbol{\pi}_C = R^C(F, \boldsymbol{\pi}_0)$ be the result of C iterations of $R_F(\boldsymbol{\pi})$ starting at π_0. Then define

$$Y_C = Y(F, \boldsymbol{\pi}_C) = \left(\frac{\pi_{C,k} f_k(x^{(n)})}{\sum_{i=1}^{K} \pi_{C,i} f_i(x^{(n)})} \right)_{k,n}. \tag{4.9}$$

The heuristic for this process, inspired by Neal and Hinton, suggests that for each training pass of the neural network an exact computation of $\hat{\boldsymbol{\pi}}$ is not necessary. Instead the network may 'take a few steps in the right direction' on each iteration. Starting with some initial $\boldsymbol{\pi}_0$, forward passes calculate $\boldsymbol{\pi}_C$ and then set $\boldsymbol{\pi}_0 = \boldsymbol{\pi}_C$ for the next forward pass. This does introduce a new hyper-parameter Cvc, but as shown in section 5.1, smaller values of C (*e.g.* $C = 1$) are very effective.

The general process for using responsible softmax starts with some reasonable C and $\boldsymbol{\pi}_0$, *e.g.* $\frac{1}{K}\mathbb{1}_K$ then iteratively applies the following two steps:

1. *Forward pass*: Receive F from previous layer. Calculate $\boldsymbol{\pi}_c = R_F^c(\boldsymbol{\pi}_0)$ for $c = 1, \ldots, C$ and $Y_C = Y(F, \boldsymbol{\pi}_C)$. Pass Y_C to the loss layer and store $\{\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_C\}$ for use in backward pass. Set $\boldsymbol{\pi}_0^{new} = \boldsymbol{\pi}_C$.

2. *Backward pass*: Receive $\frac{\partial L}{\partial Y_C}$ from the loss layer, and $\{\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_C\}$ from the forward pass. Use these to calculate $\frac{\partial Y_C}{\partial F}$, and pass this to the previous layer.

Algorithms 3 and 4 summarize the forward and backward steps respectively. As a general concept, a responsible softmax layer uses dynamic responsibility to mimic the expectation step of the EM-algorithm. Similarly, backpropagation in the neural net imitates the maximization step. The responsible softmax layer has several novelties in this regard.

Bishop's early book [Bis95, ch.6, equation 6.159], discusses how the bias of a fully connected layer is related to class probabilities. Under the assumption that p.d.f. of a sample x given the class label ℓ, $p(\boldsymbol{x}|\ell = i) = f_i(\boldsymbol{x}, \boldsymbol{\theta}_i)$ is in the exponential family for all i, then the following holds

$$\beta_i = A(\boldsymbol{\theta}_i) + \ln(\hat{\pi}_i), \tag{4.10}$$

where β_i is the i-th bias element and $A(\boldsymbol{\theta}_i)$ is a function of the parameters $\boldsymbol{\theta}_i$ specific to the realization of $f_i(\boldsymbol{x}, \boldsymbol{\theta}_i)$ as a member of the exponential family. This means that while $e^{\beta_i} \sim \hat{\pi}_i$, the class probabilities are not directly calculated. Using a responsible softmax layer, finding a direct estimator $\hat{\boldsymbol{\pi}}$ of $\boldsymbol{\pi}^*$ is almost surely guaranteed.

Another interesting characteristic presented by the use of a responsible softmax layer is the optimization of the log likelihood $\ell_F(\boldsymbol{\pi})$ as an added effect of the process. This presents a potential opportunity to add explainable structure to the neural network architecture. Doing so without breaking the convexity of $\ell_F(\boldsymbol{\pi})$ may prove to be difficult. Optimization of $\ell_F(\boldsymbol{\pi})$ by responsible softmax also implies interesting facts about F. Early on, F may have rank less than K, but converge for later

---

Algorithm 3 RS Forward Prediction Algorithm

Require: F a $K \times N$ matrix
Require: $\boldsymbol{\pi}_0^{new}$, C
  procedure ITERATION($F, \boldsymbol{\pi}_0, C$)   ▷ There are always C iterations
      $n \leftarrow 1$, $\pi_0 \leftarrow \boldsymbol{\pi}_0^{new}$
      $\boldsymbol{\pi}_n \leftarrow \mathrm{R}(F, \boldsymbol{\pi}_0)$
      $orbit \leftarrow \boldsymbol{\pi}_0, \boldsymbol{\pi}_1$
      while $n < C$ **do**
          $\boldsymbol{\pi}_{n+1} \leftarrow \mathrm{R}(F, \boldsymbol{\pi}_n)$
          $orbit \leftarrow \boldsymbol{\pi}_0, \ldots, \boldsymbol{\pi}_{n+1}$
          $n \leftarrow n + 1$
      end while
      $Y \leftarrow \mathrm{Y}(F, \boldsymbol{\pi}_C)$, $\boldsymbol{\pi}_0^{new} \leftarrow \boldsymbol{\pi}_C$
      return $orbits$, Y ▷ The variable $orbits$ is required for backpropagation
  end procedure

Table 4.1: Forward prediction algorithm for responsible softmax.

training iterations. This might allow for an increasing C in later training. The majority of these ideas are for later exploration.

Because of the difficulty in taking derivatives over high dimensional spaces, algorithm 4 is much more complicated than it seems. The important insight offered by algorithm 4 is that the first C iterates of $\boldsymbol{\pi}_0$ under iteration by R_F are required for backpropagation. This will be contrasted to the behavior of R_F near its fixed point in section 4.4.

4.3 Backpropagation and Responsibility

4.3.1 Computation of $\frac{\partial Y}{\partial F}$

Following the discussion in section 4.2, recall the definition

$$Y_i^j = \frac{\pi_{C,i} F_i^j}{\sum_{k=1}^{K} \pi_{C,k} F_k^j}$$

Where $F_i = \boldsymbol{\sigma}\{(W \cdot X)_i\}$ and $\boldsymbol{\pi}_C = R^C(F, \boldsymbol{\pi}_0)$, is C-th iterate of $\boldsymbol{\pi}_0$ under R_F.

In order to successfully do backpropagation, the neural network must calculate the gradient of the loss L with respect to the weights, W. As mentioned in section 2.4.3 this is done via the chain rule.

Algorithm 4 RS Backpropagation Algorithm

Require: F a $K \times N$ matrix
Require: $\nabla_Y L =: \frac{\partial L}{\partial Y}$, the gradient of the loss with respect to Y.
Require: $orbits = \boldsymbol{\pi}_0, \ldots, \boldsymbol{\pi}_C$
  procedure DERIVATION($F, \boldsymbol{\pi}_0, \ldots, \boldsymbol{\pi}_C$)
      $n \leftarrow 1$
      $\frac{\partial \boldsymbol{\pi}_C}{\partial F} \leftarrow \frac{\partial R}{\partial F}$
      while $n < C$ **do**
         $\frac{\partial \boldsymbol{\pi}_C}{\partial F} \leftarrow \frac{\partial R}{\partial F} + \frac{\partial R}{\partial \boldsymbol{\pi}}\big|_{\boldsymbol{\pi}_n} \cdot \frac{\partial \boldsymbol{\pi}_C}{\partial F}$
         $n \leftarrow n + 1$
      end while
      $\nabla_F Y \leftarrow \frac{\partial Y}{\partial F} + \frac{\partial Y}{\partial \boldsymbol{\pi}}\big|_{\boldsymbol{\pi}_C} \cdot \frac{\partial \boldsymbol{\pi}_C}{\partial F}$
      return $\nabla_F Y$
  end procedure

Table 4.2: Backward propagation algorithm for responsible softmax.

Thus to calculate

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial Y} \cdot \frac{\partial Y}{\partial F} \cdot \frac{\partial F}{\partial W}, \tag{4.11}$$

the neural network must first compute $\frac{\partial Y}{\partial F}$. To aid in computation of the derivative and gradient of Y appropriately, consider the bipartite directed graph in figure 4.2.
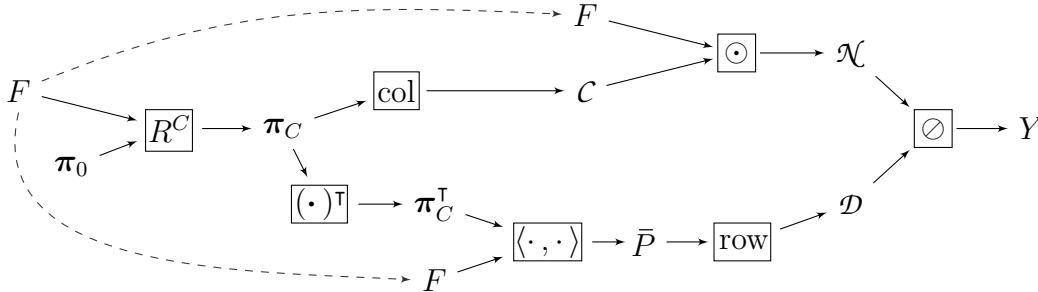
Figure 4.2: A directed bipartite graph for computing Y. Unboxed nodes represent variables, boxed nodes indicate functions, which perform computations on the variables. Dashed lines represent a repetition of the node F at the given locations.

This graph presents a very detailed view of computing Y. This will help when computing $\frac{\partial Y}{\partial F}$. Each boxed vertex represents a single step *intermediate function* computation in calculating $Y(F, \boldsymbol{p}_C)$. The unboxed vertices are the variable nodes. While the only variables that Y directly depends on are F and $\boldsymbol{p}_0$, there are several *intermediate variables* which are calculated along the way. To calculate the gradient of Y with respect to F, the gradients can be calculated for each intermediate function

with respect to the intermediate variables. Then these gradients may be appropriately combined to compute $\frac{\partial Y}{\partial F}$. This is very similar to the discussion in Deisenroth *et al.* [DFO20, example 5.14]. Many other resources also cover this material.

The method of breaking a single computation into several computations can become progressively granular to a point that the intermediate functions are the basic operations performed by a computer. When this is implemented numerically in a computer, either through software or hardware, the result is called *automatic differentiation* (AD). The reference by Baydin *et al.* [BPRS17] contains a recent survey of AD.

Automatic differentiation pertains to the current discussion not only because of its relation to figure 4.2, but also as an implementation discussion. Many software platforms include AD as a part of their machine learning and neural net implementations. As of release 2019b, this is also true of MATLAB software. An implementation of the responsible softmax layer in MATLAB code that uses AD can be found in appendix B. While this makes coding easier, it is possible for automatic differentiation to be slower than well implemented vectorized code to compute gradients. For this, in addition to academic reasons, it is a good idea to calculate $\frac{\partial Y}{\partial F}$ directly.

Table 4.3 summarizes the intermediate vertices from the computation graph in figure 4.2. The gradient of most intermediate functions will be calculated in section 4.5, but first section 4.4 will cover derivatives of R_F^C.

Name	Intermediate Variable Description	Name	Intermediate Function Description
$\mathcal{N}$	$\text{col}(\boldsymbol{p}_C) \odot F$	$\odot$	Hadamard product
$\mathcal{D}$	$\text{row}(\bar{P}) := \mathbb{1}_K \cdot \bar{P}$	$\oslash$	Hadamard division
$\bar{P}$	$\boldsymbol{\pi}_C^{\mathsf{T}} \cdot F$	$\langle \cdot, \cdot \rangle$	Appropriate Multiplication
$\boldsymbol{\pi}_C$	$R_F^C(\boldsymbol{\pi}_0)$	$\text{col}(\cdot)$	$(\cdot) \cdot \mathbb{1}_N^{\mathsf{T}}$, N columns, all of them the same.
Y	$\mathcal{N} \oslash \mathcal{D}$	$\text{row}(\cdot)$	$\mathbb{1}_K \cdot (\cdot)$, K rows all of them the same.
$\mathcal{C}$	$\text{col}(\boldsymbol{\pi}_C) := \boldsymbol{\pi}_C \cdot \mathbb{1}_N^{\mathsf{T}}$	$(\cdot)^{\mathsf{T}}$	Transpose of matrix or vector

Table 4.3: This table lists a brief description of the vertices found in the graph 4.2. The exact definition of each term while be discussed when computing the gradients in section 4.5.

The Hadamard product and division mentioned in table 4.3 are simply the entrywise product and division respectively, of two matrices having the same size.

4.4 Methods to compute DR

This section focuses on derivatives of the function $R_F(\boldsymbol{\pi})$. Because the goal will be to calculate the gradient $\frac{\partial L}{\partial F}$, this section first requires better exposition of the function R_F changes as F changes. To this end, define the space $M_{K,N}$ as the subset of full rank $K \times N$ matrices with positive entries. Then with S_K defined as in 2.23, let $\mathcal{M} = M_{K,N} \times S_K$. Define a map

$$R : \mathcal{M} \to S_K : R(F, \boldsymbol{\pi}) = R_F(\boldsymbol{\pi}). \tag{4.12}$$

Under this notation consider DR, the Frechét of R. For a given point $(F, \boldsymbol{\pi}) \in \mathcal{M}$, the map $DR(F, \boldsymbol{\pi})$ is a linear map between $T_{(F,\boldsymbol{\pi})}\mathcal{M}$ the tangent space to $\mathcal{M}$ at $(F, \boldsymbol{\pi})$ and $T_{R(F,\boldsymbol{\pi})}S_K$. This section refers to this operator as Frechét derivative and uses the notation $DR(F, \boldsymbol{\pi})$ or just DR when the point $\boldsymbol{\pi}$ is given in context.

Because $\mathcal{M}$ is a product manifold, DR is the sum of two linear operators

$$DR(F, \boldsymbol{\pi})[H, h] = D_F R(F, \boldsymbol{\pi})[H] + D_{\boldsymbol{\pi}} R(F, \boldsymbol{\pi})[h]$$

for $H \in TM_{K,N}$, $h \in TS_K$. In this case, define the maps $D_F R[H] := DR[H, 0]$ and $D_{\boldsymbol{\pi}} R[h] := DR[0, h]$ as the derivative of R holding $\boldsymbol{\pi}$ or F constant respectively.

To explicitly calculate $D_F R$ an $D_{\boldsymbol{\pi}} R$, let $P : \mathcal{M} \to \mathbb{R}^N$ be given by $P(F, \boldsymbol{\pi}) = \boldsymbol{\pi}^\intercal F$. Note that P is bilinear. Let log represent the component wise natural log, and define

$$\ell : \mathcal{M} \to \mathbb{R} : \ell(F, \boldsymbol{\pi}) = \frac{1}{N} \mathbb{1}_N \cdot \log \circ P(F, \boldsymbol{\pi}).$$

Then for fixed F, $\ell(F, \boldsymbol{\pi}) = \ell_F(\boldsymbol{\pi})$ where $\ell_F(\boldsymbol{\pi})$ is the log likelihood as defined in chapter 3. This new definition will allow a closer look at the derivatives of $R_F(\boldsymbol{\pi})$, and examine the behavior of the level sets described by $R_F(\boldsymbol{\pi}) - \boldsymbol{\pi} = 0$ for changing F.

As before, define $D_F \ell$ and $D_{\boldsymbol{\pi}} \ell$ as the portions of $D\ell$ that act on $TM_{K,N}$ and TS_K respectively. Further, by using the standard inner product on S_K, and the Frobenius inner product $\langle U, V \rangle = \text{tr}(U^\intercal V)$ on $M_{K,N}$ define $\nabla_F \ell$ and $\nabla_{\boldsymbol{\pi}} \ell$ by

$$D_F \ell(G)[H] = \text{tr}(H^\intercal \nabla_F \ell(G))$$

and

$$D_{\boldsymbol{\pi}}(\boldsymbol{\pi})[\boldsymbol{h}] = \boldsymbol{h}^{\mathsf{T}} \nabla_{\boldsymbol{\pi}} \ell(\boldsymbol{\pi}) \tag{4.13}$$

Under this notation,

$$R(\boldsymbol{\pi}, F) = \mathrm{diag}(\boldsymbol{\pi}) \cdot \nabla_{\boldsymbol{\pi}} \ell(\boldsymbol{\pi}, F) \tag{4.14}$$

4.4.1 Calculating DR on S_K

Now calculate the partial derivatives of R to better understand DR focusing first on $D_{\boldsymbol{\pi}} R$. Note that equation (4.15) uses the standard coordinate system for $\mathbb{R}^K$ and so does not use the full strength of the Frechét derivative. However, using a coordinate system clarifies computation of the Frechét derivative $D_{\boldsymbol{\pi}} R$ in this case.

Corollary 3.3 gives

$$R(F, \boldsymbol{\pi}) = \left(\frac{\partial \ell}{\partial \pi_i} \cdot \pi_i \right)_{1 \leq i \leq K}.$$

if $r_j = \frac{\partial \ell}{\partial \pi_j} \cdot pi_j$,

$$
\begin{aligned}
\frac{\partial r_j}{\partial \pi_i} &= \frac{\partial}{\partial \pi_i} \left(\frac{\partial \ell}{\partial \pi_j} \cdot \pi_j \right) \\
&= \frac{\partial^2 \ell}{\partial \pi_i \partial \pi_j} \pi_j + \frac{\partial \ell}{\partial \pi_j} \frac{\partial \pi_i}{\partial \pi_j} \\
&= \frac{\partial^2 \ell}{\partial \pi_i \partial \pi_j} \pi_j + \delta_{ij} \frac{\partial \ell}{\partial \pi_j}
\end{aligned}
\tag{4.15}
$$

where δ_{ij} is the Kronecker delta function.

From equation (4.15) it follows that

$$D_{\boldsymbol{\pi}} R(\boldsymbol{\pi}) = \mathrm{diag}(\boldsymbol{\pi}) \cdot \nabla^2 \ell(\boldsymbol{\pi}) + \mathrm{diag}(\nabla \ell(\boldsymbol{\pi})). \tag{4.16}$$

In equation (4.15), it is assumed that the matrix F is held constant.

By theorem 3.8 for each $F \in M_{K,N}$ there is a unique fixed point $\hat{\boldsymbol{\pi}}_F$ such that

$$R(F, \hat{\boldsymbol{\pi}}_F) = \hat{\boldsymbol{\pi}}_F. \tag{4.17}$$

Since the matrix F is clear in the previous equation, it is omitted except when emphasis of the dependence on F is required. Because this fixed point may be obtained via iteration of R_F, it would

be expected that $D_{\boldsymbol{\pi}}R_F$ be a linear contraction mapping. For $\boldsymbol{\pi} \in \text{Int}(S_K)$ sufficiently close to $\hat{\boldsymbol{\pi}}_F$, this is true. As a consequence, equation (4.16) may be used to show that the point $\hat{\boldsymbol{\pi}}_F$ depends continuously differentiably (C^1) on F. Proving this becomes easier lemma 4.2 which comes from [Ryc20].

Lemma 4.1 (Local Contraction at critical fixed points). *Let $T : M \subset \mathbb{R}^m \to M$ be a continuously differentiable mapping that describes a discrete (semi-)dynamical system. Let $V : M \to \mathbb{R}$ be a twice differentiable function that acts as a strict Lyapunov function for the system described by T. Further suppose that $\boldsymbol{x} \in M$ is a critical fixed point of (T, V) and $\nabla^2 V(\boldsymbol{x})$ is positive definite. Then in some neighborhood $U \subset M$ of $\boldsymbol{x}$, the linear map $DT(\boldsymbol{x}) : TM \to TM$ is a contraction mapping, i.e. all the eigenvalues of $DT(\boldsymbol{x})$ have norm less than one.*

Proof. First, without loss of generality, it may be assumed that $\boldsymbol{x} = \boldsymbol{0}$, and $V(x) = 0$. Then because V is twice continuously differentiable, it has a second order Taylor approximation near $\boldsymbol{0}$,

$$V(\boldsymbol{m}) \approx V(\boldsymbol{0}) + DV(\boldsymbol{0})[\boldsymbol{m}] + \frac{1}{2}D^2V(\boldsymbol{0})[\boldsymbol{m}, \boldsymbol{m}]. \tag{4.18}$$

However, by assumption $V(\boldsymbol{0}) = DV(\boldsymbol{0}) = 0$, so $V(\boldsymbol{m}) \approx \frac{1}{2}D^2V(\boldsymbol{0})[\boldsymbol{m}, \boldsymbol{m}]$.

Now since V is a strict Lyapunov equation for T, $\dot{V}(\boldsymbol{m}) := V(\boldsymbol{m}) - V(T(\boldsymbol{m})) > 0$ unless $T(\boldsymbol{m}) = \boldsymbol{m}$. Then if $H := \nabla^2 V(\boldsymbol{0})$, and $S := DT(\boldsymbol{0})$ it follows that $\nabla^2 \dot{V}(0) = H - S^*HS$ where $(\cdot)^*$ represents the usual conjugate transpose.

Now let $\boldsymbol{y}$ be any eigenvector for S, $S \cdot \boldsymbol{y} = \lambda \boldsymbol{y}$. Then

$$0 < \boldsymbol{y}^* \cdot H \cdot \boldsymbol{y} - \boldsymbol{y}^* \cdot S^* \cdot H \cdot S \cdot \boldsymbol{y} \tag{4.19}$$

$$= \boldsymbol{y}^* \cdot H \cdot \boldsymbol{y} - |\lambda|^2 \boldsymbol{y}^* \cdot H \cdot \boldsymbol{y} \tag{4.20}$$

$$= (1 - |\lambda|^2)\boldsymbol{y}^* \cdot H \cdot \boldsymbol{y}. \tag{4.21}$$

Since H is positive definite by assumption, it follows from (4.21) that $1 - |\lambda|^2 > 0$. and $|\lambda|^2 < 1$. Since $\boldsymbol{y}$ was an arbitrary eigenvector of $DT(\boldsymbol{0})$, this holds for all eigenvalues of $DT(\boldsymbol{m})$, with $\boldsymbol{m}$ near $\boldsymbol{0}$ where equation (4.18) holds.

$\square$

Corollary 4.2. *Let $V_K := \{\boldsymbol{x} \in \mathbb{R}^K | \langle \boldsymbol{x}, \mathbb{1}_K \rangle = 0\}$, then define $g : M_{K,N} \times V_K \to V_K$ by $g(F, \boldsymbol{h}) = R(F, \hat{\boldsymbol{\pi}}_F + \boldsymbol{h}) - (\hat{\boldsymbol{\pi}}_F + \boldsymbol{h})$. Then for any given $F_0 \in M_{K,N}$, $g(F_0, \boldsymbol{0}) = \boldsymbol{0}$ is the only fixed point of*

$g(F_0, \cdot)$. *Further, if $\hat{\boldsymbol{\pi}}_F \in \mathrm{Int}(S_K)$, then the linear mapping given by $D_{\boldsymbol{h}}g(\mathbf{0})$ is invertible on some set $U \subset S_K$ containing $\hat{\boldsymbol{\pi}}_F$.*

Proof. First note that $g(F, \mathbf{0}) = \mathbf{0}$ is equivalent to the statement that $R(F, \hat{\boldsymbol{\pi}}_F) = \hat{\boldsymbol{\pi}}_F$. Thus by theorem 3.8, for any given F_0, $\mathbf{0}$ must be the only fixed point of $g(F_0, \cdot)$.

Now consider the derivative $D_{\boldsymbol{h}}g(\mathbf{0}) = DR(\hat{\boldsymbol{\pi}}) - I_K$. This is invertible iff no eigenvalue of $DR(\hat{\boldsymbol{\pi}})$ has norm 1. But applying lemma 4.1 to $(R_F, -\ell_F)$ at $\hat{\boldsymbol{\pi}}$ shows that $DR_F(\hat{\boldsymbol{\pi}})$ is a local contraction, and so it has no eigenvalues of norm greater than one. $\qquad\square$

Now apply the implicit function theorem [KPC02, theorem 3.2.4]. Given an $F_0 \in M_{K \times N}$ such that $\hat{\boldsymbol{\pi}}_{F_0} \in \mathrm{Int}(S_K)$, there is an open neighborhood $U \in M_{K \times N}$ of F_0 such that $\hat{\boldsymbol{\pi}}_F$ is a continuously differentiable function of F for all $F \in U$. In this situation, write

$$D\hat{\boldsymbol{\pi}}_F = D_{\boldsymbol{\pi}}R \cdot D\hat{\boldsymbol{\pi}}_F + D_F R$$

which gives

$$D\hat{\boldsymbol{\pi}}_F = (I - D_{\boldsymbol{\pi}}R)^{-1} \cdot D_F R \tag{4.22}$$

where I_K is the $K \times K$ identity matrix. Note that at $\hat{\boldsymbol{\pi}}_F$, $I - D_{\boldsymbol{\pi}}R = -P \cdot H$, which will be a positive definite matrix. Because $R_F(\boldsymbol{\pi})$ is actually smooth on S_K, with a little extra work it is possible to show that $\hat{\boldsymbol{\pi}}_F$ varies smoothly with F. Fortunately, having a continuous derivative is sufficient. Taken together, the previous comments prove theorem 4.3

Theorem 4.3. If $F_0 \in M_{K \times N}$ has $R(F_0, \hat{\boldsymbol{\pi}}) = \hat{\boldsymbol{\pi}}$ for some $\hat{\boldsymbol{\pi}} \in \mathrm{Int}(S_K)$, then there is some neighborhood $U \in M_{K \times N}$ such that $\hat{\boldsymbol{\pi}} = \hat{\boldsymbol{\pi}}(F_0)$ is a continuously differentiable function of $F \in U$ and

$$D\hat{\boldsymbol{\pi}}(F) = (I - D_{\boldsymbol{\pi}}R(\hat{\boldsymbol{\pi}}(F)))^{-1} \cdot D_F R(F)$$

as in equation (4.22).

The process described in algorithm 4 has equation 4.22 as a limiting behavior. For example, if $\boldsymbol{\pi}_0 \approx \hat{\boldsymbol{\pi}}$, then for all $c \geq 1$, $\boldsymbol{\pi}_c \approx \boldsymbol{\pi}_C$ and

$$\frac{\partial \boldsymbol{\pi}_C}{\partial F} = \frac{\partial R}{\partial F} + \frac{\partial R}{\partial \boldsymbol{\pi}}\bigg|_{\boldsymbol{\pi}_{C-1}} \cdot \frac{\partial \boldsymbol{\pi}_{C-1}}{\partial F} \approx \frac{\partial R}{\partial F} + \frac{\partial R}{\partial \boldsymbol{\pi}}\bigg|_{\boldsymbol{\pi}_C} \cdot \frac{\partial \boldsymbol{\pi}_C}{\partial F}.$$

This leads to the same conclusion as 4.22. On the other hand, $\frac{\partial \pi_0}{\partial F} = 0$, so $\frac{\partial \pi_1}{\partial F} = \frac{\partial R}{\partial F}$ and then

$$\frac{\partial \pi_2}{\partial F} \approx \frac{\partial R}{\partial F} + \left.\frac{\partial R}{\partial \pi}\right|_{\pi_1} \cdot \frac{\partial R}{\partial F},$$

but this is just indicative of the fact that $I + A \approx (I - A)^{-1}$ when $\|A\| < 1$, i.e. $\sum_{j=0}^{\infty} A^j$ is the Nuemann series for the operator $(I - A)^{-1}$.

4.4.2 Calculating DR on Parameter Matrices

Next, the focus turns to calculating $D_F R$. As before, it is illustrative to consider partial derivatives. As in equation (4.15), $r_j = \frac{\partial \ell}{\partial \pi_j} \cdot \pi_j$ represents the j-th coordinate function of $R(F, \boldsymbol{\pi})$.

$$
\begin{aligned}
\frac{\partial r_j}{\partial f_{rs}} &= \frac{\partial}{\partial f_{rs}} \left(\frac{\partial \ell}{\partial \pi_j} \pi_j \right) \\
&= \frac{\pi_j}{N} \frac{\partial}{\partial f_{rs}} \left(\sum_{n=1}^{N} \frac{f_{jn}}{\sum_{k=1}^{K} f_{kn} \pi_k} \right) \\
&= \frac{\pi_j}{N} \sum_{n=1}^{N} \delta_{ns} \left(\frac{\delta_{jr}}{\sum_{k=1}^{K} f_{kn} \pi_k} - \frac{f_{js} \pi_r}{\left(\sum_{k=1}^{K} f_{kn} \pi_k \right)^2} \right) \\
&= \frac{\pi_j}{N} \left(\frac{\delta_{jr}}{\sum_{k=1}^{K} f_{ks} \pi_k} - \frac{f_{js} \pi_r}{\left(\sum_{k=1}^{K} f_{ks} \pi_k \right)^2} \right).
\end{aligned}
\tag{4.23}
$$

Unlike calculation of $\frac{\partial r_j}{\partial \pi_i}$, equation (4.23) does note readily translate to an easy formula for $D_F R$. This is because F is a matrix, and the derivative of a vector valued function with respect to a matrix is a Tensor. This situation may be handled through careful use of the column stacking function $\text{vec}(F)$ and the commutative diagram (2.9).

The benefit of this is that the calculation of the partials in 4.23 is also the calculation of the Jacobian of r_j. Put more precisely, if the Fréchet derivative of r_j is represented by Dr_j, and the Jacobian is represented by $\frac{\partial r_j}{\partial F} = \nabla r_j(F)$, then

$$Dr_j(F)[H] = \text{tr}\left(\nabla r_j(F) \cdot H^{\intercal} \right) \tag{4.24}$$

where $\text{tr}(\cdot)$ represents the trace map.

Now by the relationship described in the commutative diagram (2.9),

$$\text{tr}\left(\nabla r_j(F) \cdot H^\intercal\right) = \langle \text{vec}(\nabla r_j(F)), \text{vec}(H) \rangle \tag{4.25}$$

Let $A(F)$ be the $K \times KN$ matrix given by

$$A(F) = \begin{pmatrix} \text{vec}(\nabla r_1(F))^\intercal \\ \text{vec}(\nabla r_2(F))^\intercal \\ \vdots \\ \text{vec}(\nabla r_K(F))^\intercal \end{pmatrix}. \tag{4.26}$$

Then the column vector given by $A(F) \cdot \text{vec}(H) = (\langle \text{vec}(\nabla r_j(F)), \text{vec}(H) \rangle)_{1 \leq j \leq K}^{intercal}$ is clearly obtained by $Dr_j(F)$ acting on H for $j = 1, \ldots, K$. Thus

$$D_F R(F)[H] = A(F) \cdot \text{vec}(H), \tag{4.27}$$

which gives a precise form for the term $D_F R$ in equation 4.22.

4.5 Using Derivatives of R to compute $\frac{\partial L}{\partial Y}$

This section seeks to describe the process by which the gradient $\frac{\partial L}{\partial F}$ may be calculated. In consideration of equation (4.11), if the backpropagation algorithm is given $\frac{\partial L}{\partial Y}$, then the gradient $\frac{\partial Y}{\partial F}$ must be calculated. More accurately, lemma 2.1 shows that the adjoint operator D^*Y of the derivative DY is required to compute $\frac{\partial L}{\partial F}$.

Using the computation graph for Y from figure 4.2 allows the calculation of this adjoint operator to be broken up into several small pieces. This section will be organized by calculation of the adjoint on each of the pieces. There will also be some more general lemmas proven where necessary.

At each step, lemma 2.1 is used implicitly, where each calculation is given by passing back gradient information received from previous calculations steps, as is typical with backpropagation. Also, both π_C and F are used in multiple computation steps. The correct thing to do in this case is add the gradients from each computation to get the final gradients $\frac{\partial L}{\partial \pi_C}$ and $\frac{\partial L}{\partial F}$. In the cases where the gradient calculate is only a summand of the total gradient it will be noted with an additional subscript, *e.g.* $\left(\frac{\partial L}{\partial F}\right)_{\mathcal{N}} = D_F^* \mathcal{N}[\frac{\partial L}{\partial \mathcal{N}}]$.

The following two lemmas facilitate the first calculation.

Lemma 4.4 (Adjoint of Hadamard product)**.** *The linear function $f_A(X) = A \odot X$ is self adjoint on the space of $K \times N$ matrices.*

Proof. Because the Hadamard product is a linear operator, it is its own derivative *i.e.* $Df_A = f_A$. While it can be shown that this operator is self adjoint through careful computation, it will be more powerful to use the vec operator as described in the commutative diagram (2.9).

Under the diffeomorphism vec, consider the function $g_A(X) = \text{vec}^{-1}(f(\text{vec}(A), \text{vec}(X))) = \text{vec}^{-1}(\text{diag}(\text{vec}(A)$ vec$(X))$. In this representation, the Hadamard product with A is given by multiplication by a diagonal matrix, which is clearly self adjoint as $\text{diag}(\text{vec}(A))^\intercal = \text{diag}(\text{vec}(A))$. Hence $g_A(X)$ is self adjoint. Since $g_A(X)$ is not only similar, but actually equal to f_A, f_A must also be self adjoint. □

Lemma 4.4 applies directly to Hadamard division $\oslash$, because for two matrices X, Y of the same size $X \oslash Y = X \odot \overline{Y}$, where $Y = (y_{ij})_{i,j}$ $\overline{Y} = (y_{ij}^{-1})_{i,j}$. This definition necessarily requires that $y_{ij} \neq 0$. This relationship between Hadamard product and division is the key observation in lemma 4.5 below.

Lemma 4.5 (Adjoint of Hadamard Division)**.** *The function $X \oslash Y$ has derivative $D(X \oslash Y)[dX, dY] = dX \oslash Y - dY \odot X \oslash (Y \odot Y)$, under the assumption that $\frac{\partial X}{\partial Y} = \frac{\partial Y}{\partial X} = 0$. Because $\odot$ is self adjoint, so is $D(X \oslash Y)$.*

Proof. Since Hadamard division works elementwise, the derivative may be calculated by looking at partial derivatives. Then

$$d\left(\frac{x_{ij}}{y_{ij}}\right) = \frac{1}{y_{ij}}dx_{ij} - \frac{x_{ij}}{y_{ij}^2}dy_{ij}$$

which gives the correct formula for the derivative.

Given the relationship between Hadamard product and Hadamard division the derivative formula may be rewritten as $D(X \odot \overline{Y})[dX, dY] = dX \odot \overline{Y} - dY \odot X \odot \overline{Y} \odot \overline{Y}$. In this form, lemma 4.4 clearly shows that the derivative operator is self adjoint. □

These two lemmas carry most of the weight in calculating of the first two adjoint derivatives. Since the gradient in backpropagation are calculated in the opposite direction, so the first two adjoints to be calculated are the rightmost calculations done in the computation graph 4.2.

Calculation 4.1 (Adjoint #1)**.** So consider the equation $Y = \mathcal{N} \oslash \mathcal{D}$. By lemma 4.5,

$$DY[d\mathcal{N}, d\mathcal{D}] = d\mathcal{N} \oslash \mathcal{D} - d\mathcal{D} \odot \mathcal{N} \oslash (\mathcal{D} \odot \mathcal{D}). \tag{4.28}$$

Let $D_{\mathcal{N}}Y[d\mathcal{N}] = d\mathcal{N} \oslash \mathcal{D}$ and $D_{\mathcal{D}}Y[d\mathcal{D}] = -d\mathcal{D} \odot \mathcal{N} \oslash (\mathcal{D} \odot \mathcal{D})$. By lemmas 4.4 and 4.5, it follows that if $dY := \frac{\partial L}{\partial Y}$ then

$$\frac{\partial L}{\partial \mathcal{N}} = D_{\mathcal{N}}^*Y[dY] = dY \oslash \mathcal{D} = dY \odot Y \oslash \mathcal{N} \tag{4.29}$$

$$\frac{\partial L}{\partial \mathcal{D}} = D_{\mathcal{D}}^*Y[dY] = -dY \odot \mathcal{N} \oslash (\mathcal{D} \odot \mathcal{D}) = -dY \odot Y \oslash \mathcal{D} \tag{4.30}$$

Calculation 4.2 (Adjoint #2)**.** Now consider the equation $\mathcal{N} = F \odot C$. Following a similar pattern as calculation 4.1, write

$$D\mathcal{N}[dF, dC] = dF \odot C + F \odot dC. \tag{4.31}$$

Define $D_F\mathcal{N}[dF] = dF \odot C$ and $D_C\mathcal{N}[dC] = F \odot dC$. Then using lemma 4.4, and letting $d\mathcal{N} := \frac{\partial L}{\partial \mathcal{N}}$ gives

$$\left(\frac{\partial L}{\partial F}\right)_{\mathcal{N}} = D_F^*\mathcal{N}[d\mathcal{N}] = d\mathcal{N} \odot C \tag{4.32}$$

$$\frac{\partial L}{\partial C} = D_C^*\mathcal{N}[d\mathcal{N}] = F \odot d\mathcal{N} \tag{4.33}$$

The next two calculations are similar to each other in that they both create a rank 1 matrix out of a row or column vector so that the Hadamard product (or division) may be used. In programming languages such as MATLAB, an operation such as $\boldsymbol{x} \odot A$, with $\boldsymbol{x}$ an $n \times 1$ column vector and A an $n \times m$ matrix (for some $n, m \in \mathbb{N}$), is interpreted as taking the Hadamard product of $\boldsymbol{x}$ with each column of A. To create precision in this document and still represent the same operation, the col and row operators must be used.

The col and row operators create rank 1 $K \times N$ matrix from a column or row vector respectively. This is done by copying the given vector an appropriate amount of times. For example, if $\boldsymbol{x} \in \mathbb{R}^K$ is a $K \times 1$ column vector, then $\text{col}(\boldsymbol{x}) := \boldsymbol{x} \cdot \mathbb{1}_N^{\mathsf{T}}$ is the $K \times N$ matrix with N columns each equal to $\boldsymbol{x}$. In this manner, the Hadamard product of a vector and a matrix can be well defined both mathematically and in programming.

Of course, this also means that the derivative and adjoint derivative of these operators must be calculated. Fortunately it is clear that both of these operators are linear, so they are their own derivative. The adjoint derivatives are also as expected, *i.e.* the adjoint is calculated through multiplication by the transpose of an appropriate vector. These observations are made robust in the following lemma.

Lemma 4.6 (row and col Operators). *Define* $\mathrm{col} : \mathbb{R}^K \rightarrow M_{K \times N}$ *by* $\mathrm{col}(\boldsymbol{x}) = \boldsymbol{x} \cdot \mathbb{1}_N^\mathsf{T}$. *Similarly, define* $\mathrm{row} : \mathbb{R}^N \rightarrow M_{K \times N}$ *by* $\mathrm{row}(y) = \mathbb{1}_K \cdot \boldsymbol{y}^\mathsf{T}$. *Then for* $U \in TM_{K \times N}$, $D^* \mathrm{col}[U] = U \cdot \mathbb{1}_N$ *and* $D^* \mathrm{row}[U] = (\mathbb{1}_K^\mathsf{T} \cdot U)^\mathsf{T}$.

Proof. As these two operators are so close, it suffices to show this only for $D^*\mathrm{col}$. First it is clear that col is linear, so $D\,\mathrm{col} = \mathrm{col}$. For two vectors $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^N$, their dot product is characterized by $\mathrm{tr}(\boldsymbol{a} \cdot \boldsymbol{b}^\mathsf{T}) = \boldsymbol{b}^\mathsf{T} \cdot \boldsymbol{a} = \langle \boldsymbol{a}, \boldsymbol{b} \rangle$. Then for $\boldsymbol{h} \in T\mathbb{R}^K$ and $U \in TM_{K \times N}$

$$\mathrm{tr}((\boldsymbol{h} \cdot \mathbb{1}_N^\mathsf{T})^\mathsf{T} \cdot U) = \mathrm{tr}(\mathbb{1}_N \cdot \boldsymbol{h}^\mathsf{T} \cdot U) = \boldsymbol{h}^\mathsf{T} \cdot U \cdot \mathbb{1}_N = \langle \boldsymbol{h}, U \cdot \mathbb{1}_N \rangle. \tag{4.34}$$

Which gives the lemma for col. The proof for row is almost exactly the same, *mutatis mutandis.* □

Calculation 4.3 (Adjoint #3 & #4). Consider the equation $\mathcal{C} = \mathrm{col}(\boldsymbol{\pi}_C)$. Since lemma 4.6 lays all the groundwork, let $d\mathcal{C} = \frac{\partial L}{\partial \mathcal{C}}$. Then

$$\left(\frac{\partial L}{\partial \boldsymbol{\pi}_C}\right)_\mathcal{C} = D^*\mathcal{C}[d\mathcal{C}] = d\mathcal{C} \cdot \mathbb{1}_N. \tag{4.35}$$

For the equation $\mathcal{D} = \mathrm{row}(\bar{P})$, a bit more care must be used. Since $\bar{P}$ is a row vector (as shown in table 4.3), the formula given in 4.6 must be changed by taking the transpose. Thus if $d\mathcal{D} = \frac{\partial L}{\partial \mathcal{D}}$ then

$$\frac{\partial L}{\partial \bar{P}} = D^*\mathcal{D}[d\mathcal{D}] = \mathbb{1}_K^\mathsf{T} \cdot d\mathcal{D}. \tag{4.36}$$

Calculation 4.4 (Adjoint #5). Now consider the computation $\bar{P} = \boldsymbol{\pi}_C^\mathsf{T} \cdot F$. Remember here that $\bar{P}$ is a $1 \times N$ row vector. Clearly multiplication of a vector and matrix is linear. Thus

$$D\bar{P}[dF, d\boldsymbol{\pi}_C] = d\boldsymbol{\pi}_C^\mathsf{T} \cdot F + \boldsymbol{\pi}_C^\mathsf{T} \cdot dF. \tag{4.37}$$

Defining $D_F\bar{P}[dF] = \boldsymbol{\pi}_C^\mathsf{T} \cdot dF$ and $D_{\boldsymbol{\pi}_C}\bar{P}[d\boldsymbol{\pi}_C] = d\boldsymbol{\pi}_C^\mathsf{T} \cdot F$ shows that

$$D_F^*\bar{P}[d\bar{P}] = \boldsymbol{\pi}_c \cdot d\bar{P} \tag{4.38}$$

$$D_{\boldsymbol{\pi}_C}^*\bar{P}[d\bar{P}] = F \cdot d\bar{P}^\mathsf{T}. \tag{4.39}$$

Then if $d\bar{P} = \frac{\partial L}{\partial \bar{P}}$ it follows that

$$\left(\frac{\partial L}{\partial F}\right)_{\bar{P}} = \boldsymbol{\pi}_c \cdot d\bar{P} \tag{4.40}$$

$$\left(\frac{\partial L}{\partial \boldsymbol{\pi}_C}\right)_{\bar{P}} = F \cdot d\bar{P}^{\intercal}. \tag{4.41}$$

Note that because $\boldsymbol{\pi}_C$ is involved in two calculations, summing (4.35) and (4.41) gives

$$\frac{\partial L}{\partial \boldsymbol{\pi}_C} = \left(\frac{\partial L}{\partial \boldsymbol{\pi}_C}\right)_C + \left(\frac{\partial L}{\partial \boldsymbol{\pi}_C}\right)_{\bar{P}}. \tag{4.42}$$

Finally, the adjoint with respect to the calculation $\boldsymbol{\pi}_C = R^C(F, \boldsymbol{\pi}_0)$ must be determined. Here the iterative nature of dynamic responsibility allows for some simplification. Though as shown in section 4.4, the derivatives of $R(F, \boldsymbol{\pi})$ are not simple. Fortunately the care taken in that section will aid in computing adjoints. The association of $D_{\boldsymbol{\pi}}R$ and D_FR with matrices particularly aids the adjoint calculation because the adjoint of a real matrix operator is the transpose of that matrix.

Calculation 4.5 (Adjoint #6). This computation is different from the previous ones in that it is actually working with C equations of the form $\boldsymbol{\pi}_i = R(F, \boldsymbol{\pi}_{i-1})$ for $i = 1, \ldots, C$. This means that the final gradient is the sum of several smaller gradients, $\left(\frac{\partial L}{\partial F}\right)_R = \sum_{i=1}^{C} \left(\frac{\partial L}{\partial F}\right)_{\boldsymbol{\pi}_i}$. This means that the computation must come in the form of an iterative algorithm, which is summarized in algorithm 5.

Because this computation is iterative, it suffices to show the calculation for a single iteration. Recalling equations (4.16) and (4.27), calculation of the adjoints gives

$$D_{\boldsymbol{\pi}}^* R(F, \pi_n)[d\boldsymbol{\pi}_n] = \left(\nabla^2 \ell(\boldsymbol{\pi}) \cdot \text{diag}(\boldsymbol{\pi}) + \text{diag}(\nabla \ell(\boldsymbol{\pi}))\right) \cdot d\boldsymbol{\pi}_n \tag{4.43}$$

$$D_F^* R(F, \boldsymbol{\pi}_n)[d\boldsymbol{\pi}_n] = \text{vec}^{-1}\left(A(F)^{\intercal} \cdot d\boldsymbol{\pi}_n\right). \tag{4.44}$$

Then define $\frac{\partial L}{\partial \boldsymbol{\pi}_C}$ as in equation (4.42), and set

$$d\boldsymbol{\pi}_{n-1} = D_{\boldsymbol{\pi}}^* R(F, \pi_n)[d\boldsymbol{\pi}_n] \quad n = 2, \ldots, C \tag{4.45}$$

$$\left(\frac{\partial L}{\partial F}\right)_R = \sum_{n=1}^{C} D_F^* R(F, \boldsymbol{\pi}_n)[d\boldsymbol{\pi}_n]. \tag{4.46}$$

Algorithm 5 Gradient Iteration Algorithm

Require: F a $K \times N$ matrix
Require: $orbit = (\boldsymbol{\pi}_0, \boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_C)$, $d\boldsymbol{\pi}_C$ $\triangleright$ $d\boldsymbol{\pi}_C$ is as defined in (4.42)

  procedure ITERATION(F, $orbits$, $\boldsymbol{\pi}_C$)

     $n \leftarrow C$

     $d\boldsymbol{\pi} \leftarrow d\boldsymbol{\pi}_C$

     $dF \leftarrow \mathbf{0}$

     while $n \geq 0$ **do**

        $dF \leftarrow D_F^* R(F, \boldsymbol{\pi}_n)[d\boldsymbol{\pi}] + dF$ $\triangleright$ Use (4.44)

        $d\boldsymbol{\pi} \leftarrow D_{\boldsymbol{\pi}}^* R(F, \pi_n)[d\boldsymbol{\pi}]$ $\triangleright$ Use (4.43)

        $n \leftarrow n - 1$

     end while

     return dF $\triangleright$ $d\boldsymbol{\pi}$ at this point would represent $\frac{\partial L}{\partial \boldsymbol{\pi}_0}$

  end procedure

Table 4.4: Computing gradients with backpropagation, iterative portion

Given the calculations Adjoint 1-6, use equations (4.32), (4.40), and (4.46) to get

$$\frac{\partial L}{\partial F} = \left(\frac{\partial L}{\partial F} \right)_{\mathcal{N}} + \left(\frac{\partial L}{\partial F} \right)_{\bar{P}} + \left(\frac{\partial L}{\partial F} \right)_{R}. \tag{4.47}$$

Chapter 5

Applied Examples

Having described the responsible softmax layer, this dissertation now focuses on performance of that layer comparison of this layer to standard techniques. In doing so, the analysis rests on two aspects, running time and measures of correctness. For measures of correct classification, many methods rely on metrics such as precision and accuracy. For multiclass problems, and for training with imbalanced data, such measures can be uninformative or deceptive in their approach. This chapter discusses some of the more common approaches and uses a few of them to evaluate how the responsible softmax layer affects classification.

In the area of running time, convergence of dynamic responsibility to a stable point greatly affects any additional compute time. Lemma 3.4 and the discussion from chapter 4 about differentiation of the responsibility operator help immensely in understanding how the distance $a_n := \|R(F, \boldsymbol{\pi}_n) - \boldsymbol{\pi}_n\|$ changes as $n \to \infty$. The hessian of $\ell(F, \boldsymbol{\pi})$ with respect to $\boldsymbol{p}$ is closely related to the matrix F. Thus iteration of $R(F, \boldsymbol{\pi})$ could reasonably converge like $e^{-\lambda}$, where $\lambda = \max \operatorname{svd}(F)$ is the largest of singular values of F. Empirically this is the case, as discussed in section 5.1.

When measuring correct classification, comparison of models to reasonable baselines provides intuition and discernment. Two base models for comparison to responsible softmax are the standard softmax layer, and use of appropriate weights to match priors on training labels. Using weights to match the mixture proportions may be accomplished through several methods. In this dissertation, the model that will be used is a responsible softmax layer with a fixed $\boldsymbol{\pi}_0$. This will be discussed further in section 5.2.

In training several different neural nets, numerous settings produce diverse effects on final performance. Initialization of weights and hyperparameter selection provide good changes to study such effects. This means running numerical experiments involving several mostly identical neural nets. Since the responsible softmax layer introduces a new hyperparameter C, this means testing layers with different values of C. As will be seen in sections 5.1 and 5.5, smaller values of C are generally better.

Classification performance of a given method also varies greatly among data sets. Both synthetically

produced and naturally gathered data sets accentuate different qualities of classification performance. Because the responsible softmax layer is inspired by mixture modeling, training on generated Gaussian mixture model data is one natural choice for synthetic data. The MNIST data set was chosen as a familiar baseline for the behavior of an responsible softmax layer. More detailed discussion of the choices that went into data set selection is in section 5.3.

It is also the case that distinct evaluation metrics provide distinct insights to distinct classification methods. The fact that no model does well (or poorly) on all metrics necessitates application of several metrics to compare different models. A standard set of metrics used for classification are precision and accuracy. These have several drawbacks, especially in the multiclass setting, but they are still very informative. In the case such as testing with generated data, an idealized model is available so the performance of the neural nets can be compared to the idealized model. Discussions on metric selection are in section 5.4. Results of the experiments are covered in sections 5.5 and 5.6.

The final section of this chapter discusses The conclusions of the analyses and focuses on possible future studies.

5.1 Empirical Evidence of Convergence Rate

The responsible softmax layer requires additional compute time over a standard softmax layer. The extra compute time is governed mostly by convergence of $R_F^n(\boldsymbol{\pi}_0)$ to the fixed point $\hat{\boldsymbol{\pi}}$. Thus understanding how dynamic responsibility converges informs and directs the discussion of performance of the responsible softmax layer.

In addition, a discussion of convergence guides the choice of hyperparameter C. Recall that this parameter was introduce in section 4.2 to prevent dynamic responsibility from converging to a point on the boundary of the simplex. Thus C should likely be set inversely proportional to the convergence rate. As will be seen, this might suggest a connection of C to the batch size used for training.

Where discussion turns to convergence, the discussion must establish a measure of convergence rate. One method is to measure the growth rate of $a_n^d := d(\boldsymbol{\pi}_n, \boldsymbol{\pi}_{n+1})$ for a given distance function $d(\cdot, \cdot)$. Here $\boldsymbol{\pi}_n := R_F^n(\boldsymbol{\pi}_0)$ is the n-th iteration of $R_F(\boldsymbol{\pi}_0)$. For the purposes of this section, the distance function $d(\cdot, \cdot)$ will be euclidean distance unless otherwise noted.

Because DR is a contraction mapping in a neighborhood of $\hat{\boldsymbol{\pi}}$, a_n decreases exponentially. This exponential decrease is connected to the spectrum of DR, which is closely tied to the hessian of ℓ_F, and therefore to the singular values of F. This connection will be explored below.

Another measure of convergence rate is the error $\epsilon_n := \|R^n(\boldsymbol{\pi}_0) - \hat{\boldsymbol{\pi}}\|$. For Newton methods the error decreases quadratically, in that $\epsilon_{n+1} \sim c\epsilon_n^2$ for some constant $c < 1$. Iterative methods such as dynamic responsibility do not typically converge quadratically, but rather linearly. In other words for dynamic responsibility it is expected that $\epsilon_{n+1} \sim c\epsilon_n$. Quadratic decrease is desirable because the convergence happens in many fewer steps. It will be sufficient to look at the distances between iterates a_n, because by the triangle inequality $\epsilon_n - \epsilon_{n+1} \le a_n \le \epsilon_n + \epsilon_{n+1}$.

As a practical matter, algorithm 1 determines convergence of the series $\boldsymbol{\pi}_n$ when the distance a_n is less than a specified tolerance. This means that any comparison of the sequence a_n generated by different F must look at some invariant. Since the expected convergence is linear, the graph of $\log(a_n)$ versus n should be roughly linear, and so slopes can be compared.

To empirically estimate the convergence rate of dynamic responsibility, perform an experiment as follows:

Experiment 5.1.

1. Choose K different normal distributions $f_k(\boldsymbol{x}) \sim \mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\theta}_k)$ and mixing proportions $\{\pi_k\}_{k=1}^K$, $\sum_{k=1}^K \pi_k = 1$.

2. Generate N samples $\mathcal{X} = \{\boldsymbol{x}^n\}_{n=1}^N$ as in experiment 2.1.

3. Set $F_k^n = f_k(x^n)$ for $1 \le k \le K$ and $1 \le n \le N$. We will have $F \in M_{K \times N}$.

4. Set $\boldsymbol{\pi}_0 = \dfrac{1}{K}\mathbb{1}_K$ and iterate $R_F(\boldsymbol{\pi})$ until convergence. Record $\{\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots, \boldsymbol{\pi}_L\}$ where L is the number of steps that it takes to reach tolerance criteria.

5. Set $a_n := d(\boldsymbol{\pi}_n, \boldsymbol{\pi}_{n-1})$ $n = 1, \ldots, L$, and perform regression on a_n. In practice, $\log(a_n)$ appears to be linear once n is about $\dfrac{L}{4}$.

6. Repeat steps 2 through 5 many (*e.g.* 100) times, and compare the regression parameters.

Experiment 5.1 was performed several times in MATLAB with $K = 5$ and N sampled among the integers in $[10^3, 5 \cdot 10^5]$. Results showed that $\log(a_n)$ is linear in n, with slope close to $-\sigma_F^{-1}$, where $\sigma_F = \max \text{svd}(F)$ is the largest singular value F. This holds for each randomly generated F. Figure 5.2 shows how both a_n and ϵ_n behave for different F. Both plots show sections of linear behavior for $\log(a_n)$ and $\log(\epsilon_n)$. Code for these runs can be found in appendix A.4, in the file `convRates.m`.

Given the lines formed by a_n for a particular matrix F, let b_F be the slope of that line. For the same F let σ_F be the largest singular value of F. Figure 5.1 shows a scatter plot of $\sigma_F^{-1} + b_F$ compared
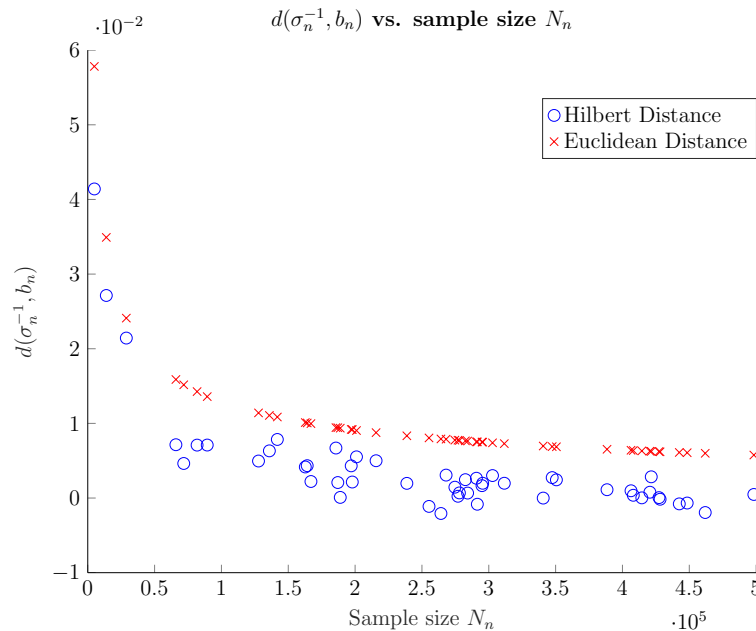
Figure 5.1: Linear regression was performed on the curves in figure 5.2 to get equations of the form $\log(a_n) = b_F n + c_F$ for various F. Then the max singular value σ_F for the same F was computed. The scatter plot above shows the distances $d(\sigma_F^{-1}, |b_F|)$ plotted against N, the number of columns of F for two different distances.

to N, the number of columns of F, for two different runs of experiment 5.1. Notice that as N increases the residues go to zero like $\frac{1}{\sqrt{N}}$. This shows an expected relationship between convergence rate and sample size, or batch size in the case of neural network training.

While it is clear that the curves in figure 5.2 are eventually linear, in the first few steps many of the curves shrink much faster than linear. This suggests that most of the convergence happens in the first few iterations. It also encourages setting the hyperparameter C for the responsible softmax layer to a small value. As will be seen in section 5.5, optimal values for C are less than 10.

$\log(a_n)$ vs. n
Euclidean distance

(a) Plot of a_n for several F

$\log(\epsilon_n)$ vs. n
Euclidean distance
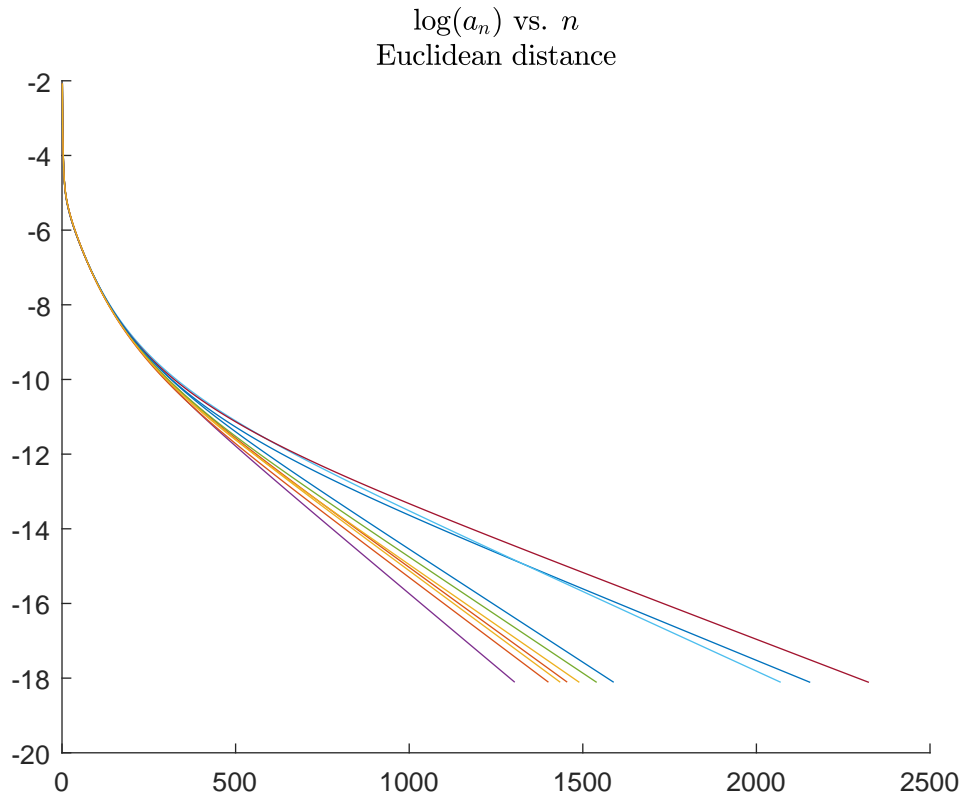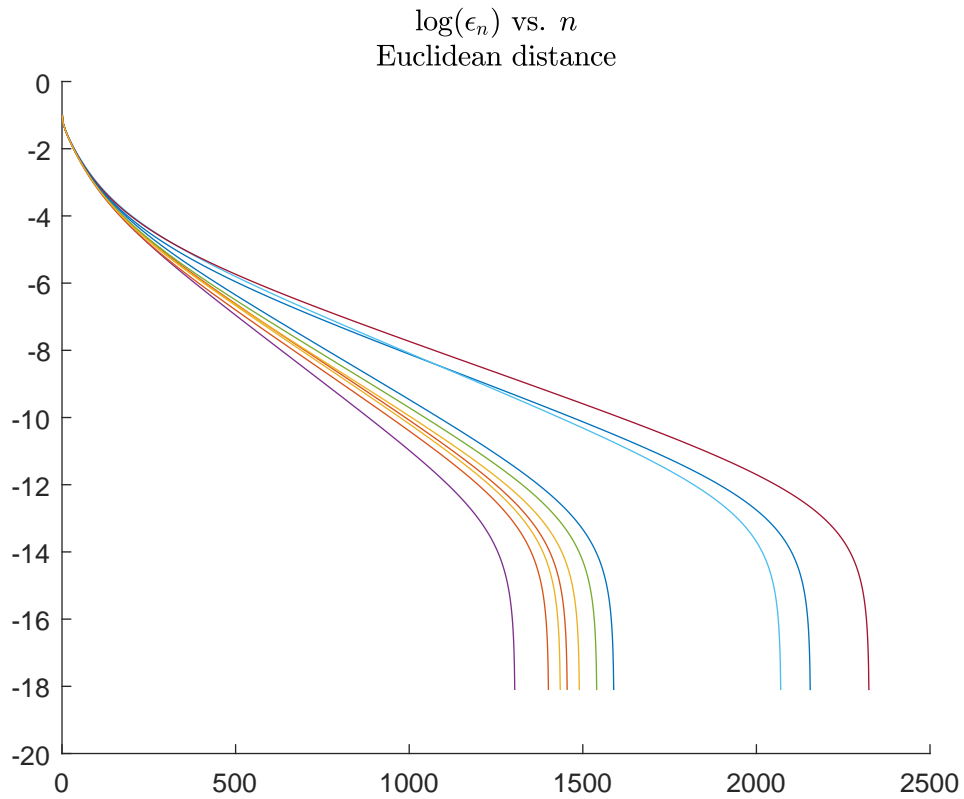
(b) Plot of ϵ_n for several F

Figure 5.2: The two plots show a_n and ϵ_n for euclidean distance. Each individual curve represents a different parameter matrix F.

5.2 Experimental Neural Network Setup

When comparing multiple neural nets, all nets in consideration must posses identical structures and initialization aside from the changes tested. This makes it more likely that any changes in the outcomes are solely from changes in the final layers. For testing responsible softmax , it is possible to use any neural network where softmax would be used, which provides many potential options for testing. This section covers the common settings of basic neural nets for the experiments to be performed. This includes choices of layer structure, transfer functions, weight initialization, learning rate and batch size.

This paper proposes a change only to the final layers of a neural network used for classification, so for each experiment on a data set, the initial layers and parameters for those layers will be exactly the same. However, weight initialization can have a large effect on final results for generalization so several training runs are done with different random initializations of the weights. Each final layer gets the same initialization of the weights on previous layers through the use of random number generator seeds.

Because the dynamic responsibility supposes differntiability of the responsible softmax layer reacts sensitively to differentiability, the choice of activation layers requires careful consideration. In all of the experiments, tanh layers were used to preserve differentiability of layer predictions. While several runs tried to use ReLU layers, this led to catastrophic instability in the neural nets with responsible softmax layers.

For the generated data, the numerical experiments use low dimensionality $D = 2$ data sets, so that only fully connected layers are necessary. Since wider nets tend to perform better, after the input layer there are 2 fully connected layers with a tanh activation layer between them. The first layer has width $K * 4$, and the second layer is required to be of width K for using either softmax or responsible softmax. Table 5.1 summarizes this setup.

Layer	Width
Input	$D = 2$
Fully connected	$4K$
tanh layer	N/A
Fully connected	K

Table 5.1: A summary of the initial layers for classifying data generated from a GMM.

For the MNIST data set [LBBH98] the dimensionality $D = 28^2 = 784$ of the data set encourages

the use of a convolutional neural net (CNN) for reasonable results. It is not the purpose of this paper to cover CNN architecture in detail, so only the specific implementation used in training will be mentioned here. Those interested in details of CNNs are encouraged to consult common references such as [LBH15, Sch15, GBC16].

In the interest of reduced training time, experiments used only a very shallow CNN with only one convolutional layer. This layer is followed by a tanh activation layer, a max pooling layer, and finally a fully connected layer. The parameters of the layers used for all neural net classification of MNIST data in this dissertation are as described in table 5.2.

Layer	Parameters
Input	28×28 inputs
Convolutional layer	11 5×5 filters
tanh layer	N/A
Max pooling layer	2×2 with stride 3
Fully connected Layer	K

Table 5.2: Parameters of initial neural net layers used in MNIST classification.

As discussed in sections 4.2 and 5.1, the responsible softmax layer requires a hyperparameter C. Testing how C affects classification requires numerical experiments on multiple responsible softmax layers, so several nets were trained with varying values of C. Most of the experiments trained two nets with responsible softmax layers with iteration parameters $C = 1, 4$. In the case of classifying generated data, several more values of C were explored. These results are shared in section 5.5.

An reasonable baseline to test the responsible softmax layer against is the standard softmax layer. Another is the softmax layer weighted with priors derived from the relative frequency of classes in the training labels T. For the purpose of comparing to responsible softmax layers, a 'fixed' responsible softmax layer was used in numerical experiments. A fixed responsible softmax layer is equivalent to taking $C = 0$ in a normal responsible softmax layer. The initial value of $\boldsymbol{\pi}_0$ is then set to an appropriate value for testing. In the case of all experiments run the fixed value $\boldsymbol{\pi}_0 = \boldsymbol{\pi}^*$, where $\boldsymbol{\pi}^*$ is established in creating or sampling the training and test data.

In summary, for most of the experiments run, four nets will be used. For illustration purposes some nets were also trained with large values of C on GMM data. Table 5.3 outlines the most common nets used in experiments.

For all the neural net training runs, the batch size and initial learning rate were all set the same. Batch size was set to 100. The learning rate was controlled by the Adam adaptive learning rate algo-

Net	Initial Layers	Classification layer
GMM net #1	GMM Layers	Softmax
GMM net #2	GMM Layers	Responsibility Softmax $C = 1$
GMM net #3	GMM Layers	Responsibility Softmax $C = 4$
GMM net #4	GMM Layers	Fixed Weight Softmax
MNIST net #1	MNIST Layers	Softmax
MNIST net #2	MNIST Layers	Responsibility Softmax $C = 1$
MNIST net #3	MNIST Layers	Responsibility Softmax $C = 4$
MNIST net #4	MNIST Layers	Fixed Weight Softmax

Table 5.3: Each net is classified by the type of data it is designed to process (initial layers), the type of softmax layer used, and the weights or hyperparameter used relative to the type of softmax layer used. GMM layers is a reference to table 5.1, and MNIST layers is a reference to table 5.2.

rithm. In consideration of the relationship between the hyperparameter C and batch size mentioned in section 5.1, a good choice for future experiments would entail an exploration of this connection.

5.3   Data Selection

As has been mentioned two types of data sets, synthetic and natural, are used for numerical experiments in this dissertation. The purpose of using different data is to highlight strengths and weaknesses of distinct models. Only reasonably simple data sets were chosen, to act as proof of concept. This also allows faster exploration of responsible softmax behavior. The generated data sets were various forms of GMMs, and the natural data set used is the MNIST data set of LeCun et.al. [LBBH98].

As the responsible softmax layer drew inspiration from the soft K-means and EM algorithms [Mac02], a natural choice for synthetic data is pseudorandom data drawn from a Gaussian mixture model. The first data sets used in this vein were one dimensional mixtures and informed the selection of later mixture models. The final data used was a 2 dimensional Gaussian mixture with $K = 5$ clusters and anisotropic covariance matrices which varied among clusters.

In all cases, the means of the clusters were separated widely, though two of the clusters were chosen to be minority classes. Since a responsible softmax layer is designed to find a MLE for class weights, it is uniquely suited to direct training with imbalanced data. For this reason many tests were performed with varying mixture components. A sample from the final GMM on which the majority of

experiments were run is shown in figure 5.3. Details and code to reproduce this data set are included in appendix C.
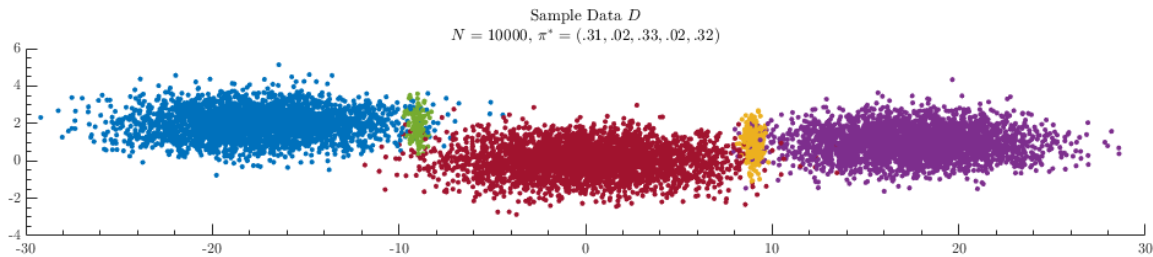
Figure 5.3: This figure represents data samples from the most common GMM used to train, test and compare different types of neural nets. The minority classes are the ones with smaller variance. Data is generated for train and tests sets on an 'as needed' basis.

The EM and soft K-means algorithms are designed unsupervised learning and therefore have some heuristics on the data built in. This naturally provides some rigidity in data analysis and exploration which can provide a sense of interpretation to the results. Standard neural networks are more flexible in the data they can model, but tend to lack interpretability. Because the responsible softmax layer uses neural networks and supervised learning to make predictions, there is more flexibility in the type of data it can process, but the use of dynamic responsibility provides a sens of interpretability. The choice of GMM data and MNIST data to test responsible softmax reflects this tradeoff between flexibility and interpretability.

Many data sets and methods were used before the experiments reported in this dissertation were run. These trial data sets informed selection of the final data sets. Most of the exploratory trials are not reported here, but the final choice of data sets is highly reflective of this process. For example, when re-sampling the MNIST data set to introduce imbalance in training and test data, I chose Benford's law for class frequencies to illustrate that responsible softmax has a reasonable effect on highly imbalanced data sets. Sections 5.5 and 5.6 mention these choices where appropriate. The result sections also suggest some other data sets to explore in the future.

5.4   Evaluation Methods

The methods employed to evaluate the classification ability of the various neural nets fall into three broad categories: confusion matrix methods, per class precision and recall, and comparison of the classifiers to an idealized classifier. These methods inform each other to create a clearer picture of the performance for each neural net.

While accuracy is a commonly used metric for classification, in cases where data is imbalances, it can be a grossly misleading metric. For example, consider a binary classification problem where more than 95% of samples fall into a single class. Call this set class 1. Then a classifier that always returns class 1 would have an accuracy of 95%, but would be useless for any sort of discrimination. Similar problems occur with other metrics such as precision and recall. One simple method that is a partial remedy to this problem is the use of *per class* metrics.

In reporting per class attributes, confusion matrices offer a quick summary of how data points were classified and misclassified. A confusion matrix $\mathcal{C}$ has entries $c_{i,j}$, $i,j = 1, \ldots, K$, determined by the number of samples belonging to class i that were classified into class j. Thus the trace of $\mathcal{C}$ is the number of correctly identified examples. The sum of off-diagonal entries is the number of incorrectly classified samples. The ratio of these two sums is the overall accuracy of the classifier. Similar combined and per class metrics may be calculated using a confusion matrix. Thus confusion matrices contribute heavily to the reporting of results in sections 5.5 and 5.6.

Per class precision and recall scores suggest important trade-offs that must be considered when training a classifier. These metrics can be expressed in terms of the entries of the confusion matrix $\mathcal{C}$. If p_i represents the precision of class i for a given classifier, and r_i represents the recall of the same class, then

$$p_i = \frac{c_{i,i}}{\sum_j c_{j,i}} \tag{5.1}$$

$$r_i = \frac{c_{i,i}}{\sum_j c_{i,j}} \tag{5.2}$$

In other words p_i is the number of data points correctly classified in class i divided by the number of data points put into class i by mistake (diagonal entry divided by the column sum). The per class recall r_i is the number of correctly classified data points divided by the number of data points from class i incorrectly classified into a different class (diagonal entry divided by the row sum). There are many other metrics that may be calculated from a confusion matrix, but these are the two used in the results sections.

Idealized classifiers are difficult to identify and frequently even more burdensome on computation. Fortunately for GMMs, Bayes rule gives an excellent classifier when confidence in parameter values is high. In the case of generated data, the functions $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$ and mixing proportions $\boldsymbol{\pi}^*$ are known. Then given data points $\boldsymbol{x}_n$ of unknown classifications, the classes may be approximated by the maximum *a*

posteriori (MAP) estimate

$$c_n = \arg\max_k \frac{\pi_k^* f_k(\boldsymbol{x}_n, \boldsymbol{\theta}_k)}{\sum_j \pi_j^* f_j(\boldsymbol{x}_n, \boldsymbol{\theta}_j)}. \tag{5.3}$$

In some sense, the MAP classifier is the best that can be used in this situation.

One visually pleasing method to compare classifiers of low dimensional data is to plot classification regions. While neural net classification regions are not usually the same as the classification regions of a MAP classifier, the comparison is interesting and useful. For typical deep neural nets, the arxiv paper by Fawzi *et al.* [FMDFS17] shows empirically that classifications regions tend to be connected. Section 5.5 presents some neural nets using a responsible softmax layer with large hyperparameter C which have locally disconnected classification regions.

The easiest way to compare with an idealized classifier for the MNIST data set is to assume that the confusion matrix for such a classifier is perfect, *i.e.* the sum of off diagonal entries is zero. Obtaining such a classifier from a neural net is often an indication of overfitting during training. Section 5.6, explores an example where using a responsible softmax layer helps get closer to this ideal, but still appears to generalize well.

As purpose of this chapter is not to fully explore the performance, but rather to show that responsible softmax works in a predictable manner. Though there are many metrics to choose from, the performance metric chosen show that in cases of extreme imbalance, responsible softmax is a reasonable choice. The metrics chosen also hint at avenues of further exploration.

5.5 Results for GMM

The Gaussian mixture model used for this section has parameters determined by code in appendix C.

The training and test data for each run are generated independently by MATLAB. Where appropriate, the RNG seed was saved to reproduce results. Each net was trained on the same training data for each run, and the RNG was seeded with the same seed before each training of a neural net to ensure the same initialization parameters. The training set was partitioned into training and validation sets prior to beginning training. An independent test set is also generated before each run. Training, validation, and test sets are used 300 of times for each run, and 5 runs are performed. This section only reports on the outcomes of the first run, but similar results were achieved on each run.

The confusion matrices for the neural nets described in section 5.2 are shown in table 5.4.

Table 5.5 shows per class precision and recall for the same nets on the same training and test data.

One interesting aspect pf these tables is that as the hyperparameter C increases, the per class precision and recall also increase. However, as figure 5.7 shows, this increase in per class precision and recall is not strict. This suggests that choosing a value of C which maximizes per class precision and recall may be a reasonable option.

Where the parameters for $\boldsymbol{\pi}^*$, $f_k(x, \boldsymbol{\theta}_k)$ are known, a reasonable 'ideal' classifier may be obtained by using a maximum *a posteriori* estimate as discussed in section 5.4. An even better option is to treat the outputs of the neural nets as point estimates of per class probabilities for data points, and compare that to the discrete posterior distribution given by Bayes rule. This might be a good method to study in future work.

The images in figures 5.4 and 5.5 show classification regions for all the mentioned classifiers. As discussed, the neural net with the softmax layer completely misses the underrepresented classes, and does not even have a classification region for one of those classes. Looking at these figures helps to explain why despite doing poorly on classification for minority classes, softmax is the best at overall precision and recall. However, when looking at per class precision and recall, standard softmax does poorly, as shown in table 5.5.
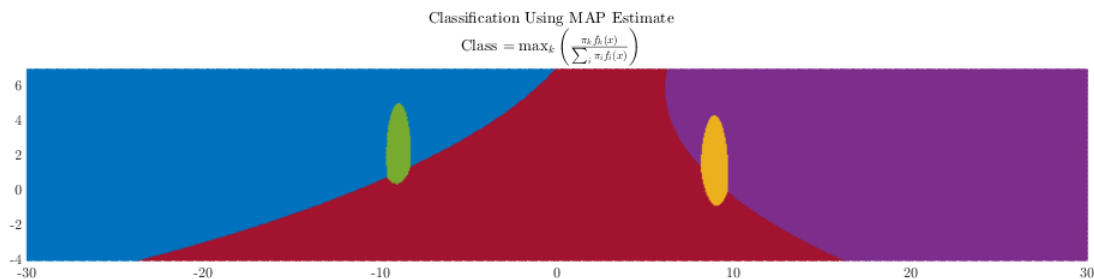
Figure 5.4: This figure represents the classification regions for a MAP estimator of the data in figure 5.3. The class is estimated via equation (5.3), since the distributions and mixing components are known.
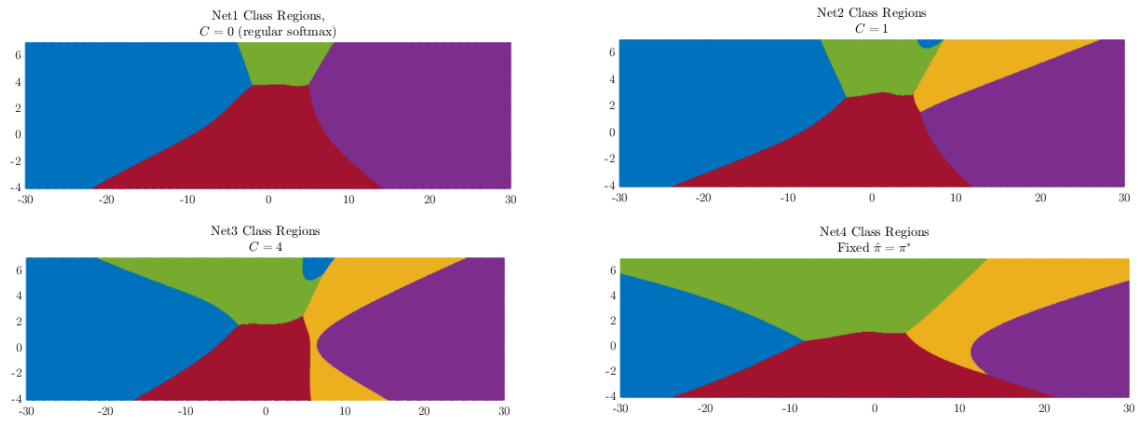
Figure 5.5: This shows classification regions for GMM Nets #1-#4. Colors are the same as in figures 5.3 and 5.4.

30.253±.001	0.0	.027±.001	0.0	0.0
1.680±.000	0.0	0.0	0.0	0.0
.328±.004	0.0	32.206±.008	0.0	.706±.006
0.0	0.0	.033±.001	0.0	3.207±.001
0.0	0.0	.021±.001	0.0	31.539±.001

(a) Confusion table for GMM Net #1.

30.165±.004	.101±.004	.014±.001	0.0	0.0
1.616±.003	.063±.003	0.0	0.0	0.0
.398±.006	.114±.003	31.739±.010	.330±.008	.659±.009
0.0	0.0	.031±.001	.333±.012	2.875±.012
0.0	0.0	.012±.000	.082±.004	31.466±.004

(b) Confusion table for GMM Net #2.

29.897±.010	.374±.010	.009±.001	.000±.001	0.0
1.273±.011	.406±.011	.001±.001	0.0	0.0
.658±.016	.595±.017	29.916±.036	1.329±.031	.743±.018
0.0	.000±.001	.013±.001	1.221±.027	2.006±.027
0.0	0.0	0.0	.340±.009	31.220±.009

(c) Confusion table for GMM Net #3.

26.842±.035	3.438±.035	0.0	0.0	0.0
.044±.006	1.636±.006	0.0	0.0	0.0
.075±.003	1.841±.024	28.737±.037	2.540±.025	.047±.002
0.0	0.0	.027±.001	3.122±.004	.092±.004
0.0	0.0	0.0	2.463±.023	29.097±.023

(d) Confusion table for GMM Net #4.

Table 5.4: The nets were tested on a set of samples drawn independently from the training set. Values are reported as percentages for clarity. Test data sample size $N = 2500$ for all runs. Error intervals are 95% confidence standard error. An entry of 0.0 indicates that all values were zero to 3 decimal places.

GMM Net 1		
Class	Precision	Recall
1	0.936	0.999
2	0.000	0.000
3	0.998	0.966
4	0.000	0.000
5	0.979	0.999

(a) Precision and Recall table for GMM Net #1.

GMM Net 2		
Class	Precision	Recall
1	0.935	0.997
2	0.209	0.027
3	0.998	0.953
4	0.401	0.090
5	0.898	0.997

(b) Precision and Recall table for GMM Net #2.

GMM Net 3		
Class	Precision	Recall
1	0.934	0.988
2	0.279	0.194
3	0.999	0.894
4	0.385	0.364
5	0.918	0.989

(c) Precision and Recall table for GMM Net #3.

GMM Net 4		
Class	Precision	Recall
1	0.988	0.930
2	0.289	0.882
3	0.999	0.868
4	0.380	0.969
5	0.996	0.922

(d) Precision and Recall table for GMM Net #4.

Table 5.5: This table shows per class precision and recall for GMM nets trained and tested on the same data as in table 5.4

The responsible softmax layers each converged to a value for $\hat{\boldsymbol{\pi}}$ as an estimate of $\boldsymbol{\pi}^*$. As summarized in table 5.6, the neural nets made reasonable estimates of $\boldsymbol{\pi}^*$. Note that these values are similar to the percentages found in the counfusion matrices of table 5.4.

	π_1	π_2	π_3	π_4	π_5
GMM Net #2	0.2359	0.0389	0.2299	0.2440	0.2512
GMM Net #3	0.2441	0.0254	0.2286	0.2478	0.2541
$\boldsymbol{\pi}^*$	0.2489	0.0206	0.2441	0.2431	0.2433

Table 5.6: A representative example of trained class weights for GMM nets #2 and #3. The class weight vector $\boldsymbol{\pi}^*$ used for generating the GMM data is shared for comparison.

A one-off experiment was also performed with different class weights and on GMM nets with large values of C. As seen in figure 5.7, larger values of the Hyperparameter C can lead to unexpected outcomes. Figure 5.6 shows the classification regions for the MAP estimator on the data set used to train the nets in figure 5.7.
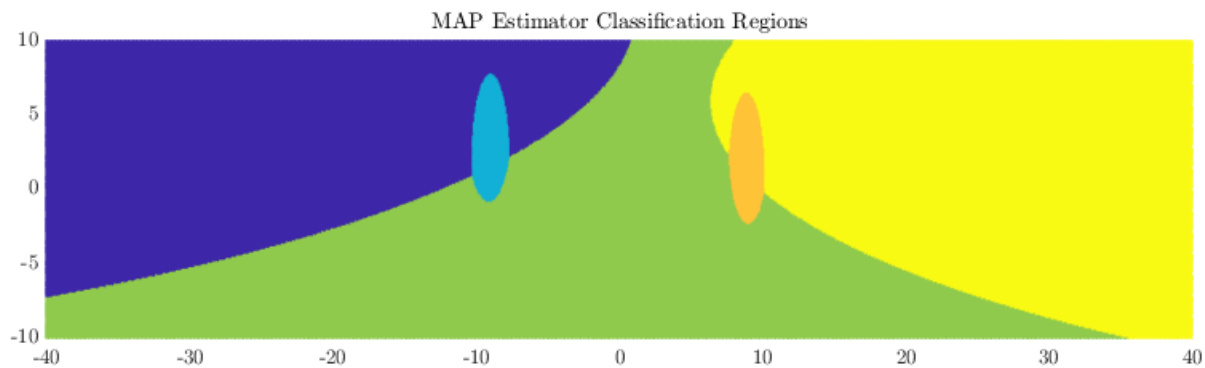
Figure 5.6: In this data set, $\boldsymbol{\pi}^* = (0.4466, 0.0227, 0.0302, 0.0417, 0.4589)$, and all other GMM parameters are as in figure 5.4.
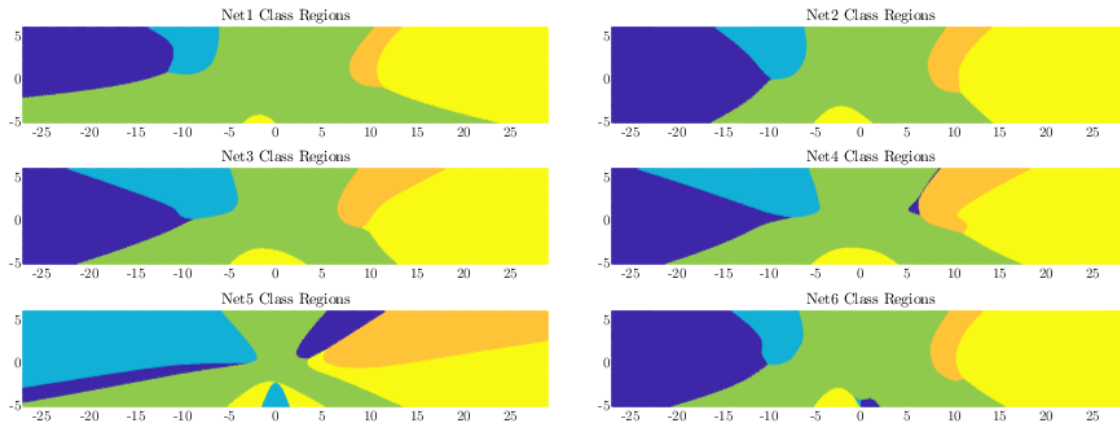
Figure 5.7: The first and last classification region images represent softmax and fixed weight softmax respectively. Images 2-5 show the classification regions for nets with responsible softmax layer and hyperparameter $C = 1, 4, 8, 16$ respectively. Colors of each class are as in figure 5.6.

5.6 Results for MNIST

The standard MNIST data set of LeCun *et al.* [LBBH98] consists of $60,000$ digitized grayscale handwritten digits. The digits are a curated selection from the much larger $NIST$ handwritten digit data set. The MNIST data is separated into a training set of $50,000$ digits and a test set of $10,000$ digits. The digits 0 thru 9 are roughly equally represented in the data, as there are roughly 6,000 of each digit in the entire data set.

A preliminary experiment was run with training MNIST nets #1-#3 on the unaltered MNIST data set. All nets performed roughly equally well, in that there was no significant statistical difference at the 95% confidence level. The nets with a responsible softmax layer also had $\hat{\pi}$ converge fairly closely to the proportions of classes in the data. The test set has slightly different proportions than the training set, and the responsible softmax layers still generalized reasonably well. Code for this experiment can be found in appendix D.

Results such as this are not surprising. Given close to equal proportions in the data, the responsible softmax layer is approximately equal to the softmax layer. As seen in section 5.5, the responsible softmax layer handles imbalanced data better than the softmax layer. Thus to see different results between responsible softmax and softmax. the MNIST data set must be resampled to create a unbalanced training and test sets.

Several different weights of all 10 digits were tried, and ultimately weights corresponding to Benford's law [Ben38] were selected. While Benford's law is common in some natural processes, the choice

to use these weights is reflective only of the fact that there is a large discrepancy in the relative frequency of the samples. For example, the digit 1 will occur 6 times more often than the digit 9.

All the charts and tables in this section are for MNIST nets trained on data resampled to have weights closer to the distribution described by Benford's law. This means that the training and test sets for these experiments are subsets of the original MNIST data. To account for the difference that comes from initial weights of a neural network, each net was trained 40 times on the same (disjoint) subsets of training and test data. As mentioned in section 5.2 all nets were initialized the same for each run. This experiment was run 3 times with different selections of test and training data, only results from the first experiment are shown here. Code to run this numerical experiment can be found in appendix D.

	Sum of off diagonals
MNIST Net 1	$4.043 \pm .612$
MNIST Net 2	$3.783 \pm .600$
MNIST Net 3	$3.788 \pm .579$
MNIST Net 4	$3.546 \pm .600$

Table 5.7: Relative percent sums of off diagonal entries of the confusion matrices for MNIST. This is equivalent to 100 minus accuracy of the net as a percentage. Reported intervals are standard error 95% intervals
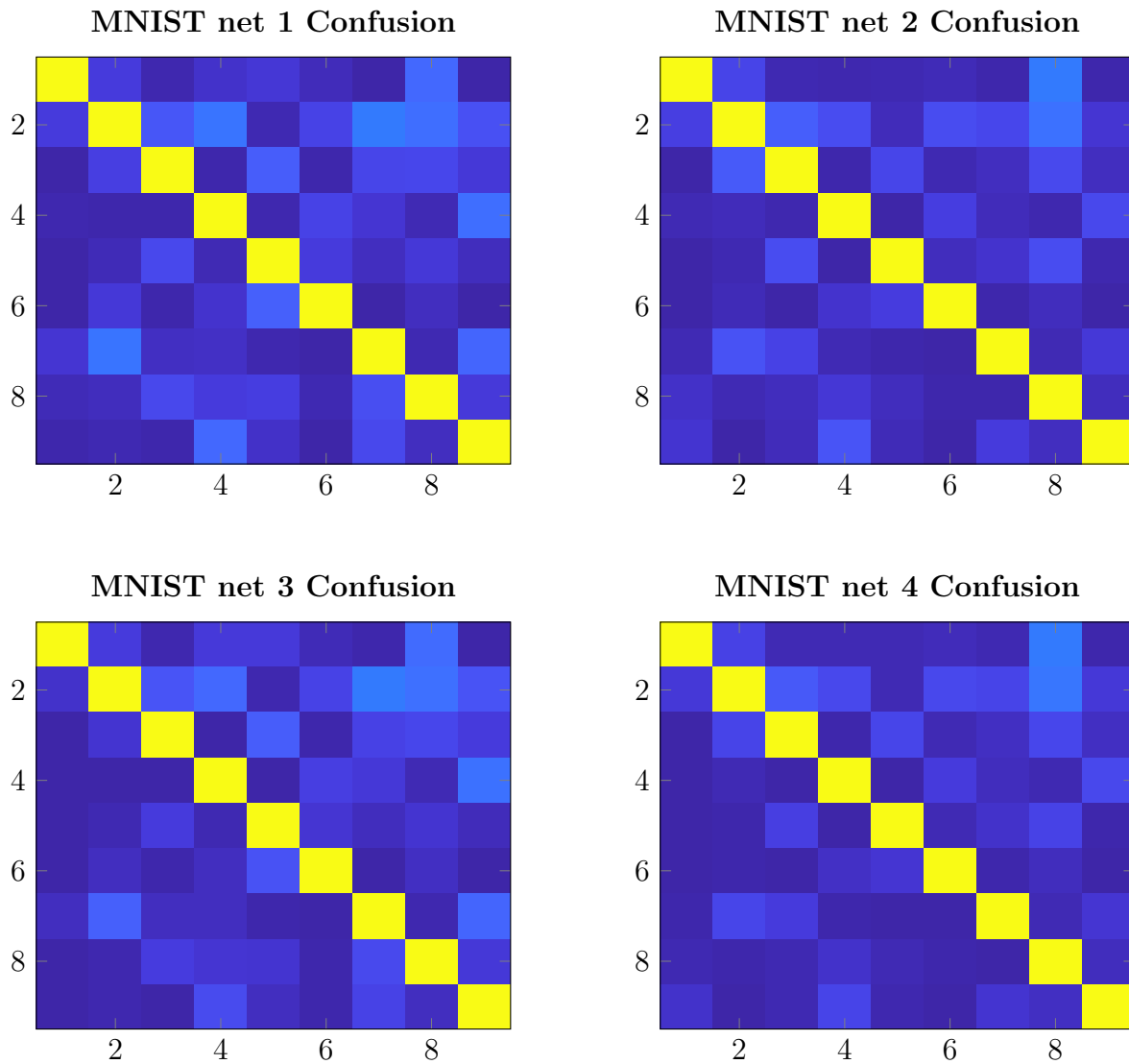
Figure 5.8: This figure represents the confusion matrices for MNIST nets #1-#4. The images are color coded to indicate high versus low values. Higher values are brighter yellow, and lower values are blue. Notice that the responsible softmax layer confusion maps have deeper blues in the off diagonal.

	Class 1 rel. pct.	Class 2 rel. pct.	Class 3 rel. pct.
softmax	32.730 ± .028	16.745 ± .040	11.869 ± .027
RS, $C = 1$	32.652 ± .041	16.715 ± .037	11.806 ± .030
RS, $C = 4$	32.670 ± .036	16.720 ± .052	11.867 ± .028
Fixed	32.586 ± .049	16.681 ± .045	11.824 ± .036
Benford's Law	30.1	17.6	12.5
	Class 4 rel. pct.	Class 5 rel. pct.	Class 6 rel. pct.
softmax	8.434 ± .030	6.155 ± .028	6.355 ± .023
RS, $C = 1$	8.546 ± .027	6.083 ± .035	6.472 ± .015
RS, $C = 4$	8.427 ± .026	6.208 ± .021	6.411 ± .022
Fixed	8.555 ± .026	6.148 ± .036	6.502 ± .016
Benford's Law	9.7	7.9	6.7
	Class 7 rel. pct.	Class 8 rel. pct.	Class 9 rel. pct.
softmax	5.501 ± .022	4.126 ± .025	4.041 ± .030
RS, $C = 1$	5.590 ± .019	4.309 ± .021	4.044 ± .027
RS, $C = 4$	5.561 ± .023	4.223 ± .021	4.125 ± .022
Fixed	5.657 ± .021	4.389 ± .015	4.113 ± .024
Benford's Law	5.8	5.1	4.6

Table 5.8: Confusion matrix diagonal for various nets trained on nets using MNIST data resampled according to Benford's law. If a perfect accuracy classifier were trained, then the percentages would reflect the bottom row of each subtable. Confidence intervals are standard error 95% confidence intervals

	$\hat{\pi}_1$	$\hat{\pi}_2$	$\hat{\pi}_3$	$\hat{\pi}_4$	$\hat{\pi}_5$	$\hat{\pi}_6$	$\hat{\pi}_7$	$\hat{\pi}_8$	$\hat{\pi}_9$
RS, $C = 1$	32.16	16.78	12.26	8.97	6.66	6.61	6.33	5.61	4.58
RS, $C = 4$	32.30	16.83	12.19	9.03	6.54	6.79	6.11	5.37	4.80
Benford	30.1	17.6	12.5	9.7	7.9	6.7	5.8	5.1	4.6

Table 5.9: Final $\hat{\boldsymbol{\pi}}$ for MNIST nets #2 and #3 trained on Benford weighted MNIST data. These weights are recorded as percentages for comparison to Benford's law.

5.7   Summary of Conclusions

This dissertation defined dynamic responsibility and showed that the only stable point of dynamic responsibility the is MLE of the log likelihood function for a discrete mixture model. Future studies could provide further exploration of convergence rate, and bounds on singular values for families of the matrix of parameters F. Another option for future exploration is to use Newton-Kantorovic method on R_F to get better convergence bounds. Newton methods can give negative weights, but exploration of how to prevent this could provide better comparison to dynamic responsibility. This line of thought reflects the original question that lead to this research.

Having proved good properties of dynamic responsibility, this paper then defined responsible softmax , computed derivatives for backpropagation, showed that the fixed point $\hat{\boldsymbol{\pi}}_F$ varies smoothly with F, and found a connection between empirical convergence rate and the singular values of F. Numerical tests were performed on some simple data sets with good results. Future studies could identify and try other data sets. The adaptability of responsible softmax would also allow experiments with other neural net architectures, *e.g.* LSTM, VAE, MOE. Given the general assumptions on F, responsible softmax might also work well with nonparametric methods, like Gaussian Processes.

Much of the analysis of dynamic responsibility and the responsible softmax layer required understanding of the Frechét derivative DR_F. The close relation that DR_F has to the second derivative of the log likelihood function ℓ_F implies a similarly close relation to the Fisher information matrix $\mathcal{I}(\boldsymbol{\pi}) = -E\left[\frac{\partial^2 \ell_F}{\partial \boldsymbol{\pi}^2}\,\middle|\,\boldsymbol{\pi}\right]$. Exploration of this connection beyond what is mentioned in section 3.5 could be interesting on its own. This would require more assumptions on the distributions $f_k(\boldsymbol{x}, \boldsymbol{\theta}_k)$, but may provide additional further avenues of research.

Appendix A

Dynamic Responsibility Code

A.1   Implementation of Responsibility Map

Below gives an implementation of the responsibility map from equation (3.1) in MATLAB [Mat20]. All code in the appendices is written for MATLAB, more information may be found at the personal website `https://www.math.arizona.edu/~rcoatney`.

simplex_map.m

```matlab
function [ new_p ] = simplex_map(  f_dist, old_p, method )
%SIMPLEX_MAP Take in the K by N paramaters f_dist and K by 1 input old_p to create new_p.
% new_p will be a K by 1 vector.
% The purpose of this map is to apply a nonlinear function defined by the
% parameters F_DIST, and apply it to the inputs OLD_P.  This function is
% useful in the study of K-means clustering.
%  F_DIST is an N by K matrix of values taken from K probability
%   distributions on N samples.  The only requirement on F_DIST is that the
%   entries are positive.
%  OLD_P is a 1 by K vector such that the sum of the entries is equal to 1.
%  NEW_P is a 1 by K vector such that the sum of the entries is equal to 1.


[K, N]=size(f_dist);
assert(length(old_p)==K);

if strcmp(method,'ratio')
    prods=bsxfun(@times,f_dist,old_p);
    sums=sum(prods,1);
    assert(length(sums)==N);
    ratios=bsxfun(@rdivide,prods,sums);
    new_p=sum(ratios, 2)/N;

elseif strcmp(method,'diff')
    %Worries about underflow!
    denoms=1/N*(1./(f_dist'*old_p));%good here 2/22
    dl=f_dist*denoms;%good here!2/22
    new_p=dl.*old_p;

else
    sprintf('You must enter a method of diff or ratio');
end
```

A.2   Implementation of Algorithm 1

Algorithm 1 iterates the responsibility map until convergence. Here the stopping point is determined by a given absolute tolerance. This may need to change for large K.

stablepoint.m

```matlab
function [stable_dist, orbit] = stablepoint(f_dist, p_dist, err,...
                                            method, orbitON)
%STABLEPOINT   Implements iteration of simplex_map to convergence.
% Different computation methods are available.
%    F_DIST Is a K by N matrix representing the values of the K
%        distributions over the N sample points. The parameters that
%        determine the SIMPLEX_MAP
%    P_DIST Is a K by 1 vector representing initial Mixing probabilities
%    ERR is an error term deciding when to stop.  Should not be bigger than
%    about 12 or 14. This is because log10(eps())-4 = 12 for doubles. For
%    single floating points, this should not be bigger than 4 or 6.
%
%    STABLE_DIST is the stable point to which the iteration of SIMPLEX_MAP
%    converges with the given starting parameters.
%    ORBIT is the orbit of P_DIST under iterations of SIMPLEX_MAP


%% TODO:defaults and varargin handling
stop=2*10^(-err);
p=p_dist;
if orbitON
    orbit=zeros(size(f_dist,1),10^(ceil(err/2)));
end

if strcmp(method,'newton')
    [stable_dist,orbit]=stablepointNewton(f_dist,p,err);
else
    i=1;
    new=simplex_map(f_dist,p,method);
    %Current implemetation uses absolute toleranve. Relative tolerance may
    %be considered at a future date.
    while (sum(abs(p-new))> stop)
        if orbitON
            orbit(:,i)=p;
        end
        i=i+1;
        p=new;
        new=simplex_map(f_dist,p,method);
    end
    stable_dist=new;
end
default = [p_dist,stable_dist];
if orbitON
    orbit=orbit(:,any(orbit,1));
    if isempty(orbit)
        orbit=default;
    end
else
    orbit=default;
end
end
```

A.3 Implementation of Algorithm 2

The code here is only ever called through the file `stablepoint.m` with 'newton' as the method option.

stablepointNewton.m

```matlab
function [p_star,iterates] = stablepointNewton(F,p,err)
%STABLEPOINTNEWTON A cheap and easy way to determine a stable point of
%SIMPLEX_MAP
%   F Is a K by N matrix representing the values of the K
%       distributions over the N sample points. The parameters that
%       determine the SIMPLEX_MAP
%   P Is a K by 1 vector representing initial Mixing probabilities
%   ERR is an error term deciding when to stop.  Should not be bigger than
%   about 12 or 14.
%
%   P_STAR is the stable point to which the iteration of SIMPLEX_MAP
%   converges with the given starting parameters.
%   ITERATES is the orbit of P_DIST under iterations of the Newton method
K=size(F,1);
dx=ones(K,1);
newp=p;
iterates=p;
while (norm(dx)>10^-err*norm(p))
    [Hl,dl]=lDifferentials(F,newp);
    BHl=[Hl,ones(K,1);ones(1,K),0];
    dlf=[dl;0];
    dx=BHl^-1*dlf;
    dx=dx(1:K);
    newp=abs(newp-dx); %must stay positive
    newp=newp/sum(newp); %keep it on the simplex
    iterates=[iterates,newp];
end
p_star=newp;
end
%% TODO: See todos for simplex_map.m
```

The function `lDifferentials` called by `stablepointNewton` calculates the gradient and Hessian of $\ell_F(\boldsymbol{\pi})$ with respect to $\boldsymbol{\pi}$ at the point `newp`.

lDifferentials.m

```matlab
function [Hl,dl] = lDifferentials(F,p)
%LDifferentials calculateds the gradient and hessian of the averaged
%log likelihood of a joint distribution of N samples from a
%mixture of K different distributions.
%   F is a K by N matrix, the evaluations of each point in the various
%   mixture pdf's
%   P is a K by 1 vector of the probability components. sum(P) = 1.
N=size(F,2);

%This has the effect of multiplying each row by the same entry of p,
%and then summing the columns.
denoms=p'*F;%F is K by N, P is K by 1.
G=F./denoms;
dl=1/N.*G*ones(N,1);%

%This comes directly from lemma 4.3 in dissertation
Hl=-1/N.*(G*G');
end
```

A.4 Code for Experiments on Convergence

The script `error_samples` calculates $\hat{\boldsymbol{\pi}}_F$ for F calculated from several samples of Gaussian mixture data. It uses the helper function gfg`GMMData.m` to generate Gaussian mixture data via PRNG.

error_samples.m

```
1   %% Inital values setup
2   K=20;
3   N=1000;
4   sampleSize = 10000;
5   %The use of pwd requires that this script be run from the containing
6   %folder.
7   dir=strcat(pwd,'\errors data\');
8   %Shuffle chages the seeds every run. Set to 'default' or some other seed
9   %if consistent results are required
10  rng('shuffle');
11  scurr = rng;
12  %Filename has rng seed in it.
13  filename=strcat(dir,'gmm_errors_',num2str(K),'_',num2str(N),'_',num2str(scurr.Seed),'.csv');
14  %% Generate data
15  %currently creates GMM data.
16  seeds=randi([1000000,90000000],1,sampleSize);
17  errors=zeros(K,sampleSize);
18  D=2;
19  init=ones(1,K)./K;
20  W=waitbar(1/sampleSize,sprintf('Calculating Sample %d of %d',1,sampleSize));
21  for i=1:sampleSize
22      [F,P]=GMMData(K,N,D,seeds(i));
23      Phat=stablepoint(F,P',12);
24      errors(:,i)=P-Phat';
25      waitbar((i+1)/sampleSize,W,sprintf('Calculating Sample %d of %d',i+1,sampleSize));
26  end
27
28  %% File I/O
29  %CSVWRITE checks to be sure filename isn't already there.
30  % default behavior if the file exists is to overwrite data.
31  csvwrite(filename,errors');
```

GMMData.m

```matlab
function [GMMDistValues,P, data] = GMMData(K,N,D,seed)
%% Create GMM distribution parameters from the given seed
%% K-Number of Clusters, N-sample size, D-dimension of data
rng(seed,'twister');
Mu=rand(K,D);
Sigma=zeros(D,D,K);
for i=1:K
    Sigma(:,:,i)=full(sprandsym(D,(rand(1)+2)/4,rand(D,1)));
end
s=rand(1,K);
P=s./sum(s);

%% Create gmdistribution object, and a sample
gm = gmdistribution(Mu,Sigma,P);
X=random(gm,N);
data = X;

%% Create parameter matrix F as evaluation of PDFs at sample.
GMMDistValues =zeros(K,N);
for i=1:K
    GMMDistValues(i,:)=mvnpdf(X,Mu(i,:),Sigma(:,:,i));
end
end
```

convRates.m

```matlab
%TODO: make nice latex versions of these graphs!
clear;clc;
%rng(10072019);
%rng(91376491);
rng('default')
%rng(123545)
K = 5;%randi(16)+4;%number of classes
N_min = 1000;%Total sample size minimum
N_max = 500000;
%% GMM for sampling
mu = randi(20,K,1); %means
sigma = rand(1,1,K)/2+1; %variances
pStar = rand(1,K);
pStar = pStar./sum(pStar);%mixture coefficients
gm = gmdistribution(mu, sigma, pStar);%gmm
% figure
numiter = 50;
h = waitbar(0/numiter,sprintf('On iteration %d of %d',0,numiter));
for J=numiter:-1:1
    waitbar((numiter-J+1)/numiter,h,sprintf('On iteration %d of %d',(numiter-J+1),numiter))
    N = randi(N_max-N_min)+N_min;
    [X,T] = random(gm,N);%total sample

    F=zeros(K,N);
    for i=K:-1:1
        F(i,:)=normpdf(X,mu(i),sigma(1,1,i));
    end
    p_0 = ones(K,1)./K;
    [pHat,fullOrbit] = stablepoint(F,p_0,8,'diff',true);

    % For verification that the finite iterated method does the same as the
    % 'infinitely' iterated method.
    % [finitePi,orbitFinite] = iteratedSimplexMap(F,p_0,900);
    M = size(fullOrbit,2);
    distancesHilb = zeros(1,M);
    distancesEuc = zeros(1,M);
    errorsHilb = zeros(1,M-1);
    errorsEuc = zeros(1,M-1);
    for n=(M-1):-1:1
        distancesHilb(n) = hilbertDistSimplex(fullOrbit(:,n),fullOrbit(:,n+1),K);
        errorsHilb(n)= hilbertDistSimplex(fullOrbit(:,n),fullOrbit(:,M),K);
    end

    for n=(M-1):-1:1
        distancesEuc(n) = norm(fullOrbit(:,n)-fullOrbit(:,n+1));
        errorsEuc(n) = norm(fullOrbit(:,n)-fullOrbit(:,M));
    end


    % use the first 40% for 'break in' and do not fit to the regression
    L = floor(M*.4);
%     hold on
%     plot(log(distancesEuc(L:M-1)),'-')
%     plot(log(distancesHilb(L:M-1)),'--')
%     hold off
%     pause(.05)
%     modelstr = 'y~b1+b2/x';
%     opts = statset('MaxIter',M);
%     b0 = [-1,-1];
    expMdlEuc = fitlm((1:(M-L))',log(distancesEuc(L:end-1))',...
        'RobustOpts','on');
%         modelstr,b0,'Options',opts);
    expMdlHilb = fitlm((1:(M-L))',log(distancesHilb(L:end-1))',...
        'RobustOpts','on');
%         modelstr,b0,'Options',opts);
    rates.hilbDists{J} = distancesHilb;
    rates.hilbErr{J} = errorsHilb;

    rates.eucDists{J} = distancesEuc;
```

```matlab
70         rates.eucErrs{J} = errorsEuc;
71
72         rates.lambdaE(J)  = expMdlEuc.Coefficients.Estimate(2);
73         rates.lambdaH(J)  = expMdlHilb.Coefficients.Estimate(2);
74
75         rates.lambdaF(:,J) = svd(F);
76         rates.sz(J)  = M;
77         rates.samplesz(J) = N;
78  end
79  close(h);
80  euc = rates.lambdaE(:);
81  hilb = rates.lambdaH(:);
82  maxsvdF = 1./rates.lambdaF(1,:);
83  sizes = rates.samplesz(:);
84
85  figure
86  scatter(sizes,euc+maxsvdF',20,'b','o')
87  hold on
88  scatter(sizes,hilb+maxsvdF',20,'r','x')
89  hold off
90  title('$\sigma_n^{-1} - b_n$ vs. sample size $N_n$','Interpreter','latex')
91
92  figure
93  hold on
94  for ii=1:ceil(numiter/10)
95  plot(1:rates.sz(ii),log(rates.eucDists{ii}(:)));
96  end
97  title({'$\log(a_n)$ vs. $n$', 'Euclidean distance'},'Interpreter','latex')
98
99  figure
100 hold on
101 for ii=1:ceil(numiter/10)
102 plot(1:rates.sz(ii)-1,log(rates.eucErrs{ii}(:)));
103 end
104 title({'$\log(\epsilon_n)$ vs. $n$', 'Euclidean distance'},'Interpreter','latex')
```

Appendix B

Responsible Softmax Code

B.1 Responsible Softmax Layer

responsibilityLoss.m

```
1   classdef responsibilityLoss < nnet.layer.ClassificationLayer
2
3       properties
4           % (Optional) Layer properties
5           %A hard coded pi_0 version of this layer is in fixedRespLoss.m
6           piOper;
7           K;
8           err;
9           numIter;
10      end
11      methods
12          function layer = responsibilityLoss(numClasses, err, varargin)
13              % (Optional) Create a responsibilityLoss Layer
14              p = inputParser;
15              validScalarPosNum =@(x) isnumeric(x) && isscalar(x) && (x > 0);
16              validScalarPosInt =@(x) (int32(x)==x) && validScalarPosNum(x);
17              addRequired(p,'numClasses',validScalarPosInt);
18              addRequired(p,'err',validScalarPosNum);
19
20              C=numClasses;
21              defaultPi_0 = 1/C.*(ones(C,1));
22              validNonNegVec = @(x)(isnumeric(x)&&isvector(x)&&all(x>=0));
23              validRatio=@(x)((sum(x)-1)<1e-4) &&...
24                  (size(x,1)==C) &&...
25                  validNonNegVec(x);
26              addOptional(p,'ratios',defaultPi_0,validRatio);
27              addOptional(p,'iterations',1,validScalarPosInt)
28
29              parse(p,numClasses,err,varargin{:});
30
31              layer.Name = 'Responsibility Loss Layer';
32              layer.K = C;
33              layer.err = p.Results.err;
34              tol = tolCheckerHilb(10^(-err));
35              layer.piOper = responsibilityOperator(C,tol,...
36                  'ratios',p.Results.ratios);
37              layer.numIter = p.Results.iterations;
38          end
39
40          function loss = forwardLoss(layer, Y, T)
41              % Return the loss between the predictions Y and the
42              % training targets T
43              %
44              % Inputs:
45              % layer Output layer
46              % Y Predictions made by network
47              % T Training targets
48              %
49              % Output:
50              % loss - Loss between Y and T
51              U=Y.extractdata();
52              assert(all(U(:)>=0,'all'),"Y must be positive")
```

```matlab
53
54                %                  sprintf('Class of Y is %s.', class(Y))
55                %                  sprintf('Class of U is %s.', class(U))
56                errTol = eps(class(U));
57
58                %to resolve issue below.  Hopefully this doesn't run in layer
59                %training.
60                if ~isa(layer.piOper.pi_0,class(U))
61                    C= layer.K;
62                    layer.piOper.setPi_0(cast(ones(C,1)./C,'like',U));
63                end
64
65                N = size(T,4);
66                F = squeeze(Y);
67                T = squeeze(T);
68                piHat = layer.iteratedResponsibility(F);
69                %note: this is carrying over pi hats from previous runs, causes errors!
70                layer.piOper.setPi_0(piHat);
71                P = piHat'*F;
72                B = 1./(P+errTol);
73                V = piHat*B;
74                Z = V.*F;
75                loss = -sum(sum(T.*log(Z)))/N;
76                V = loss.extractdata();
77                if ~isa(V,class(U))
78                    disp(class(V))
79                    disp(class(U))
80                end
81            end
82        %                dF = dFhadamard + dFdot + dFpo;
83        %                dLdY = reshape(dF,szY);
84        %            end
85
86        function [pHat] = iteratedResponsibility(layer,F)
87            p = layer.piOper.pi_0;
88            new_p = responsibilityLoss.responsibilityMap(F,p);
89            for ii = 1:layer.numIter
90                p = new_p;
91                new_p = responsibilityLoss.responsibilityMap(F,p);
92            end
93            switch class(new_p)
94                case 'dlarray'
95                    u = new_p.extractdata();
96                    tol = max(eps(u));
97                otherwise
98                    tol = max(eps(new_p));
99            end
100
101
102            new_p(new_p<=tol) = new_p(new_p<=tol) + tol;%prevent overflow?
103            pHat=new_p./sum(new_p);%place it back in the right space.
104        end
105
106        function [pHat] = fixedResponsibility(layer,F)
107            stop = 2*10^(-layer.err);
108            p = layer.pi_0;
109            %i=0;
110
111            new = responsibilityLoss.responsibilityMap(F,p);
112            while (sum(abs(p-new))> stop*(1+norm(p)+max(eps(p)))) %%check stopping criterion?
113                %i=i+1;
114                p = new;
115                new = responsibilityLoss.responsibilityMap(F,p);
116            end
117            tol = abs(max(eps(new)));
118            new(new<=tol) = new(new<=tol) + tol;%prevent overflow?
119            pHat=new./sum(new);%place it back in the right space.
120        end
121    end
122    methods(Access=private,Static)
123
124        function [newP] = responsibilityMap(F,oldP)
```

```
125              % validateattributes(F,{'numeric'},{'>',0})
126              % validateattributes(oldP,{'numeric'},{'>=',0})
127              % assert(ismatrix(F),"F must be a K by N matrix")
128 %                sprintf('Class of oldP is %s.', class(oldP))
129              switch class(oldP)
130                  case 'dlarray'
131                      u = oldP.extractdata();
132                      err = eps(class(u));
133                  otherwise
134                      err = eps(class(oldP));
135              end
136              %msg =strcat("oldP must sum to 1: ",sprintf('%d, ',oldP));
137              %assert(abs(sum(oldP)-1)<=(2^10)*err,msg)
138              [k,N] = size(F);
139              assert(k == length(oldP),"F must have as many columns as oldP")
140
141              D = F'*oldP;
142              denoms = 1./(D+err);
143              newP = 1/N.*oldP.*(F*denoms);
144          end
145
146      function [dFpiHat] = dpiHatAdj(F, piHat, dpiHat)
147          K=size(F,1);
148          [Hl,dl]=lDifferentials(F,piHat);%%
149          dRdPi = Hl.*piHat+diag(dl);
150          V = (eye(K) - dRdPi)^-1;
151          dFpiHat = derivRFvecAdj(F,piHat,(V)'*dpiHat);%%
152      end
```

responsibilityOperator.m

```
1  classdef responsibilityOperator < handle
2     %RESPONSIBILITY OPERATOR is an obect that stores and updates class
3     %likelihoods using Dynamic responsibility.
4      properties
5          dim;
6          pi_0;
7          tolChecker;
8      end
9
10     methods
11         function obj = responsibilityOperator(dimension, tolerance, varargin)
12             %RESPONSIBIILTYOPERATOR Construct an instance of this class
13             %   PI_0, DIM, and TOLCHECKER must be set. PI_0 must be a DIM
14             %   by 1 vector, and the entries of PI_0 must sum to 1.
15             %   TOLCHECKER is used to determine convergence.
16             p = inputParser;
17             addRequired(p,'dimension',...
18                 @responsibilityOperator.validScalarPosInt);
19             addRequired(p,'tolerance',@responsibilityOperator.isTol);
20             C=dimension;
21             ratioCheck = @(x) responsibilityOperator.validRatio(x,C);
22             defaultPi_0 = (ones(C,1))./C;
23             addOptional(p,'ratios',defaultPi_0,ratioCheck);
24             parse(p,dimension,tolerance,varargin{:});
25
26             obj.dim = p.Results.dimension;
27             obj.pi_0 = p.Results.ratios;
28             obj.tolChecker = p.Results.tolerance;
29         end
30
31         function obj = setDim(obj,D)
32             %SETDIM checks that D is a positive integer then sets the
33             %dimension of OBJ to D.
34             if responsibilityOperator.validScalarPosInt(D)
35                 obj.dim = D;
36             else
37                 error('Dimension must be a positive integer')
38             end
39         end
```

```matlab
40
41          function obj = setPi_0(obj,V)
42              %SETPI_0 Checks that V is a valid mixing ratio, and sets
43              %OBJ.PI_0 to V if it is.
44              if responsibilityOperator.validRatio(V,obj.dim)
45                  obj.pi_0 = V;
46              else
47                  error('Mixing ratios must be positive reals summing to one')
48              end
49          end
50
51          function obj = setTolerance(obj,tol)
52              if responsibilityOperator.isTol(tol)
53                  obj.tolChecker = tol;
54              else
55                  error('%s %s. %s %s.',...
56                      'The input to setTolerance must be of type',...
57                      class(matlab.unittest.constraints.Tolerance),...
58                      'The variable tol is of type', class(tol));
59              end
60          end
61
62          function resp = iteratedResp(obj, F, p, n)
63              %ITERATEDRESP Takes parameters F, and starting point P. It
64              %returns RESP which is the first n points in the iteration of
65              %the responsibility map. i.e. first n elements of the orbit of
66              %PI_0
67              resp = zeros(obj.dim,n);
68              resp(:,1) = p;
69              for i=2:n+1
70                  resp(:,i) = obj.responsibilityMap(F,resp(:,i-1));
71              end
72          end
73
74          function resp = fixedResp(obj,F,p)
75              %FIXEDRESP is theoreticallty equivalent to
76              %obj.iteratedResp(F,p,Inf). It finds the fixed point of
77              %iterating responsibility. Stopping is determined by
78              %obj.tolChecker
79              tol = obj.tolChecker;
80              old_p = p;
81              new_p = obj.responsibilityMap(F,p);
82              resp = [old_p,new_p];
83              while ~tol.satisfiedBy(new_p,old_p)
84                  old_p = new_p;
85                  new_p = obj.responsibilityMap(F,old_p);
86                  resp = [resp,new_p]; %#ok<AGROW>
87              end
88          end
89
90          %the the following functions be static? Probably so!
91          function [dpResp,obj] = derivRp(obj,F,resp,dp_n)
92              [K,n] = size(resp); %The number of iterations done is n
93              assert(obj.dim == K,'Resp must have same rows as operator dim')
94              assert(sum(dp_n)<1e-4,'dp_n must sum to zero')
95              %dpResp = zeros(K,n);
96              dpResp(:,n) = dp_n;
97              for i=n-1:-1:1
98                  dpResp(:,i) = obj.dRdPiAdj(F,resp(:,i),dpResp(:,i+1));
99              end
100         end
101
102         function [dFResp, dpResp] = derivRFIter(obj,F,resp,dp_n)
103             dpResp = obj.derivRp(F,resp,dp_n);
104             n = size(resp,2); %The number of iterations done is n
105             dFResp = obj.derivRFvecAdj(F,resp(:,1),dpResp(:,2));
106             for i = 2:n-1
107                 dFResp = dFResp + obj.derivRFvecAdj(F,resp(:,1),dpResp(:,2));
108             end
109         end
110     end
111
```

```matlab
methods(Access = 'private', Static)
    function tf = validScalarPosNum(x)
        tf = isnumeric(x) && isscalar(x) && (x > 0);
    end

    function tf = validScalarPosInt(x)
        tf = (int32(x)==x) && responsibilityOperator.validScalarPosNum(x);
    end

    function tf = validNonNegVec(x)
        tf = (isnumeric(x)&&isvector(x)&&all(x>=0));
    end

    function tf = validRatio(x,dim)
        tf = (abs(sum(x)-1)<10^-4)...
            &&(size(x,1)==dim)&&responsibilityOperator.validNonNegVec(x);
    end

    function tf = isTol(x)
        import matlab.unittest.constraints.Tolerance
        tf = isa(x,'Tolerance');
    end

    function [newP] = responsibilityMap(F,oldP)
        %               replace with assert
        %               validateattributes(F,{'numeric'},{'>',0})
        %               validateattributes(oldP,{'numeric'},{'>=',0})
        assert(ismatrix(F),"F must be a K by N matrix")
        err = eps(class(oldP));
        %               msg =strcat("oldP must sum to 1: ",sprintf('%d, ',oldP));
        %               assert(abs(sum(oldP)-1)<=(2^14)*err,msg)
        [k,N] = size(F);
        assert(k == length(oldP),"F must have as many columns as oldP")

        D = F'*oldP;
        denoms = 1./(D+err);
        newP = 1/N.*oldP.*(F*denoms);
    end

    function [dFpiHat] = dpiHatAdj(F, piHat, dpiHat)
        K=size(F,1);
        [Hl,dl]=responsibilityOperator.lDiff(F,piHat);%%
        dRdPi = Hl.*piHat+diag(dl);
        V = (eye(K) - dRdPi);
        dFpiHat = responsibilityOperator.derivRFvecAdj(F,piHat,(V)')\dpiHat;%%
    end

    function [dPiAdj] = dRdPiAdj(F, p, dpi)
        [Hl,dl]=responsibilityOperator.lDiff(F,p);%%
        dRdPi = Hl.*p+diag(dl);
        dPiAdj = dRdPi'*dpi;
    end

    function [DFRhadj] = derivRFvecAdj(F,p,h)
        [K,N] = size(F);
        H = 1/N.*(h*ones(1,N));

        Pbar=1./(p'*F);

        DFRhadj = p.*H.*Pbar-(p*ones(1,K))*(p.*F.*(Pbar.^2).*(H));
    end

    function [Hl,dl] = lDiff(F,p)
        %LDifferentials calculates the gradient and Hessian of the averaged
        %log likelihood of a joint distribution of N samples from a
        %mixture of K different distributions.
        %   F is a K by N matrix, the evaluations of each point in the various
        %   mixture pdf's
        %   P is a K by 1 vector of the probability components. sum(P) = 1.
        N=size(F,2);

        %This has the effect of multiplying each row by the same entry of p, and then summing the
```

```
184            denoms=p'*F;%F is K by N, P is K by 1.
185            err = eps(class(F));
186            G=F./(denoms+err);%add err in case an entry of denoms is small.
187            dl=1/N.*sum(G,2);
188
189            %This comes directly from lemma 4.3 in dissertation
190            Hl=-1/N.*(G*G');
191         end
192      end
193 end
```

tolCheckerEuc.m

```
 1  classdef tolCheckerEuc < matlab.unittest.constraints.Tolerance
 2      %TOLCHECKEREUC Uses standard Euclidean distance to check if two
 3      %vectors in the probability simplex in R^N are within a set tolerance.
 4      %
 5
 6      properties
 7          value;
 8          diagON;
 9      end
10
11      methods
12          function obj = tolCheckerEuc(val, varargin)
13              %VAL gets assigned to the value of the tolerance checker if it
14              %is a positive numeric scalar.
15              %   TODO Add customization of distance function
16              p = inputParser;
17              diagDefault = false;
18              addRequired(p,'val',@tolCheckerEuc.checknum);
19              addOptional(p,'diagON',diagDefault);
20              parse(p,val,varargin{:});
21              obj.value = p.Results.val;
22              obj.diagON = p.Results.diagON;
23          end
24
25          function tf = supports(~,V)
26              tf = tolCheckerEuc.inSimplex(V);
27          end
28
29          function tf = satisfiedBy(tol, actual, expected)
30              if ~tol.supports(actual) || ~tol.supports(expected)
31                  tf = false;
32                  return
33              end
34              if length(actual) ~= length(expected)
35                  tf = false;
36                  return
37              end
38              dist = tolCheckerEuc.distance(actual,expected);
39              tf = (dist<=tol.value);
40              if ~tf && tol.diagON
41                  sprintf(tol.getDiagnosticFor(actual,expected))
42              end
43          end
44
45          function diag = getDiagnosticFor(tolerance, actual, expected)
46              import matlab.unittest.diagnostics.StringDiagnostic
47
48              if length(actual) ~= length(expected)
49                  str = 'Compared vectors must have the same dimension.';
50              else
51                  str = sprintf('%s%d.\n%s%d.', ...
52                      'The vectors have a Euclidean distance of ', ...
53                      distance(actual, expected), ...
54                      'The allowable distance is ', ...
55                      tolerance.value);
56              end
```

```
57                    diag = StringDiagnostic(str);
58              end
59
60        end
61
62        methods(Access = private, Static)
63
64              function tf = checknum(num)
65                    tf = false;
66                    if ~isscalar(num)
67                          error('Input is not scalar');
68                    elseif ~isnumeric(num)
69                          error('Input is not numeric');
70                    elseif (num <= 0)
71                          error('Input must be > 0 and <14');
72 %                     elseif (num~=int32(num))
73 %                         error('Input must be a whole number');
74                    else
75                          tf = true;
76                    end
77              end
78
79              function d = distance(U,V)
80                    if any(isnan(U))||any(isnan(V))
81                          error('Cannot compare two vectors containing NaN')
82                    else
83                          d = sqrt(norm(U-V));
84                    end
85              end
86
87              function tf = inSimplex(V)
88                    tf = (isnumeric(V)) && (isvector(V)) && (abs(sum(V)-1)<=1e-4);
89              end
90        end
91 end
```

B.2 Fixed Responsibility Softmax layer

fixedRespLoss.m

```
1  classdef fixedRespLoss < nnet.layer.ClassificationLayer
2
3      properties
4          % (Optional) Layer properties
5          pi_0;
6          K;
7          err;
8      end
9      methods
10         function layer = fixedRespLoss(numClasses, err, varargin)
11             % (Optional) Create a fixedRespLoss Layer
12             p = inputParser;
13             C = numClasses;
14             validScalarPosNum =@(x) isnumeric(x) && isscalar(x) && (x > 0);
15             validScalarPosInt =@(x) (int32(x)==x) && validScalarPosNum(x);
16             validNonNegVec = @(x)(isnumeric(x)&&isvector(x)&&all(x>=0));
17             validRatio=@(x)(abs(sum(x)-1)<10^-err)...
18                 &&(size(x,1)==C)&&validNonNegVec(x);
19             addRequired(p,'numClasses',validScalarPosInt);
20             addRequired(p,'err',validScalarPosNum);
21
22             defaultPi_0 = 1/C.*(ones(C,1));
23
24             addOptional(p,'ratios',defaultPi_0,validRatio);
25
```

```
26                parse(p,numClasses,err,varargin{:});
27
28                layer.Name = 'Responsibility Loss Layer';
29                layer.K = C;
30                layer.err = p.Results.err;
31                layer.pi_0 = p.Results.ratios;
32            end
33
34            function loss = forwardLoss(layer, Y, T)
35                % Return the loss between the predictions Y and the
36                % training targets T
37                %
38                % Inputs:
39                % layer   Output layer
40                % Y   Predictions made by network
41                % T   Training targets
42                %
43                % Output:
44                % loss   Loss between Y and T
45                assert(all(Y(:)>=0,'all'),"Y must be positive")
46                errTol = eps(class(Y));
47
48                N = size(T,4);
49                F = squeeze(Y);
50                T = squeeze(T);
51                piHat = layer.pi_0;
52                P = piHat'*F;
53                B = 1./(P+errTol);
54                V = piHat*B;
55                Z = V.*F;
56                loss = -sum(sum(T.*log(Z)))/N;
57            end
58
59            %% must implement backward when using full dynamic responsibility
60            function dLdY = backwardLoss(layer, Y, T)
61                % Backward propagate the derivative of the loss function
62                %
63                % Inputs:
64                % layer   Output layer
65                % Y   Predictions made by network
66                % T   Training targets
67                %
68                % Output:
69                % dLdY - Derivative of the loss with respect to the predictions Y
70
71                assert(all(Y(:)>=0,'all'),"Y must be positive")
72                errTol = eps(class(Y));
73                szY = size(Y);
74                N = size(T,4);
75                F = squeeze(Y);
76                T = squeeze(T);
77                piHat = layer.pi_0;
78                P = piHat'*F;
79                B = 1./(P+errTol);
80                V = piHat*B;
81                Z = V.*F;
82
83                dLdZ = -1/N*(T./(Z+errTol));
84
85                dFhadamard = V.*dLdZ;
86                dV = F.*dLdZ;
87                %dpiHatdot = dV*B';
88                dB = piHat'*dV;
89                dP = -dB.*B.^2;
90                dFdot = piHat*dP;
91                %dpiHatT = F*dP';
92                %dpiHat = dpiHatT + dpiHatdot;
93                %below might be a source of trouble!
94                %dFpo = responsibilityLoss.dpiHatAdj(F, piHat, dpiHat);
95                dF = dFhadamard + dFdot;% + dFpo;
96                dLdY = reshape(dF,szY);
97            end
```

```matlab
        function [pHat] = fixedResponsibility(layer,F)
            stop = 2*10^(-layer.err);
            p = layer.pi_0;
            %i=0;

            new = responsibilityLoss.responsibilityMap(F,p);
            while (sum(abs(p-new))> stop*(1+norm(p)+max(eps(p)))) %%check stopping criterion?
                %i=i+1;
                p = new;
                new = responsibilityLoss.responsibilityMap(F,p);
            end
            tol = abs(max(eps(new)));
            new(new<=tol) = new(new<=tol) + tol;%prevent overflow?
            pHat=new./sum(new);%place it back in the right space.
        end
    end
    methods(Access=private,Static)

        function [newP] = responsibilityMap(F,oldP)
            validateattributes(F,{'numeric'},{'>',0})
            validateattributes(oldP,{'numeric'},{'>=',0})
            assert(ismatrix(F),"F must be a K by N matrix")
            err = eps(class(oldP));
%            msg =strcat("oldP must sum to 1: ",sprintf('%d, ',oldP));
%            assert(abs(sum(oldP)-1)<=(2^10)*err,msg)
            [k,N] = size(F);
            assert(k == length(oldP),"F must have as many columns as oldP")

            D = F'*oldP;
            denoms = 1./(D+err);
            newP = 1/N.*oldP.*(F*denoms);
        end

        function [dFpiHat] = dpiHatAdj(F, piHat, dpiHat)
            K=size(F,1);
            [Hl,dl]=lDifferentials(F,piHat);%%
            dRdPi = Hl.*piHat+diag(dl);
            V = (eye(K) - dRdPi);
            dFpiHat = derivRFvecAdj(F,piHat,(V)'\dpiHat);%%
        end
        %% TODO: write a few more functions to keep dpiHatAdj contained
    end
end
```

Appendix C

Code for Examples on GMM Data

GMMoverlap.m

```matlab
1   clear;
2   clc;
3
4   rng(1085456391); %for testing
5   %rng('shuffle');
6   % scurr = rng;
7
8   K = 5;%randi(16)+4;%number of classes
9   N = 10000;%Total sample size
10  D = 2;
11  TestSz = .3; %size of verification sample
12
13  %% GMM for sampling
14  mu = [linspace(-K,K,K)*3.5;randi(4,1,K)-2]'; %means
15  density = 1;
16  rc = .9;
17  for k=K:-1:1
18      x = rand();
19      if mod(k,2) == 0
20          A = [0,-1;1,0];
21          evals = [5*x,x];
22      else
23  %        tmp = sprand(D,D,density,rc);
24          A = -eye(D);%[1,-1;1,1]./sqrt(2);%full(tmp);
25          evals = [19*x,x];
26      end
27
28      Sig = diag(evals);
29
30      sigma(:,:,k) = A*Sig*A' ; %variances
31  end
32
33  %adjust for specific example
34  mu(2,1) = -9;
35  mu(4,1) = 9;
36  sigma(:,:,4) = sigma(:,:,2);
37  sigma(:,:,5) = sigma(:,:,1);
38
39  idx = [1,3,5];%unique(randi(K,1,K-1));
40  pStar = rand(1,K);
41  pStar(idx) = pStar(idx)+10;
42
43  pStar = pStar./sum(pStar);%mixture coefficients
44  gm = gmdistribution(mu, sigma, pStar);%gmm
45
46  [X,T] = random(gm,N);%total sample
47
48  figure
49  scatter(X(:,1),X(:,2),10,T)
50
51  %% Layers to train
52
53  inputLayer = imageInputLayer([D 1 1], 'Normalization','none');
54  baseLayers = [inputLayer
55      fullyConnectedLayer(K*4)
56      tanhLayer
```

```matlab
57        fullyConnectedLayer(K)
58        softmaxLayer];
59
60   layers{1} = [baseLayers
61        classificationLayer];
62
63   layers{2} = [baseLayers
64        responsibilityLoss(K,4,'ratios',pStar')];
65
66   layers{3} = [baseLayers
67        responsibilityLoss(K,4,'iterations',4, 'ratios',pStar')];
68
69   layers{4} = [baseLayers
70        fixedRespLoss(K,4,'ratios',pStar')];
71
72   %% Filename info
73   % ensure working directory is parent of testresults before running!!
74   % alternatively, use what()
75   rng('shuffle');
76   scurr = rng;
77
78   file = what('data2');
79   dir = strcat(file.path,'\');%strcat(pwd,'\testResults\data2\');
80
81   accFile = strcat(dir,...
82        'acc_GMM_ovrlp_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
83   confusionFile = strcat(dir,...
84        'confusion_GMM_ovrlp_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
85   pctFile = strcat(dir,...
86        'tpr_GMM_ovrlp_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
87
88   %% Create Test and Validation Data
89
90   sz = [2,1];
91   numRuns = 1;
92   runSz = 1;
93   seeds = randi([1000000,90000000],1,numRuns*runSz);
94   W = waitbar(0,sprintf('Calculating Run %d of %d',1,numRuns));
95   for i=1:numRuns
96        % Prepare Training Data
97        [train_set,validate_set] =...
98            prepare_training_set(gm,N,TestSz,sz,'categorical');
99        %options setup
100       options = trainingOptions('adam', ...
101            'LearnRateSchedule','piecewise', ...
102            'LearnRateDropFactor',0.2, ...
103            'LearnRateDropPeriod',5, ...
104            'MaxEpochs',15, ...
105            'MiniBatchSize',100, ...
106            'Verbose',false, ...
107            'ValidationData',{validate_set.data,validate_set.targets});
108       %Create test set
109       rng('default')
110       [Y,C]=random(gm,N/4);
111       Ytest(:,:,1,:) = reshape(Y',D,1,[]);
112
113       waitbar(i/numRuns,W,sprintf('Calculating Sample %d of %d',i,numRuns))
114       H = waitbar(0,sprintf('Training set %d of %d',0,runSz));
115       tic
116       for j = 1:runSz
117            waitbar(j/runSz,H,sprintf('Training set %d of %d',j,runSz));
118            %Train Nets
119            [nets] = train_nets(layers,...
120                train_set.data,...
121                train_set.targets,...
122                options,...
123                seeds(j+runSz*(i-1)));
124            toc
125            %Test Nets
126            [acc,confMat,pcts] = test_nets(nets, Ytest, C);
127            %Write results to csv file
128            cellIdx = 1+runSz*K*(i-1)+K*(j-1);
```

```
129            cell = strcat('A',num2str(cellIdx));
130            accCell = strcat('A',num2str(j+runSz*(i-1)));
131            writematrix(acc,accFile,'Range',accCell)
132            writematrix(confMat,confusionFile,'Range',cell)
133            writematrix(pcts,pctFile,'Range',cell)
134        end
135        close(H)
136    end
137 close(W)
```

test_nets.m

```
1  function [acc,confMat,pcts] = ...
2      test_nets(nets, test_data, test_targets)
3  %TEST_NETS uses TEST_DATA to evaluate the performance of neural networks in
4  %   NETS via confusion matrices.  TEST_TARGETS are the target
5  %   classifications of TEST_DATA.
6  S = length(nets);
7  targets = full(ind2vec(test_targets'));
8
9  for i = S:-1:1
10     outputs = zeros(size(targets));
11     [~,classHat{i}]= max(nets{i}.predict(test_data),[],2);
12     outputs = full(ind2vec(classHat{i}'));
13     if ~all(size(outputs) == size(targets))
14         o = size(outputs);
15         t = size(targets);
16         if (o(1) < t(1))%it should be the case that o(1)<=t(1)
17             pad = zeros(t(1)-o(1),o(2));
18             outputs = [outputs;pad];
19         end
20         if(o(2)~=t(2))
21             error('incompatible target and output data size');
22         end
23     end
24     [c(i),cm(:,:,i),~,per(:,:,i)] = confusion(targets,outputs);
25 end
26  acc = c;
27  confMat = cm;
28  pcts = per;
29 end
```

train_nets.m

```
1  function [nets] = train_nets(layers,data,targets,options,seed)
2  %TRAIN_NETS trains the neural networks represented by LAYERS using DATA for
3  %training. OPTIONS should contain validation data if desired. SEED is used
4  %for initiating neural net weights in the same manner for each training
5  %run.
6  S = length(layers);
7  for ii = S:-1:1
8      rng(seed);
9      nets{ii} = trainNetwork(data, targets, layers{ii}, options);
10 end
11
12 end
```

Appendix D

CODE FOR EXAMPLE ON MNIST DATA

respMNISTimba.m

```matlab
1   clear; clc;
2   %% Load MNIST data
3   %get file structure for prepared MNIST data
4   s = what('MNIST');
5   filename = fullfile(s.path,'\','MNISTdata.mat');
6
7   load(filename)
8
9   K=size(unique(Labels),1);
10  [N,w,h] = size(images2d);
11
12  %% Set PARAMETERS up for training
13  testSz = .33; %size of verification sample
14
15  %rng('shuffle');
16  rng(1218763585);
17  scurr = rng;
18  %run 1   5 feb 2020:[10,10,10,1,10,5,10,10,10,5];
19  %run 2/3   6 feb 2020:[1,7,10,1,10,5,2,8,5,4];
20  ratios = [0,30.1,17.6,12.5,9.7,7.9,6.7,5.8,5.1,4.6];%benford's law;
21  ratiosFull = ratios./sum(ratios);
22  if sum(ratios>0)~=K
23      K=sum(ratios>0);
24      ratios = ratiosFull(ratios>0);
25  end
26
27  %% Neural Network Layers
28  %layer setup
29  %possible different input layer
30  inputLayer = imageInputLayer([w h 1], 'Normalization','none');
31  baseLayers = [inputLayer
32      convolution2dLayer(5,11)
33      tanhLayer
34      maxPooling2dLayer(2,'Stride',3)
35      fullyConnectedLayer(K)
36      softmaxLayer];
37
38  layers{1} = [baseLayers
39      classificationLayer];
40
41  layers{2} = [baseLayers
42      responsibilityLoss(K,4,'ratios',ratios')];
43
44  layers{3} = [baseLayers
45      responsibilityLoss(K,4,'iterations',4, 'ratios',ratios')];
46
47  layers{4} = [baseLayers
48      fixedRespLoss(K,4,'ratios',ratios')];
49
50  %% Filename info
51  % ensure working directory is parent of testresults before running!!
52  % alternatively, use what()
53  file = what('data2');
54  dir = strcat(file.path,'\');%strcat(pwd,'\testResults\data2\');
55
56  accFile = strcat(dir,...
```

```
57          'acc_imba_MNIST_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
58   confusionFile = strcat(dir,...
59          'confusion_imba_MNIST_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
60   pctFile = strcat(dir,...
61          'tpr_fpr_imba_MNIST_',num2str(K),'_',num2str(scurr.Seed),'.xlsx');
62
63   numRuns = 3;
64   runSz = 40;
65   seeds = randi([1000000,90000000],1,numRuns*runSz);
66   W = waitbar(0,sprintf('Calculating Run %d of %d',1,numRuns));
67   for i=1:numRuns
68       % Establish ratios
69       rng('default')
70       % Prepare Training Data
71       [train_set,validate_set] =...
72           unbalanced_MNIST(Labels,Images,40000,testSz,ratiosFull,w,h);
73       %options setup
74       options = trainingOptions('adam', ...
75           'LearnRateSchedule','piecewise', ...
76           'LearnRateDropFactor',0.2, ...
77           'LearnRateDropPeriod',5, ...
78           'MaxEpochs',5, ...
79           'MiniBatchSize',100, ...
80           'Verbose',false, ...
81           'ValidationData',{validate_set.data,validate_set.targets});
82       %Create test set
83       [test1,test2] =  unbalanced_MNIST(TestLabels,TestImages,...
84                                         6600,.5,ratiosFull,w,h);
85
86       waitbar(i/numRuns,W,sprintf('Calculating Sample %d of %d',i,numRuns))
87       H = waitbar(0,sprintf('Training set %d of %d',0,runSz));
88       tic
89       for j = 1:runSz
90           waitbar(j/runSz,H,sprintf('Training set %d of %d',j,runSz));
91           %Train Nets
92           [nets] = train_nets(layers,...
93               train_set.data,...
94               train_set.targets,...
95               options,...
96               seeds(j+runSz*(i-1)));
97           toc
98           %Test Nets
99           [acc,confMat,pcts] = test_nets(nets,...
100                                          test1.data,...
101                                          double(string(test1.targets)));%+1);
102                                      %only use +1 for datasets with all 10
103                                      %digits
104           [acc2,confMat2,pcts2] = test_nets(nets,...
105                                          test2.data,...
106                                          double(string(test2.targets)));%+1);
107           %Write results to csv file
108           cellIdx = 1+runSz*K*(i-1)+K*(j-1);
109           cell = strcat('A',num2str(cellIdx));
110           accCell = strcat('A',num2str(j+runSz*(i-1)));
111           writematrix([acc,acc2],accFile,'Range',accCell)
112           writematrix([confMat,confMat2],confusionFile,'Range',cell)
113           writematrix([pcts,pcts2],pctFile,'Range',cell)
114       end
115       close(H)
116   end
117   close(W)
```

unbalanced_MNIST.m

```
1   function [train_set,validate_set] = ...
2       unbalanced_MNIST(labels,data,subsamplesize,pct,ratios,W,H)
3   %UNBALANCED_MNIST Creates training and validation sets from the MNIST data
4   %  set. The sets produced have samples reweighted according to the convex
5   %  mixture given in RATIOS.
6   numSamples = length(labels);
```

```matlab
 7  weights = zeros(numSamples,1);
 8  for i=1:numSamples
 9      weights(i) = ratios(labels(i)+1);
10  end
11
12  [labelsub,subIdx] = datasample(labels,subsamplesize,'Weights',weights);
13  datasub = data(subIdx,:);
14
15  c = cvpartition(labelsub,"HoldOut",pct);
16
17  testIdx = test(c);
18  trainIdx = training(c);
19
20  labelTest = labelsub(testIdx);
21  dataTest = datasub(testIdx,:);
22
23  labelTrain = labelsub(trainIdx);
24  dataTrain = datasub(trainIdx,:);
25
26  train_set.targets = categorical(labelTrain);
27  validate_set.targets = categorical(labelTest);
28
29  train_set.data(:,:,1,:) = reshape(dataTrain',W,H,[]);
30  validate_set.data(:,:,1,:) = reshape(dataTest',W,H,[]);
31
32  end
```

References

[AR67] Ralph Abraham and Joel Robbin, *Transversal mappings and flows*, WA Benjamin New York, 1967.

[Ben38] Frank Benford, *The law of anomalous numbers*, Proceedings of the American Philosophical Society **78** (1938), no. 4, 551–572.

[BHH19] Pierre Blanchard, Desmond J. Higham, and Nicholas J. Higham, *Accurate computation of the log-sum-exp and softmax functions*, arXiv **arXiv:1909.03469** (2019).

[Bis95] Christopher M. Bishop, *Neural networks for pattern recognition*, Oxford University Press, Inc., New York, NY, USA, 1995.

[Bis06] Christopher M. Bishop, *Pattern recognition and machine learning*, 1st ed. 2006. corr. 2nd printing ed., Information science and statistics, Springer, 2006.

[BPM04] Gustavo E. A. P. A. Batista, Ronaldo C. Prati, and Maria Carolina Monard, *A study of the behavior of several methods for balancing machine learning training data*, SIGKDD Explor. Newsl. **6** (2004), no. 1, 20–29.

[BPRS17] Atılım Güneş Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind, *Automatic differentiation in machine learning: a survey*, The Journal of Machine Learning Research **18** (2017), no. 1, 5595–5637.

[BPSW70] Leonard E. Baum, Ted Petrie, George Soules, and Norman Weiss, *A maximization technique occurring in the statistical analysis of probabilistic functions of Markov chains*, Ann. Math. Statist. **41** (1970), no. 1, 164–171.

[CS96] Peter Cheeseman and John Stutz, *Bayesian classification (AutoClass): Theory and results*, Advances in Knowledge Discovery and Data Mining (Usama M. Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramasamy Uthurusamy, eds.), American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996, pp. 153–180.

[Dev89] R.L. Devaney, *An introduction to chaotic dynamical systems*, Addison-Wesley advanced book program, Addison-Wesley, 1989.

[DFO20]   Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong, *Mathematics for machine learning*, Cambridge University Press, 2020.

[DLR77]   A. P. Dempster, N. M. Laird, and D. B. Rubin, *Maximum likelihood from incomplete data via the EM algorithm*, JOURNAL OF THE ROYAL STATISTICAL SOCIETY, SERIES B **39** (1977), no. 1, 1–38.

[EUD17]   Stefan Elfwing, Eiji Uchibe, and Kenji Doya, *Sigmoid-weighted linear units for neural network function approximation in reinforcement learning*, CoRR **arxiv:1702.03118** (2017).

[FMDFS17]   Alhussein Fawzi, Seyed-Mohsen Moosavi-Dezfooli, Pascal Frossard, and Stefano Soatto, *Classification regions of deep neural networks*, arXiv preprint **arXiv:1705.09552** (2017).

[Fri12]   Greg Friedman, *Survey article: an elementary illustrated introduction to simplicial sets*, The Rocky Mountain Journal of Mathematics (2012), 353–423.

[GBC16]   Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep learning*, MIT Press, 2016, `http://www.deeplearningbook.org`.

[GFG06]   Alex Graves, Santiago Fernández, and Faustino Gomez, *Connectionist temporal classification: Labelling unsegmented sequence data with recurrent neural networks*, Proceedings of the International Conference on Machine Learning, ICML 2006, 2006, pp. 369–376.

[GW08]   Andreas Griewank and Andrea Walther, *Evaluating derivatives*, second ed., Society for Industrial and Applied Mathematics, 2008.

[Har58]   H. O. Hartley, *Maximum likelihood estimation from incomplete data*, Biometrics **14** (1958), no. 2, 174–194.

[Hor91]   Kurt Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural Networks **4** (1991), no. 2, 251 – 257.

[HTF09]   Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning: data mining, inference and prediction*, 2 ed., Springer, 2009.

[JJNH91]   Robert A. Jacobs, Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton, *Adaptive mixtures of local experts*, Neural Comput. **3** (1991), no. 1, 79–87.

[KPC02] S.G. Krantz, H.R. Parks, and J.R. Cnops, *The implicit function theorem: History, theory, and applications*, Modern Birkhäuser classics, Birkhäuser, 2002.

[KW13] Diederik P Kingma and Max Welling, *Auto-encoding variational Bayes*, arXiv preprint **arXiv:1312.6114** (2013).

[LBB+12] Steve Lawrence, Ian Burns, Andrew Back, Ah Chung Tsoi, and C. Lee Giles, *Neural network classification and prior class probabilities*, Neural Networks: Tricks of the Trade: Second Edition (Grégoire Montavon, Geneviève B. Orr, and Klaus-Robert Müller, eds.), Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 295–309.

[LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner, *Gradient-based learning applied to document recognition*, Proceedings of the IEEE **86** (1998), no. 11, 2278–2324.

[LBH15] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, *Deep learning*, Nature **521** (2015), no. 7553, 436–444.

[Lin76] Seppo Linnainmaa, *Taylor expansion of the accumulated rounding error*, BIT Numerical Mathematics **16** (1976), no. 2, 146–160.

[Llo82] Stuart P. Lloyd, *Least squares quantization in PCM*, IEEE Trans. Information Theory **28** (1982), no. 2, 129–136.

[LLPS93] 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** (1993), no. 6, 861 – 867.

[LS76] J. P. La Salle, *The stability of dynamical systems*, Society for Industrial and Applied Mathematics, 1976.

[LWYY16] Weiyang Liu, Yandong Wen, Zhiding Yu, and Meng Yang, *Large-margin softmax loss for convolutional neural networks.*, Proceedings of the International Conference on Machine Learning, ICML, vol. 2, 2016, p. 7.

[Mac67] J. MacQueen, *Some methods for classification and analysis of multivariate observations*, Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Statistics (Berkeley, Calif.), University of California Press, 1967, pp. 281–297.

[Mac02]   David J. C. MacKay, *Information theory, inference & learning algorithms*, Cambridge University Press, New York, NY, USA, 2002.

[Man12]   Jonathan H. Manton, *Differential calculus, tensor products and the importance of notation*, arXiv preprint **arXiv:1208.0197** (2012).

[Mat20]   The Mathworks, Inc., Natick, Massachusetts, *MATLAB version 9.8.0.1359463 (R2020a)*, 2020.

[MB17]   Wenjun Mei and Francesco Bullo, *Lasalle invariance principle for discrete-time dynamical systems: A concise and self-contained tutorial*, arXiv preprint **arXiv:1710.03710** (2017).

[MHL15]   A.N. Michel, L. Hou, and D. Liu, *Stability of dynamical systems: On the role of monotonic and non-monotonic Lyapunov functions*, Systems & Control: Foundations & Applications, Springer International Publishing, 2015.

[MKH19]   Rafael Müller, Simon Kornblith, and Geoffrey E. Hinton, *When does label smoothing help?*, CoRr **arXiv:1906.02629** (2019).

[MN85]   JR Magnus and H Neudecker, *Matrix differential calculus with applications to simple, Hadamard, and Kronecker products*, Journal of Mathematical Psychology **29** (1985), no. 4, 474–492.

[MP43]   W S McCulloch and W Pitts, *A logical calculus of the ideas immanent in nervous activity*, Bull. Math. Biophys. **5** (1943), 115–133.

[MP90]   Marvin Lee Minsky and Seymour Papert, *Perceptrons: An introduction to computational geometry*, Massachusetts Institute of Technology, Cambridge, Mass, 1990 (English).

[MVKO18]   Ashok Vardhan Makkuva, Pramod Viswanath, Sreeram Kannan, and Sewoong Oh, *Breaking the gridlock in mixture-of-experts: Consistent and efficient algorithms*, Proceedings of the International Conference on Machine Learning, ICML 2018, 2018.

[Nes03]   Y. Nesterov, *Introductory lectures on convex optimization: A basic course*, Applied Optimization, vol. 87, Springer US, 2003.

[NH98] Radford M. Neal and Geoffrey E. Hinton, *A view of the EM algorithm that justifies incremental, sparse, and other variants*, Learning in Graphical Models (Michael I. Jordan, ed.), Springer Netherlands, Dordrecht, 1998, pp. 355–368.

[nLa20a] nLab authors, *simplex*, `http://ncatlab.org/nlab/show/simplex`, May 2020, Revision 33.

[nLa20b] ——, *simplex category*, `http://ncatlab.org/nlab/show/simplex%20category`, May 2020, Revision 68.

[Pol81] David Pollard, *Strong consistency of k-means clustering*, Ann. Statist. **9** (1981), no. 1, 135–140.

[Pol82] ——, *A central limit theorem for k-means clustering*, Ann. Probab. **10** (1982), no. 4, 919–926.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams, *Learning representations by back-propagating errors*, Nature **323** (1986), no. 6088, 533–536.

[Ros58] F. Rosenblatt, *The perceptron: A probabilistic model for information storage and organization in the brain*, Psychological Review (1958), 65–386.

[Ryc16] Marek Rychlik, *Course notes*, http://alamos.math.arizona.edu/math577/fall16/, 2016.

[Ryc19] M. Rychlik, *A proof of convergence of multi-class logistic regression network*, arXiv e-prints **arXiv:1903.12600** (2019).

[Ryc20] Marek Rychlik, *Personal communication*, 2020.

[SC96] John Stutz and Peter Cheeseman, *Autoclass — a Bayesian approach to classification*, Maximum Entropy and Bayesian Methods (Dordrecht) (John Skilling and Sibusiso Sibisi, eds.), Springer Netherlands, 1996, pp. 117–126.

[Sch15] J. Schmidhuber, *Deep learning in neural networks: An overview*, Neural Networks **61** (2015), 85–117, Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[Sun74] Rolf Sundberg, *Maximum likelihood theory for incomplete data from an exponential family*, Scandinavian Journal of Statistics **1** (1974), no. 2, 49–58.

[Sun76] _____, *An iterative method for solution of the likelihood equations for incomplete data from exponential families*, Communications in Statistics - Simulation and Computation **5** (1976), no. 1, 55–64.

[The05] Fabian Theis, *Gradients on matrix manifolds and their chain rule*, Neural Information Processing LR **9** (2005), 1–13.

[vdOV⁺17] Aaron van den Oord, Oriol Vinyals, et al., *Neural discrete representation learning*, Advances in Neural Information Processing Systems, 2017, pp. 6306–6315.

[Wal49] Abraham Wald, *Note on the consistency of the maximum likelihood estimate*, The Annals of Mathematical Statistics **20** (1949), no. 4, 595–601.

[Wer94] P.J. Werbos, *The roots of backpropagation: From ordered derivatives to neural networks and political forecasting*, Adaptive and Cognitive Dynamic Systems: Signal Processing, Learning, Communications and Control, Wiley, 1994.

[Woo70] Max A Woodbury, *A missing information principle: theory and applications*, Tech. report, Duke University Medical Center Durham United States, 1970.

[XWC09] H. Xiong, J. Wu, and J. Chen, *K-means clustering versus validation measures: A data-distribution perspective*, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) **39** (2009), no. 2, 318–331.