Research Commons

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Learning Discrete and Lipschitz Representations

A thesis

submitted in fulfilment

of the requirements for the Degree

of

Doctor of Philosophy in Computer Science

at

The University of Waikato

by

Henry Gouk

THE UNIVERSITY OF

WAIKATO

Te Whare Wānanga o Waikato

2019

Abstract

Learning to embed data into a low dimensional vector space that is more useful for some downstream task is one of the most common problems addressed in the representation learning literature. Conventional approaches to solving this problem typically rely on training neural networks using labelled training data. In order to construct an accurate embedding function that will generalise to data not seen during training, one must either gather a very large training dataset, or adequately bias the learning process. This thesis focuses on the task of incorporating new inductive biases into the representation learning paradigm by constraining the set of functions that a learned feature extractor can come from.

The first part of this thesis investigates how one can learn a mapping that changes slowly with respect to its input. This is first addressed by deriving the Lipschitz constant of common feed-forward neural network architectures, and subsequently demonstrating how this constant can be constrained during training. Following this, it is investigated how a similar goal can be accomplished when one assumes that the inputs of interest lie near a low dimensional manifold embedded in a high dimensional vector space. This results in an algorithm that takes advantage of an empirical analog to the Lipschitz constant. Experimental results show that these methods have favourable performance compared to other methods commonly used for imposing inductive biases on neural network learning algorithms.

In the second part of this thesis, methods for extracting representations using decision tree models are developed. The first method presented is a problem transformation approach that allows one to reuse existing tree induction techniques. The second approach shows how one can incrementally construct decision trees using gradient information as the source of supervision, allowing one to use an ensemble of decision trees as a layer in a neural network. The experimental results indicate that these approaches improve the performance of representation learning on tabular data across multiple tasks.

Acknowledgements

Firstly, I would like to thank my supervisors: Bernhard Pfahringer, Eibe Frank, and Michael Cree. Throughout my time as a PhD student they have encouraged me to explore the topics that interest me most, while also providing useful feedback and guidance. I am appreciative of Bernhard and Michael for taking me on as a student, even though my initial topic was not aligned with their usual research interests. Although Eibe was a late addition to the supervisory panel, his attention to detail has been an invaluable asset throughout various paper and thesis writing endeavours.

I am grateful to have been around many interesting people from both the Machine Learning group in the Department of Computer Science, and also the Computer Vision group based in the School of Engineering. I have had many interesting discussions with people from these groups, particularly Steven Lang, Chen Zheng, Vithya Yogarajan, Felipe Bravo Marquez, Bob Durrant, Peter Reutemann, Lee Streeter, and John Perrone. A special thank you goes to Rory Mitchell, for the many thought-provoking conversations, and for giving me the chance to get a taste of what industry is like.

I am thankful to Anthony Blake for providing me with an opportunity to be involved in several research projects as an undergraduate, and also for supervising my honours project. Working with him is what sparked my interest in pursuing a career in academia.

The Department of Computer Science has been a wonderful environment to work in, both as a student and occasionally as a staff member, and I would like to extend my gratitude to all those who have made my time there an enjoyable period of my life.

I would like to thank Waikato Ultimate and the Waikato Youth Symphonic Band. Being involved in these communities has provided a me with a much needed work–life balance and resulted in long term friendships.

Lastly, I would like to thank my family and other friends for the encouragement and support throughout my studies, for asking me how my research was going when I first started my PhD, and for not asking as I drew closer to finishing.

Contents

1 Introduction **2**

1.1 Representation Learning . . . 5
1.2 Inductive Bias . . . 8
1.3 Discrete and Lipschitz Hypothesis Classes . . . 10
1.4 Contributions and Thesis Organisation . . . 11

2 Inductive Bias in Representation Learning **13**

2.1 Neural Network Building Blocks . . . 14
2.2 Network Architectures . . . 17
2.3 Optimisation Methods . . . 20
2.3.1 Empirical Risk Minimisation . . . 20
2.3.2 Batch Optimisation . . . 21
2.3.3 Stochastic Optimisation . . . 23
2.4 Regularising Neural Networks . . . 26
2.5 Induction of Decision Trees . . . 30
2.6 Learning Representations with Trees . . . 32

3 The Lipschitz Constant of Neural Networks **35**

3.1 Computing the Lipschitz Constant . . . 36
3.1.1 Fully Connected Layers . . . 37
3.1.2 Convolutional Layers . . . 38
3.1.3 Pooling Layers and Activation Functions . . . 42
3.1.4 Residual Connections . . . 42
3.1.5 Batch Normalisation . . . 44
3.1.6 Dropout . . . 44

4 Constraining the Lipschitz Constant of Neural Networks **46**

4.1 Enforcing the Constraint . . . 47
4.2 Experiments . . . 50
4.2.1 Synthetic Data . . . 51
4.2.2 CIFAR-10 . . . 52
4.2.3 CIFAR-100 . . . 56

4.2.4 MNIST and Fashion-MNIST 58
4.2.5 Street View House Numbers 59
4.2.6 Scaled ImageNet Subset (SINS-10) 61
4.2.7 Fully Connected Networks 64
4.2.8 Sensitivity to λ 66
4.2.9 Sample Efficiency 70
4.3 Conclusion 72

5 MaxGain Regularisation of Neural Networks 74
5.1 Regularisation by Constraining Gain 75
5.1.1 MaxGain Regularisation 77
5.1.2 Compatibility with Dropout 80
5.2 Experiments 80
5.2.1 CIFAR-10 81
5.2.2 CIFAR-100 82
5.2.3 Street View House Numbers (SVHN) 82
5.2.4 SINS-10 83
5.2.5 Gain on the Test Set 84
5.2.6 Sensitivity to γ 86
5.3 Discussion 87

6 A Problem Transformation Approach to Embedding 88
6.1 Background 91
6.2 Learning Linear Metrics 93
6.3 Extension to Nonlinear Models 97
6.4 Experiments 99
6.4.1 Classification 99
6.4.2 Target Vector Size 102
6.4.3 Visualisation 104
6.5 Conclusion 105

7 Online Learning of Discrete Representations 109
7.1 Related Work 111
7.2 Stochastic Gradient Trees 112
7.2.1 Leveraging Gradient Information 113
7.2.2 Splitting on Numeric Attributes 116
7.2.3 Determining when to Split 117
7.3 Ensembles of Stochastic Gradient Trees 119
7.3.1 Training Committees of SGTs 120
7.3.2 Training SGT Networks 120
7.4 Experiments 122

7.4.1 Incremental Learning 123
7.4.2 Batch Learning 126
7.5 Conclusion 130

8 Conclusion 132
8.1 Future Research Directions 134

List of Figures

4.1 Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_1 norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network. 53

4.2 Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_2 norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network. 54

4.3 Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_∞ norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network. 55

4.4 A critical difference diagram showing the statistically significant (95% confidence) differences between the average rank of each method. The number beside each method is the average rank of that method across all datasets. The thick black bars overlaid on groups of thin black lines indicate a clique of methods that have not been found to be statistically significantly different. . 68

4.5 This figure demonstrates the sensitivity of the algorithm to the choice of λ for each of the three p-norms when used to regularise VGG19 networks trained on the CIFAR-100 dataset. Because a different hyperparameter was optimised for each layer type, the horizontal axis represents the value of a single constant that is used to scale the three different λ hyperparameters associated with each curve. Note that when $c = 0.6$, the LCC-ℓ_1 network fails to converge. 69

4.6 Learning curves for VGG-style networks trained with each of the regularisation methods. 71

4.7 Learning curves for wide residual networks trained with each of the regularisation methods. Note that batch normalisation is used when training all of these models. 71

5.1 Boxplots showing the distributions of gains measured on each layer of the MaxGain-regularised VGG-19 network trained on CIFAR-10. The top plot shows the distributions on the training set, and the bottom plot on the test set. 85

5.2 Boxplots showing the distributions of gains measured on each layer of the unregularised VGG-19 network trained on CIFAR-10. The top plot shows the distributions on the training set, and the bottom plot on the test set. 86

5.3 Accuracy (left) and log loss (right) of the VGG-style model on both the train and test splits of the SVHN dataset as the γ hyperparameter is varied. The legend is shared between both plots. 87

6.1 This diagram provides an overview of how nonlinear metrics are constructed using the proposed method. The first step in the process is to learn the target vectors for which squared Euclidean distance is an accurate estimate of the Jaccard distance over the original label sets. Once the target vectors have been learned, they, along with the original features, are used to create a multi-target regression model. We apply a separate random forest for each target vector component. Once the multi-target regression model has been learned, it can be used to embed the features into the same space as the target vectors. Finally, the embeddeds and the original labels can be used to construct a k-NN classifier. 99

6.2 A demonstration of the impact that the target vector size has on the performance of the nonlinear models, as measured by several standard multi-label evaluation measures. The same legend applies to all plots. 104

6.3 Visualisation of a random sample of the scene dataset. Each plot indicates the presence (red) or absence (blue) of a label for each instance. 106

6.4 Visualisation of the emotions dataset. Each plot indicates the presence (red) or absence (blue) of a label for each instance. . 107

7.1 Learning curves for the incremental classification problems. . . 124

7.2 Learning curves for the incremental regression problems. . . . 125

List of Tables

4.1 Performance of VGG19 networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-10. LCC-ℓ_p denotes the Lipschitz Constant Constraint method for a given p-norm. 57
4.2 Accuracy of Wide Residual Networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-10. All WRNs are trained with batch normalisation. 58
4.3 Performance of networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-100. LCC-ℓ_p denotes our Lipschitz Constant Constraint method for some given p-norm. 59
4.4 Accuracy of Wide Residual Networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-100. All WRNs are trained with batch normalisation. 60
4.5 Test accuracies of the small convolutional networks trained with spectral decay, dropout, LCC, and combinations thereof on the MNIST and Fashion-MNSIT datasets. 61
4.6 Prediction accuracy of VGG-style networks trained with spectral decay, dropout, LCC, and combinations thereof on the SVHN dataset. 62
4.7 Prediction accuracies of WRN-16-4 networks trained with spectral decay, dropout, LCC, and combinations thereof on the SVHN dataset. 63

4.8 Prediction accuracies of VGG-style networks trained with spectral decay, dropout, batchnorm, LCC, and combinations thereof on the SINS-10 dataset. The +/-column indicates whether adding LCC to the combination of regularisers results in a statistically significant improvement in performance at the 95% confidence level. 64
4.9 Prediction accuracies of WRNs trained with spectral decay, dropout, batchnorm, LCC, and combinations thereof on the SINS-10 dataset. The +/-column indicates whether adding LCC to the combination of regularisers results in a statistically significant improvement in performance at the 95% confidence level. 65
4.10 Mean test set accuracies obtained using two repetitions of 5-fold cross validation. Statistically significant improvements and degredations (95% confidence) are marked with the +and -symbols, respectively. The highest mean accuracy achieved on each dataset is bolded. 67

5.1 Accuracy of a VGG-19 network trained in CIFAR-10 with different regularisation techniques. 81
5.2 Accuracy of a Wide Residual Network with a depth of 16 and a width factor of four trained on CIFAR-100 with different regularisation techniques. 82
5.3 Accuracy of a VGG-style network on the SVHN dataset when trained with various regularisation techniques. 83
5.4 Performance of the Wide Residual Network on the Scaled ImageNet Subset dataset using various combinations of regularisation techniques. The figures in this table are the mean accuracy $\pm$ the standard error, as measured across the 10 different folds. 84

6.1 A summary of the datasets that we use in our experiments. The label cardinality is the average number of labels assigned to each example. 100
6.2 Multi-label classification performance. LJE indicates the linear Jaccard Embedding method. 101
6.3 Multi-label classification performance. NJE indicates the non-linear Jaccard Embedding method. 103

7.1 Details of the classification and regression datasets used for evaluating incremental decision tree learners. No entry in the # Classes column indicates that the dataset is associated with a regression task. 123
7.2 Mean classification error, model size (number of nodes), and runtime (seconds) of the trees produced by the classification methods on 10 random shuffles of each dataset. 123
7.3 Mean absolute error, model size (number of nodes), and runtime (seconds) of the trees produced by the regression methods on 10 random shuffles of each dataset. 126
7.4 10-fold cross validation results for a collection of datasets found on OpenML (Vanschoren et al., 2013). 127
7.5 Results of 10-fold cross validation on multi-label datasets. . . . 128

Chapter 1

Introduction

Recent years have seen a boom in both artificial intelligence research and industry uptake. Participation in academic conferences has surged, and many large corporations have made huge investments in long-term artificial intelligence projects and endeavours. The majority of problems tackled with artificial intelligence techniques require a solution based on supervised machine learning. Systems that perform supervised learning take a training set of instances—or examples—as input, and produce a predictive model as output. Normally, each of the instances in the training set is a vector of features, sometimes referred to as attributes, and is associated with a label indicating what the model should predict if it receives that instance as an input. The goal of a supervised learning algorithm is to find a model that will generalise to instances that were not seen during the training process.

Much of the hype surrounding machine learning at the time of writing this thesis can be attributed to the successes of a family of techniques known as deep learning, which can be thought of as a rebranding of techniques for training artificial neural networks, albeit with much greater complexity than networks considered previously. These approaches to machine learning are notable because of their ability to solve tasks that not so long ago seemed insurmountable. We now have the ability to construct systems that can recognise thousands of every day objects in photos (Russakovsky et al., 2015) and

build smartphone applications that are capable of performing robust speech recognition in real time (Deng et al., 2013). Previous attempts at solving these tasks relied on hand-engineered functions that generate features from an input signal, such as the SIFT features commonly extracted from images (Lowe, 2004) and the Mel-frequency Cepstral Coefficients used when analysing speech data (Ganchev et al., 2005). Deep neural networks are distinguished by how they have been designed to learn how to extract features from the raw signals, rather than using a predefined feature extraction technique.

The core building blocks of deep learning have been around for a long time. Convolutional neural networks—the primary class of models used on image data—were initially developed in the late 1980's (LeCun et al., 1989). Long short-term memory networks, which are well suited to modelling sequence data such as audio and text, were first introduced by Hochreiter and Schmidhuber (1997). The reason these techniques did not see immediate popularity is because, in many cases, there was insufficient data to train them. Even when data is available, the computing power required to train these types of models is considerable. It is only in more recent years that these two factors have caught up: increase in uptake of computers has led to more data being collected, and improvements in computing technology have resulted in more computing power being available to process the data.

In parallel with the increase in the volume of data and computing power available, more research has been undertaken into the methods used to construct neural network models. Stochastic gradient-based optimisation algorithms, the main techniques used to fit the parameters of large networks, have received a lot of attention in recent years, yielding so-called adaptive learning rate methods such as Adam (Kingma and Ba, 2014), AdaGrad (Duchi et al., 2011), and RMSProp (Tieleman and Hinton, 2012). The low overhead of these methods, in terms of the memory and computation requirements, have allowed deep learning to scale to very large datasets. As a consequence, any novel methods for training deep networks must also be able to scale to large

datasets in order to become generally applicable.

Despite the large volumes of data being used to train deep networks, overfitting the training data is still a problem: predictions on the training set are still much more accurate than predictions on instances not seen during training. This is because deep neural networks are high capacity universal function approximators with very general inductive biases (Hornik et al., 1989). More recent work has shown that commonly used neural networks are cabaple of almost perfectly fitting training datasets that have randomised labels (Zhang et al., 2016). These results indicate that the inductive biases of these models are too general and that restricting the bias appropriately may result in even better performance. To see evidence of this, one need look no further than the existence of convolutional networks, which can be viewed as multi-layer perceptrons with sparse connections between layers and extreme weight sharing between units. This is a more restrictive inductive bias than that of fully connected networks, but it results in better performance on image data and other domains where convolution makes sense.

The price of restricting inductive biases in this domain-specific way is that the resulting algorithm is less generally applicable. Convolutional networks may work well on image data, but they should not be applied to tabular datasets that have no spatial or temporal dimension. Fully connected networks can be applied to tabular data, but their performance tends to lag behind that of decision tree induction for common tasks such as classification and regression—particularly those that train ensembles of decision trees (Breiman, 2001; Chen and Guestrin, 2016; Ke et al., 2017). However, tree-based methods lack the flexibility of deep learning in the sense that they cannot be easily retargeted towards other tasks such as dimensionality reduction.

This thesis investigates ways in which models with more specific inductive biases than conventional multi-layer perceptrons can be used for representation learning. Two avenues of research are followed:

- Constraining the set of functions that a neural network can represent in

order to enforce that its output changes slowly, motivated by the hypothesis that a function that changes slowly will have better generalisation performance;

- Adapting decision tree methods to fit into the gradient-based optimisation framework that has enabled deep learning to become so widespread, thus improving the inductive bias of representation learning models in domains where decision trees perform better.

Both of these research directions are motivated by the basic concept of designing learning algorithms that explicitly force instances with similar features to be assigned similar predictions. The first point is based around the idea of using distance metrics on real-valued vector spaces to determine whether feature vectors are similar. The second line of enquiry takes the alternative approach of quantising the feature space into a finite number of cells and associating all instances that lie in a cell with the same prediction.

The rest of this chapter discusses in more detail concepts that have already been introduced, and also summarises the contributions and structure of the remainder of the thesis.

1.1 Representation Learning

Representation learning is the process of constructing a mapping that translates data from one vector space into another space that is more useful for some downstream task. For the majority of use cases, one of these spaces is a set of observable data, such as images. The other set is usually a vector space spanned by latent variables that encode properties of the observed data. Some tasks involve mapping from the latent variable space into the observed data space: the so-called generative model that carries out this mapping is said to be performing synthesis. The other common problem is learning a function that can embed observed data items into the latent vector space. Methods

that aim to solve this second problem are known as embedding techniques, and are the main focus of this thesis.

Embeddings are useful for a large number of tasks. Facial recognition systems employ them to embed images of faces into low dimensional vector spaces that allow for efficient similarity search (Parkhi et al., 2015). Content-based recommender systems often use them to learn compact representations of audio and video that encode information useful for making predictions about user preferences (van den Oord et al., 2013). There are also embedding techniques designed to enable visualisation of high dimensional datasets by embedding them into two dimensional vector spaces (Min, 2010) that can be shown in a scatter plot. Embedding is also used for conventional machine learning tasks such as classification, where it replaces the feature engineering processes usually involved with applying machine learning to complex domains (Krizhevsky et al., 2012). As a consequence, the phrase feature extraction is sometimes used in place of embedding. Whatever the task, the role of embedding is to make some subsequent data-driven problem easier to solve.

A unifying trait in many of the best performing systems for each of these applications is the use of end-to-end training via gradient-based optimisation (LeCun et al., 2015). This is possible when both the function performing the feature extraction and the function that implements the downstream task are differentiable. By taking the composition of these two functions, and further composing them with a differentiable loss function, the entire pipeline can be treated as one function and optimised using stochastic gradient methods. For example, consider the problem of determining which abnormalities from some predefined set of K possible abnormal traits are present in an image of a retina. Convolutional neural networks have been explicitly designed to operate on image data, and they are differentiable, so they are a sensible choice for embedding the images into a space that is more manageable. The machine learning problem of assigning a set of labels to an instance is known as multi-label classification. One simple approach for solving it is to train a

binary classifier associated with each label, and to then use this classifier to predict whether the corresponding label should be in the set assigned to the instance. If logistic regression is selected as the differentiable binary classifier, the resulting model could make predictions using the following rule:

$$\hat{Y} = \left\{ c \,\middle|\, f_c(g(\vec{x})) > \frac{1}{2} \right\}, \tag{1.1}$$

where $\hat{Y}$ is the set of predictions, f_c is the logistic regression model for abnormality c, g is the convolutional network, and $\vec{x}$ is the image. The logistic regression models and convolutional feature extractor can all be trained simultaneously by solving an optimisation problem such as

$$\min_{f,g} - \sum_{\vec{x}_i, Y_i} \sum_{c=1}^{K} \mathbb{I}(c \in Y_i) \log(f_c(g(\vec{x}_i))) + \mathbb{I}(c \notin Y_i) \log(1 - f_c(g(\vec{x}_i))), \tag{1.2}$$

where $\mathbb{I}$ is the indicator function, and $\vec{x}_i$ and Y_i are the training images and corresponding ground truth labels, respectively. Equation 1.2 can be interpreted as maximising the likelihood of the predictions made by the model. End-to-end optimisation is seen as a requirement for representation learning systems, because the embedding component of the system should be optimised to produce features that are specifically designed to improve the performance of the subsequent component—in this case a multi-label classifier.

Representation learning methods provide a degree of flexibility not available to traditional machine learning approaches because they provide the ability to take a model initially designed for one task and repurpose it for another task with very little manual effort. This is very attractive from the point of view of the machine learning practitioner. For example, consider the previously proposed solution for determining the presence of abnormalities in retinal images. Suppose one decided that, rather than trying to detect abnormalities, it would be better to categorise the retinal image into one of five possible levels of severity, ranging being in perfect health to severe retinopathy—a condition

commonly associated with diabetes. Beckham and Pal (2017) show that this can be done by making a small modification to f and the loss function used for training. Performing these changes in practice is trivial, due to the existence of powerful tools that can automatically differentiate a supplied loss function and solve the optimisation problem using stochastic gradient descent on graphics processing units (Bergstra et al., 2010; Abadi et al., 2016; Paszke et al., 2017).

1.2 Inductive Bias

To talk about inductive bias of a learning algorithm in any technical sense, we require a more precise characterisation of what it is. Suppose we have a learning algorithm, A, that takes some training data, X, as input and produces a model, $h \in H$, as output. It is also common to refer to a model as a hypothesis, and the set of possible hypotheses, H, as the hypothesis space. We refer to a combination of learning algorithm and hypothesis space, (A, H), as a learning system. Mitchell (1980) provides one of the earliest examinations of inductive bias in machine learning. The first common source of inductive bias he identifies is the case where H does not contain a hypothesis capable of accurately modelling the relationship of interest. The other bias that is discussed is how the learning algorithm selects a hypothesis from the hypothesis space. Another interesting point made by Mitchell (1980) is that a learning algorithm must have some sort of inductive bias in order to be effective. In particular, he makes the observation that a learning system with no biases cannot generalise, because it can make no assumptions about how to interpolate between points seen in the training set. This was discussed without much rigor, and with several assumptions about the types of data used during training. Following on from the observation that learners must have a bias, it has also been demonstrated that there is no universally good inductive bias (Schaffer, 1994; Wolpert, 1996). This supports the claim that good generalisation is achieved by selecting an inductive bias well suited to the problem.

It should be noted that Mitchell (1980) considered only binary classification problems, and assumes that the learning algorithm will only select from those hypotheses that achieve perfect accuracy on the training data. In modern machine learning, we are interested in a much broader range of problems than just binary classification, and the noise present in most interesting datasets precludes a model from achieving perfect training set accuracy in many cases. To overcome these limitations, the inductive bias of a learning learning system will attributed to two aspects: the types of hypotheses in the subset, $G_X \subset H$, of the hypothesis space that will be considered for a given set of training instances, and the way in which the learning algorithm quantifies or ranks the quality of each of these hypotheses. In some cases G_X will be equal to H. For example, consider fitting a linear least squares regression model. In this case, the hypothesis class is the set of all affine transformations that map from the input vector space to the output vector space, and the quality of each hypothesis is measured with the squared error loss function. Any convex optimisation algorithm would implicitly consider all affine transformations, and in most cases would quite quickly find the best parameter settings, as measured by the squared error loss function.

In contrast, consider what would happen if the linear model were switched out for a multi-layer perceptron. This would transform what was a convex optimisation problem into a nonconvex problem. Gradient based algorithms that find provably optimal solutions to convex optimisation problems can still be applied, but all bets about optimality are off. The optimisation algorithm will no longer implicitly consider all possible weights for the neural network, and will now consider only a subset of the weight space near the initial guess for the weights. This is the difference between global and local optimisation techniques. Due to the complex hypothesis classes required for many representation learning tasks, it is assumed—but not required—throughout this thesis that G_X is a strict subset of H. That is, representation learning algorithms will usually explore only a subset of the hypothesis space while building a

model.

1.3 Discrete and Lipschitz Hypothesis Classes

Wolpert (1996) rigorously showed that there is no universal inductive bias that will be optimal for all problems. Hence, a large volume of machine learning research has been dedicated to finding those methods that are most useful for the types of problems most commonly encountered. A good example of this is the development of decision tree induction methods, which have been shown to be well suited to modelling tabular data (Chen and Guestrin, 2016; Ke et al., 2017). Most decision tree methods partition the feature space into a set of axis-aligned hyperrectangles, each of which is associated with a constant value used to make predictions. Popular ensembling techniques, such as random forests (Breiman, 2001) and gradient boosted decision trees (Friedman, 2001), improve upon this by combining multiple trees together to better model more complex relationships. Conventional decision trees are said to be discrete functions, due to their ability to take categorical features (i.e., discrete values) as input and because of the piecewise constant nature of their predictions.

Lipschitz functions are continuous functions that have a constant associated with them, the so-called Lipschitz constant. This value provides a measurement for how quickly the output of the function changes with respect to its inputs. Many classes of continous functions used in machine learning, such as deep neural networks, fall into this category of Lipschitz functions. Until recently, the Lipschitz property of neural networks had received little attention, but it has become apparent that it has a multitude of uses in generative neural network models and developing robustness to adversarial inputs (Arjovsky et al., 2017; Cisse et al., 2017). One of the questions investigated by this thesis is whether constraining the Lipschitz constant of a neural network will result in better predictive performance in the supervised prediction setting.

1.4 Contributions and Thesis Organisation

The main contribution of this thesis is a set of techniques for training nonlinear embedding models that have more restrictive inductive biases than the multi-layer perceptrons currently used for many representation learning tasks. The hypothesis is that the inductive biases proposed in this work improve the generalisation performance of systems that use representation learning techniques. The two primary lines of investigation are (a) Lipschitz-based regularisation approaches for neural networks, and (b) adapting decision tree methods to work in the gradient-based representation learning framework.

Chapter 2 reviews the representation learning literature related to learning Lipschitz neural networks and performing representation learning using decision tree methods.

Chapter 3 provides a derivation of the Lipschitz constant of feedforward neural networks with respect to several different p-norms. All of the standard layers appearing in fully connected and convolutional networks are considered, including recent developments such as residual connections.

Chapter 4 presents an algorithm for constraining the Lipschitz constant of a network. The algorithm is evaluated empirically using a variety of datasets and network architectures.

Chapter 5 shows how the Lipschitz constant with respect to any vector norm can be approximated empirically, and goes on to use this approximation to develop an empirical analog to the regularisation scheme given in Chapter 4.

Chapter 6 introduces a problem transformation method for learning tree-based embedding models on tabular data. Improving multi-label classification techniques based on nearest neighbour search is used as the primary motivation.

Chapter 7 adapts an online decision tree induction algorithm to learn from gradient information and shows how ensembles of these trees can be used to learn distributed representations from data.

Chapter 8 provides a summary of the contributions in this thesis, reflects on their implications, and speculates about future research directions that could result from the work that has been undertaken.

The content of Chapter 5 is based on an article that was presented at the 2018 edition of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases, in Dublin, Ireland. The problem transformation method described in Chapter 6 was first presented at the 2016 Asian Conference on Machine Learning in Hamilton, New Zealand. Chapter 7 contains significant overlap with a paper presented at the 2019 Asian Conference in Machine Learning, in Nagoya, Japan.

Chapter 2

Inductive Bias in Representation Learning

Representation learning involves mapping between two sets: one containing observable data points, and one consisting of vectors of latent variables. The task of mapping from latent variables to a data point in the observable domain is known as synthesis, or generation. The complementary task of mapping from an observed data point to a vector of latent variables is known as embedding. It is the task of learning embeddings using deep neural networks that this thesis is concerned with.

To provide background for the techniques introduced in this thesis, this chapter begins by introducing neural network building blocks, and then reviewing common strategies for assembling these blocks into functioning neural networks. This is followed by a description of the online optimisation techniques used to train networks, and the influence these optimisation algorithms and related regularisation approaches have on the inductive biases of the resulting learning systems. The chapter concludes with a high level overview of decision tree induction, and a discussion about previous attempts at learning representations with decision tree models.

2.1 Neural Network Building Blocks

The feed forward neural networks we use can be expressed as a series of function compositions,

$$f(\vec{x}) = (\phi_l \circ \phi_{l-1} \circ ... \circ \phi_1)(\vec{x}), \tag{2.1}$$

where each ϕ_i is typically an activation function, affine transformation, or pooling operation. When working at a slightly higher level of abstraction, the term layer is used to conceptually group related functions. For example, an affine transformation followed by an activation function is often referred to as a layer. The final layer in the network is often referred to as the output layer, and the data being fed into the network is often said to be coming from the input layer. If a layer is not at the beginning or end of the network it is known as a hidden layer. Some feedforward architectures contain residual connections, which cannot be explicitly represented using a linear chain of function compositions. This can be resolved by considering the composition to be a series of residual blocks, rather than invidual functions or layers.

The activation function that has been used most frequently in recent literature is the rectified linear unit (ReLU),

$$\phi_i^{relu}(\vec{x}) = \max(\vec{0}, \vec{x}), \tag{2.2}$$

where the max function is applied elementwise. This is the most prevalent activation function used in hidden layers, but the activation function used for the output layer of a network is determined by the task for which the network is being trained. For example, a network trained to perform classification will typically use a softmax activation function in the output layer, where the kth output is given by

$$\phi_{i,k}^{softmax}(\vec{x}) = \frac{e^{\vec{x}_k}}{\sum_{j=1}^{C} e^{\vec{x}_j}}, \tag{2.3}$$

where C is the total number of classes. One can interpret the output of this function as distribution indicating the confidence of the network for each of

the possible class assignments. It should be noted that neural networks are generally very poorly calibrated (Guo et al., 2017), so using these estimates as probabilities may not be sensible.

The simplest type of affine transformation is the one that is used in a so-called fully connected layer. It can be expressed as

$$\phi_i^{fc}(\vec{x}) = W^{(i)}\vec{x} + \vec{b}^{(i)}, \tag{2.4}$$

where $W^{(i)}$ is known as the weight matrix and $\vec{b}^{(i)}$ as the bias vector. The size of $W^{(i)}$ is determined by the number of components in the input vector, $\vec{x}$, and a value selected by the user that determines the number of output components of the layer. The elements of $W^{(i)}$ can be initialised using a number of different schemes, but the method of Glorot and Bengio (2010) is one of the more commonly used approaches. Suppose $W^{(i)}$ is an $N \times M$ matrix; Glorot and Bengio (2010) suggest that each element should be initialised by sampling a value from a normal distribution, $\mathcal{N}(0, \sqrt{\frac{2}{N+M}})$. They show that, under some mild assumptions related to the activation functions used in each layer, initialising weight matrices in this way greatly improves the rate at which the network will train. The elements of the bias vector are typically initialised to zero.

Convolutional layers make use of another type of affine transformation that is particularly useful for models that operate on image data. An intuitive way to derive this affine transform is to take the expression for a fully connected layer, but change the underlying ring that the matrix is defined over. In fully connected layers, each element of $W^{(i)}$, $\vec{x}$, and $\vec{b}^{(i)}$ is a scalar real number, but one can generalise this to allow the elements to come from an arbitrary ring, rather than the ring constructed by equiping real numbers with addition and multiplication operations. The set of discrete two dimensional signals can be used to construct a ring where the addition operation is elementwise addition of the discrete signals and the multiplication operation is convolution between

signals. Using this ring, each element in $W^{(i)}$, $\vec{x}$, and $\vec{b}^{(i)}$ can be chosen to be a single channel image or an image filter, as both are discrete two dimensional signals. These single channel images are often referred to as feature maps when they correspond to the outputs of a collection of hidden units. It is common for each of the signals in $\vec{b}$ to consist of a single real number repeated at every location in the signal. At a high level of abstraction, the definition of matrix multiplication over this ring of signals remains the same, but each of the scalar multiplications is replaced with a convolution operation, and the scalar additions are replaced with elementwise additions. This derivation of convolutional layers is convenient, as it allows efficient matrix multiplication algorithms, such as the method of Strassen (1969), to be directly applied to the acceleration of convolutional network implementations (Cong and Xiao, 2014).

Lastly, pooling layers can be used to reduce the size of the feature maps produced by convolutional layers. These pooling operations can be thought of as overlaying a grid on the feature map and applying an operation to all feature map elements that lie inside a grid cell. For example, two widely used operations are to (a) compute the mean element inside a grid cell, and (b) find the maximum value inside each cell. The value computed for each grid cell is then used to construct a smaller resolution feature map to be produced as part of the output of the pooling layer. This process is repeated for each of the input feature maps. The number of feature maps produced by a pooling layer is therefore equal to the number of feature maps provided as input to the layer. The size of each grid cell is left as a hyperparameter for the user to select. A typical setting is 2×2 pixels. Some architectures allow the grid cells to overlap with one another, resulting in strided pooling.

2.2 Network Architectures

The catalyst for modern neural network research is backpropagation, originally developed by Werbos (1974) and then rediscovered by Rumelhart et al. (1986). This method for training a neural network proceeds by defining a task-specific loss function, computing derivatives of this loss with respect to the network parameters, and then using a gradient-based optimisation algorithm to find a set of weights that yield a good predictive model. The large number of parameters found in neural networks has led to the use of stochastic first order optimisation methods, as they scale to larger problems much better than Newton-based methods or first order batch methods. A consequence of using first order methods is that, in networks with many layers, the magnitudes of the gradients used for training the earlier layers in the network tend to gravitate towards extreme values—both large and small—which results in unreliable convergence behaviour. Each layer a gradient vector is backpropagated through has the opportunity to increase or decrease the magnitude of the gradient vector. In practice, it has been observed that most layers in the network will behave in the same way, often resulting in the gradients shrinking or exploding as they are propagated back through the network (Hochreiter, 1991). This phenomenon—known as the vanishing (or exploding) gradient problem—has been one of the main influences for how neural network archiectures have evolved over time. In Newton-based optimisation approaches, this is compensated for through the use of second order curvature information.

Early backpropagation neural networks were based soley on a series of fully connected layers with sigmoid-shaped activation functions—usually the logistic function, although sometimes the hyperbolic tangent was used. LeCun et al. (1989) introduced the idea of using affine transformations based on convolution operations to learn a feature extractor for image data, using handwritten digit recognition as the motivating application. The neocognitron (Fukushima, 1980) is another notable attempt at integrating convolution operations into artificial neural networks, but this method makes use of a

biologically inspired unsupervised learning rule rather than backpropagation, and has not been shown to scale to complex image domains. Futher work from LeCun et al. (1998) resulted in the LeNet family of neural network models. Network architectures of this variety employ a series of blocks consisting of a convolutional layer with a sigmoid activation function, and a pooling layer. Multiple instances of these blocks are composed until the resolution of the feature maps in the final block of the composition is very small, often 1×1 pixels, due to the downsampling caused by the pooling layers. This convolutional feature extractor is then followed by several fully connected hidden layers, each with a sigmoid activation function. The success of models that use these convolutional and downsampling operations is typically attributed to their reduced parameter count and improved inductive bias, due to the translation invariance property of convolutions (LeCun et al., 2015).

The LeNet family of models set the standard for designing supervised backpropagation neural networks for image data until the work of Krizhevsky et al. (2012), which demonstrated the first deep neural network to be successfully applied to a highly complex image classification task, the ImageNet Large Scale Visual Recognition Challenge. This network, now commonly referred to as AlexNet, made several key changes compared to LeNet-based networks: using rectified linear units instead of sigmoid activation functions, using the max operation in the pooling layers, using a deeper network topology with more layers, and using significantly larger layers—most containing over a hundred feature maps. The justification for using the ReLU activation function is that it does not saturate as the weights in the network become larger. Sigmoid activation functions are said to saturate because, as the magnitudes of the values produced by the affine transformations grow larger, the output of the sigmoid activation function converges towards constant values. As the activations approach these limits, the corresponding derivative shrinks towards zero, playing havok with the stochastic first order optimisation methods.

Subsequent ImageNet challenges have acted as a catalyst for futher work in

neural network architecture research. Simonyan and Zisserman (2014) demonstrated with their VGG family of networks that using smaller filters, with a resolution of only 3×3 pixels, in convolutional layers results in better generalisation. They also show that using several convolutional layers between each pooling operation allows one to construct deeper networks, which in their case also resulted in better predictive performance. The residual networks proposed by He et al. (2016) demonstrate a further improvement in performance, while also greatly reducing the parameter count compared to comparable VGG networks. This is accomplished by having many fewer feature maps in each layer and increasing the depth of the network by several times. The main innovation is the introduction of residual connections, which enable the construction of residual blocks,

$$\phi^{res}(\vec{x}) = \vec{x} + (\phi_{j+n} \circ \ldots \circ \phi_{j+1})(\vec{x}), \tag{2.5}$$

that make use of so-called skip connections, where the input vector for a linear chain of function compositions is also added to the output of the composition. Constructing very deep networks with these residual blocks results in an architecture that is much more resilient to the vanishing gradient problem than those produced by previously described network design approaches. By adding the input of the block to the output, a series of shortcuts are formed that allow the gradient information propagated during training to pass through a much shorter chain of operations, thus reducing the chance of the gradient gradually shrinking to zero.

However, it is unclear whether resistance to the vanishing gradient problem is the real reason that residual networks perform so well. He et al. (2016) claim that the primary benefit of residual networks is that they do not suffer from the vanishing gradient problem, and therefore much deeper networks can be trained, which they claim will result in superior performance. Zagoruyko and Komodakis (2016) have shown that wide residual networks with depths similar to VGG networks perform as well as the much deeper residual networks trained by He et al. (2016), with the added bonus of being more computationally

efficient. Given that VGG networks can perfectly fit their training data, we know that they do not suffer from the type of optimisation deficiencies caused by the vanishing gradient problem. If they are not impacted by the vanishing gradient problem, then why do residual networks of similar depths outperform them? This is a question for which there is currently no answer.

2.3 Optimisation Methods

The backpropagation approach is used to evaluate derivatives of some loss function with respect to the network parameters, but it does not stipulate how the parameters should be changed in order to optimise the objective. The role of determining how the weights should be modified is delegated to an optimisation procedure. The vast majority of methods make use of only first order gradient information because computing the full inverse Hessian matrix required for Newton-type methods is intractable for almost all architectures.

2.3.1 Empirical Risk Minimisation

Before explaining how a loss function can be minimised using gradient-based optimisation methods, it is useful to first formally describe the types of optimisation problems usually encountered when training supervised learning models. Ideally, we would like to minimise the expected loss on all possible data points,

$$\min_{\vec{\theta}} \mathbb{E}[l(f_{\vec{\theta}}(\vec{x}), \vec{y})], \tag{2.6}$$

where l is a task specific loss function that measures the quality of the predictions, $f_{\vec{\theta}}$ is an embedding model parameterised by $\vec{\theta}$, $\vec{x}$ is a random variable representing an input feature vector, and $\vec{y}$ is a random variable representing the ground truth output. The objective function in Equation 2.6 is known as the expected risk. It is not possible to optimise this objective directly, because the underlying data distribution is not known. However, if each instance in the training dataset is independently drawn from the same distribution, we

can approximate the objective using

$$\mathcal{L}(f_{\vec{\theta}}) = \frac{1}{N}\sum_{i=1}^{N} l(f_{\vec{\theta}}(\vec{x}_i), \vec{y}_i) \approx \mathbb{E}[l(f_{\vec{\theta}}(\vec{x}), \vec{y})], \tag{2.7}$$

where N is the number of instances in the training set.

This idea of using independent and identically distributed training examples to approximate the expected loss, with the goal of formulating a tractable optimisation problem, is known as empirical risk minimisation. Throughout the remainder of this thesis $\mathcal{L}$ will sometimes be written as a function of the model parameters, rather than the function implementing the model itself, in order to make the exposition of some ideas more clear.

2.3.2 Batch Optimisation

The task of training backpropagation neural networks can be formalised as an empirical risk minimisation problem,

$$\vec{\theta}^* = \arg\min_{\vec{\theta}} \mathcal{L}(\vec{\theta}), \tag{2.8}$$

where $\mathcal{L}$ is the loss function composed with a neural network, as defined in Equation 2.7, and $\vec{\theta}$ represents all of the network parameters serialised into a single vector. Assuming $\mathcal{L}$ is differentiable, we wish to iteratively apply an update rule that will refine some initial estimate of $\vec{\theta}$ to a locally optimal setting. Consider the first order Taylor series approximation to the loss function,

$$\mathcal{L}(\vec{\theta} + \alpha\vec{p}) \approx \mathcal{L}(\vec{\theta}) + \alpha\vec{p} \cdot \nabla_{\vec{\theta}}\mathcal{L}(\vec{\theta}), \tag{2.9}$$

where $\vec{p}$ is a unit vector and α is a positive real number. At each iteration of the update procedure, we must find the direction, $\vec{p}$, that provides the best

decrease in loss,

$$\min_{\vec{p}} \vec{p} \cdot \nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}). \tag{2.10}$$

We know that $\|\vec{p}\| = 1$, so the minimiser of this dot product is

$$\vec{p}^* = -\frac{\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta})}{\|\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta})\|}, \tag{2.11}$$

i.e., the unit vector that points in the opposite direction of the gradient. One can then use a line search method to determine the magnitude, α, of the update. This can be used to generate a sequence of $\vec{\theta}$ vectors,

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \alpha_t \frac{\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_{t-1})}{\|\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_{t-1})\|}, \tag{2.12}$$

where α_t is the result of performing a line search to find the best step size at iteration t. The process is iterated until the magnitude of the gradient vector is sufficiently close to zero.

The backpropagation neural network proposed by Rumelhart et al. (1986) uses an update rule similar to the expression given in Equation 2.12, but the gradient magnitude is absorbed into α and a so-called momentum term is also included, yielding

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \alpha_t \nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_{t-1}) + \eta(\vec{\theta}_{t-1} - \vec{\theta}_{t-2}), \tag{2.13}$$

where η is the momentum rate, which is typically set to 0.9. Rather than using a line search method to find the best setting for α_t at each time step, Rumelhart et al. (1986) instead employ a user-specified learning rate that remains constant for all iterations.

Using a constant learning rate removes the need to perform a line search at each iteration, and thus makes each step more efficient. However, it can—and often does—lead to significantly worse performance due to poorly optimised model parameters. Much of the research into neural network optimisation has investigated how one can find a good step size without requiring multiple

evaluations of the objective function.

Another way to remove the reliance on a line search method is to incorporate second order derivative information. Consider the second order Taylor series approximation of the objective,

$$\mathcal{L}(\vec{\theta} + \vec{p}) \approx \mathcal{L}(\vec{\theta}) + \vec{p} \cdot \nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}) + \frac{1}{2} \vec{p}^{\top} \nabla^2_{\vec{\theta}} \mathcal{L}(\vec{\theta}) \vec{p}, \tag{2.14}$$

which provides a quadratic model of the loss. When the Hessian of the loss is positive definite, there exists a closed form solution for finding the $\vec{p}$ that provides the best parameter update, resulting in the following update rule:

$$\vec{\theta}_t = \vec{\theta}_{t-1} - \left(\nabla^2_{\vec{\theta}} \mathcal{L}(\vec{\theta}_{t-1}) \right)^{-1} \nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_{t-1}), \tag{2.15}$$

which is known as Newton's method. Computing the entire inverse Hessian matrix at every time step can become very computationally intensive, but several so-called Quasi-Newton methods, which instead approximate this matrix, have been developed to alleviate this issue. These more efficient methods are often still not suitable for training neural networks due to the enormous size of the approximate Hessian, which will generally not exhibit some sort of structured sparsity. Nevertheless, there are still several cases where second order information has been useful for training networks. For example, Le et al. (2011) demonstrate that the conjugant gradient method and a low memory variant of the Broyden–Fletcher–Goldfarb–Shanno algorithm are well suited to the layer-wise pretraining used in some semi-supervised learning scenarios.

2.3.3 Stochastic Optimisation

If the training set size is large, $\mathcal{L}$ is likely to contain a large number of terms. The derivative of the loss will, in turn, also contain a large number of terms, resulting in a computationally expensive update rule. In practice, stochastic optimisation methods that approximate the gradients are used to optimise the model parameters. Specifically, it is common practice to consider only a

small number of the terms in the empirical risk expression in Equation 2.7. This works because the empirical risk, and any random subset of terms in the empirical risk, are both unbiased estimates of the expected risk.

When training a neural network with a stochastic gradient-based optimisation method, one typically makes multiple passes over the training dataset. Each one of these passes is known as an epoch. The training dataset is randomly shuffled at the beginning of each epoch, and training proceeds by taking the first n instances in the shuffled collection, where n is usually referred to as the batch size. After this batch of n instances is used to estimate the gradient of the objective and perform a weight update, the process is repeated with the next n instances in the shuffled training dataset. This process continues until a stopping criterion determined by the user is met. This usually means iterating for a fixed number of epochs, or until the predictive performance measured on a validation begins to plateau.

Determining the step size at each iteration in the stochastic setting is not as straightforward as computing the step direction. A line search is generally not a feasible solution, because the gradients computed at each time step exhibit too much variance, and one is therefore forced to be conservative with how much parameters are changed at each iteration. Simple approaches, like picking a fixed learning rate, exhibit quite poor accuracy compared to methods that dynamically select the step size at each iteration. Techniques for selecting the learning rate at each time step can be roughly divided into two categories: those that are based on a hand-engineered schedule, and those that use gradient statistics to adapt the learning rate at a more fine grained level. It is often possible to combine both families of techniques to achieve particularly good performance.

Learning rate schedules determine the step size at each iteration as a function of the iteration count. Common techniques for designing learning rate schedules make use of exponential decay or step functions. Learning rate decay is implemented by multiplying the learning rate by a number slightly

less than one after each epoch. The learning rate stepping strategy involves decreasing the learning rate by an order of magnitude at a predetermined iteration count. Often there will be several learning rate steps throughout the entire training process. While learning rate decay and stepping are the two most popular learning rate scheduling techniques, other approaches also exist. The work of Smith (2017) demonstrated that a cyclical learning rate schedule that follows a sawtooth trend can often exceed the performance of a constant or exponentially decayed learning rate.

Dynamic learning rate techniques that use statistics based on historic gradient values usually attempt to maintain a per-parameter adaptation of a global learning rate. This global learning rate—which is shared accross all parameters—can be selected using one of the learning schedules described previously, or it could simply be a constant specified by the user. The common ancestor of adaptive stochastic gradient methods that are used in modern deep learning is the AdaGrad algorithm of Duchi et al. (2011). AdaGrad maintains a sum of the outer product of all observed gradient vectors,

$$G_t = \sum_{i=1}^{t} \left(\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_i)\right) \cdot \left(\nabla_{\vec{\theta}} \mathcal{L}(\vec{\theta}_i)\right)^{\top}. \tag{2.16}$$

The inverse of the square root of G_t can then be used to transform the gradient vector, similar to how the Hessian is used in Newton-type methods. In practice, it is sufficient to make use of only the elements that lie on the diagonal of G_t, which corresponds to the second moments of the gradients with respect to each parameter. The resulting update rule is

$$\vec{\theta}_t^{(j)} = \vec{\theta}_{t-1}^{(j)} - \frac{\alpha}{\sqrt{G_t^{(j,j)}}} \nabla_{\vec{\theta}} \mathcal{L}^{(j)}(\vec{\theta}_{t-1}), \tag{2.17}$$

where j is the index of a component in the parameter vector.

Several other methods can be seen as extensions of AdaGrad. The RMSProp approach, first presented in an online lecture series by Tieleman and

Hinton (2012), can be seen as a modification that uses an exponential moving average, rather than a summation, to compute the diagonal of G_t. The Adam optimiser of Kingma and Ba (2014) is another variant of AdaGrad, and is used extensively throughout this thesis. Like RMSProp, Adam uses an exponential moving average to compute the second moment of the gradient, but it also maintains an exponential moving average of the first moment. This estimate of the first moment is used similarly to the momentum term in Equation 2.13. The Adam update rule is given by

$$\begin{aligned}
\vec{m}_t^{(j)} &= \beta_1 \vec{m}_{t-1}^{(j)} + (1-\beta_1)\nabla_{\vec{\theta}}\mathcal{L}^{(j)}(\vec{\theta}_{t-1}) \\
\vec{v}_t^{(j)} &= \beta_2 \vec{v}_{t-1}^{(j)} + (1-\beta_2)(\nabla_{\vec{\theta}}\mathcal{L}^{(j)}(\vec{\theta}_{t-1}))^2 \\
\hat{\vec{m}}_t^{(j)} &= \frac{\vec{m}_t^{(j)}}{1-(\beta_1)^t} \\
\hat{\vec{v}}_t^{(j)} &= \frac{\vec{v}_t^{(j)}}{1-(\beta_2)^t} \\
\vec{\theta}_t^{(j)} &= \vec{\theta}_{t-1}^{(j)} - \alpha \frac{\hat{\vec{m}}_t^{(j)}}{\sqrt{\hat{\vec{v}}_t^{(j)}} + \epsilon}
\end{aligned} \tag{2.18}$$

This optimiser requires the user to set several hyperparameters: β_1 is the fading factor for the first moment average, β_2 is the fading factor for the second moment, α is the global learning rate that is to be adapted on a per-parameter basis, and ϵ is a small number to prevent division by zero. All applications of Adam in this thesis use values of 0.9, 0.999, and 10^{-8} for the β_1, β_2, and ϵ hyperparameters, respectively. These values are recommendations from Kingma and Ba (2014), who observe that these settings tend to transfer well across different tasks.

2.4 Regularising Neural Networks

The task of constraining the output of a network to change slowly with respect to its inputs—which is the focus of the first half of this thesis—is best viewed as a form of regularisation. A regulariser is some mechanism associ-

ated with a learning algorithm that can be used to control or influence the capacity of learned models. The capacity of a model is some measurement of the complexity of the relationships it can accurately model. An often used heuristic measure of model capacity is to count the number of free parameters that a learning algorithm can modify during training. In the case of neural networks, this would be the total number of weights and biases. Several theoretical notions of capacity have been introduced in the literature, such as the Vapnik–Chervonenkis (VC) dimension,[1] the fat shattering dimension (Kearns and Schapire, 1994), and the Rademacher complexity (Bartlett and Mendelson, 2001).

One of the oldest methods for regularising backpropagation neural networks is weight decay (Hinton, 1987; Krogh and Hertz, 1992). This method is formulated by adding a quadratic penalty term to the optimisation objective,

$$\mathcal{L}(\vec{\theta}) = \frac{1}{N} \sum_{i=1}^{N} l(f_{\vec{\theta})}(\vec{x}_i), \vec{y}_i) + \frac{\lambda}{2} \|\vec{\theta}\|_2^2, \tag{2.19}$$

where λ is a hyperparameter the user can set in order to influence the model capacity. The first term can be referred to as the data term, and the second term as the regularisation term. This method is known as weight decay because, when stochastic gradient descent is used, the resulting weight update formula contains a term that causes the weight to decay towards zero unless counteracted by the derivative of the data term.

Another common method for reducing overfitting when training neural networks is the practice of early stopping. This is done by holding out a subset of the training data as a validation set that can be used to monitor generalisation performance during training. When the performance of the network stops improving on the validation set, training can be halted. Recent work has shown that early stopping is theoretically well founded. Hardt et al. (2016) provide an analysis of how stochastic optimisation methods impact

[1] The article that first presented VC dimensions was originally printed in Russian in 1971, but Vapnik and Chervonenkis (2015) is a widely circulated English translation.

the generalisation performance of a model by characterising the relationship between the number of weight updates and the potential for overfitting.

One of the most widely applied regularisation techniques currently used for deep networks is dropout (Srivastava et al., 2014). By randomly setting the activations of each hidden unit to zero with some probability, p, during training, this method noticeably reduces overfitting for a wide variety of models. Various extensions have been proposed, such as randomly setting weights to zero instead of activations (Wan et al., 2013). Another modification, concrete dropout (Gal et al., 2017), allows one to directly learn the dropout rate, thus making the search for a good set of hyperparameters easier. Kingma et al. (2015) have also shown that the noise level in Gaussian dropout—where each activation is instead multiplied by some Gaussian random variable—can be learned during optimisation. Srivastava et al. (2014) found that constraining the ℓ_2 norm of the weight vector for each unit in isolation—a technique that they refer to as maxnorm—slightly improved the performance of networks trained with dropout, but it was mostly ineffective when used without dropout.

Batch normalisation (Ioffe and Szegedy, 2015), which was initially developed to accelerate the convergence of the training process, has also been shown to improve generalisation. It is efficient and simple to implement: it consists solely of standardising the activations of each layer by aggregating statistics over minibatches. The activations are then rescaled and translated by model parameters learned during training. A similar technique that is more effective in the small minibatch case, when the statistics required by batch normalisation cannot be computed reliably, is weight normalisation (Salimans and Kingma, 2016). Rather than computing the mean and standard deviation, the orientation and magnitude of each weight vector are decoupled and learned separately. Interestingly, it has been shown that weight decay provides no regularisation benefit when used in conjunction with these methods (van Laarhoven, 2017). It does change the effective learning rate of commonly used optimisation algorithms, but one could achieve the same effect by simply

changing the learning rate directly and thus have the advantage of optimising fewer hyperparameters.

Theoretical investigations into the performance of machine learning algorithms are concerned foremost with proving bounds on the generalisation gap of a learning system. The generalisation gap is the difference between the empirical risk measured on the training set, and the expected risk, which cannot be measured. Several papers have shown that the generalisation gap of a neural network is dependent on the magnitude of the weights (Bartlett et al., 2017; Neyshabur, 2017; Bartlett, 1998). Early results, such as Bartlett (1998), present bounds that assume sigmoidal activation functions, but nevertheless relate generalisation to the sum of the absolute values of the weights in the network. More recent work has shown that the product of spectral norms, scaled by various other weight matrix norms, can be used to construct bounds on the generalisation gap. Bartlett et al. (2017) scale the spectral norm product by a term related to the element-wise ℓ_1 norm, whereas Neyshabur et al. (2018) use the Frobenius norm. Neyshabur et al. (2018) speculate that Lipschitz continuity alone is insufficient to guarantee generalisation. However, it appears in multiple generalisation bounds (Neyshabur, 2017; Bartlett et al., 2017), and this thesis presents evidence that it is an effective means for controlling the generalisation performance of a deep network.

Yoshida and Miyato (2017) proposed a new regularisation scheme in the form of a term in the loss function that penalises the sum of spectral norms of the weight matrices. The work in this thesis differs from their method in several ways. Firstly, we investigate norms other than ℓ_2. Secondly, Yoshida and Miyato (2017) use a penalty term, whereas we employ a hard constraint on the induced weight matrix norm. Moreover, they penalise the sum of the norms, but the Lipschitz constant is determined by the product of operator norms. Finally, Yoshida and Miyato (2017) use a heuristic to regularise convolutional layers that does not correspond to penalising their spectral norm. Specifically, they compute the largest singular value of a flattened weight tensor, as opposed

to deriving the true matrix corresponding to the linear operation performed by convolutional layers. Explicitly constructing this matrix and computing its largest singular value—even approximately—would be prohibitively expensive. Chapter 3 provides efficient methods for computing the ℓ_1 and ℓ_∞ norms of convolutional layers exactly, and shows how one can approximate the spectral norm efficiently by avoiding the need to explicitly construct the matrix representing the linear operation performed by convolutional layers.

Most of the recent regularisation approaches are justified through empirical successes, rather than an explicit inductive bias that is hypothesised to improve the performance of neural networks when applied to some set of problems. However, methods such as ℓ_2 regularisation and spectral decay do have some theoretical basis related to what can be proved about the generalisation of certain hypothesis classes with bounded weight norms. This is a subtly different notion to inductive bias. For example, attempting to approximate a sine wave with a linear function will probably generalise well, because linear models with bounded weights have very strong generalisation guarantees, but the space of all linear models does not contain any function that resembles a sine wave.

2.5 Induction of Decision Trees

The second half of this thesis, which concerns the use of discrete functions to extract features from data, is based on training ensembles of decision trees to produce feature vectors. Standard decision trees hierarchically subdivide the feature space into a set of cells that each have an associated prediction value. Associated with each of the inner nodes of a decision tree is a decision function that inspects the features of an instance and determines which subtree the instance should be propagated to. Once the instance reaches a node with no children, a constant value associated with the leaf is used as a prediction of the label. Most decision trees consider two types of features: numeric and

nominal. Numeric features are elements of the real numbers, and nominal features are discrete values that may or may not have an associated ordering.

Growing decision trees is usually done in a greedy manner. The process begins with the root node of the tree, which in the case of classification will predict the most common class in the training dataset. One then searches for the split that can be applied to the root node that best optimises some splitting criterion. Common criteria include information gain and the Gini index, often with some additional constraints surrounding the minimum number of training instances that can be assigned to a single node, or the maximum depth of the tree. This process then repeats recursively until the decision tree fits the training perfectly or the constraints prevent any new nodes from being added to the tree. Some methods will then apply a pruning algorithm that removes nodes that cause the performance of the tree to deteriorate on data not seen during training.

A variety of decision functions can be utilised by the inner nodes in a decision tree. The most common type of function is a univariate split, which inspects a single component of the feature vector when determining which subtree the instance should be assigned to. If a split is acting on a nominal feature, then a common decision function is to simply have a subtree associated with each possible value of the feature and propagate each instance down the subtree associated with the value contained in the instance. For numeric attributes, there are several options. The most common approach is to construct binary splits and use a threshold to determine which branch an instance should be propagated down. Other possibilities also exist: the feature could be quantised and treated as a nominal quantity, resulting in a multiway split (Frank and Witten, 1996). Quantised numeric attributes can also be treated as ordinal quantities and a boundary between two quantiles can be selected to create a binary split (Frank and Witten, 1999).

It is also possible to construct trees that consider more than one attribute in the decision function, yielding multivariate splits (Brodley and Utgoff, 1995).

Methods that fall into this category most commonly make use of multivariate boolean predicates, or linear decision boundaries that imply a binary split. The latter of these two classes of multivariate splits—those that use thresholded linear functions—are often referred to as oblique splits. The difficulty in designing tree induction algorithms of this sort is specifying a tractable optimisation criterion for learning these multivariate decision functions.

It is common practice to construct an ensemble of trees to achieve greater predictive performance than a single decision tree. Bootstrap aggregating (Breiman, 1996), also known as bagging, and gradient boosting (Friedman, 2001) are two particularly popular approaches. Bagging trains each decision tree on a different random resampling of the training data—a so-called bootstrap. These bootstraps are constructed by sampling with replacement from the original training set to construct an artificial training set that usually contains the same number of instances as the original set. After each tree has been trained on a different bootstrap, their predictions can be aggregated by taking an average—i.e., the mode for classification, or the mean for regression. Gradient boosting ensembles are constructed sequentially; the addition of each new model attempts to fix mistakes that are made by the current ensemble.

2.6 Learning Representations with Trees

Several existing studies present methods that incorporate decision trees in the process of learning representations or training neural networks. Most of this prior work follows the example of Nguyen (2002), who proposed using "soft" splits in decision trees. The resulting trees are therefore differentiable and can be trained using gradient based optimisation methods. A multitude of works have incorporated this strategy; several particularly related papers are discussed in this section.

Kontschieder et al. (2015) replace the fully connected layers often found at the end of convolutional networks with a soft decision tree. This soft decision

tree has a predefined structure and each of the binary decision nodes performs an oblique split (Murthy et al., 1994) using a differentiable model that is not required to be linear. These models can be interpreted as producing a probability distribution over which subtree an instance should be propagated down. In contrast to conventional decision trees, which employ hard splits, the instance is actually propagated down all of the subtrees in the network, and the final prediction is a weighted sum of the predictions made by all the leaf nodes, where the weights are the estimated probabilities of the instance reaching each leaf. Because the splits are differentiable, they can be jointly optimised with the convolutional feature extractor using backpropagation, resulting in a model with comparable performance to a conventional convolutional network.

One of the primary motivations for using decision tree models for representation learning is to construct an interpretable feature extractor. Frosst and Hinton (2017) adapt the idea of neural network distillation (Hinton et al., 2015) to convert a standard fully connected neural network trained on tabular data into a soft decision tree. They show that training the soft decision tree in this way yields better performance that training it directly from the labels in the training data, and also results in a model that can be interpreted by a human. Yang et al. (2018) focus on designing a method that learns an interpretable decision tree. They present a method for training soft decision trees using backpropagation. The technique uses a soft binning function that enables the model to learn how features should be discretized. The depth of the tree is determined by the number of features in the dataset, which they acknowledge does not scale to datasets with large numbers of features. They mitigate this by training an ensemble where each tree is supplied with a subset of the attributes. They show that their method can learn interpretable models on small tabular datasets, but the accuracy is similar to conventional neural networks.

Zhou and Feng (2017) show that an ensemble of ensembles of conventional decision trees can be used to learn useful representations for classification,

but their method cannot be generalised to other tasks and requires a very large number of decision models to achieve similar accuracy to systems such as XGBoost—a library for training ensembles of trees using gradient boosting (Chen and Guestrin, 2016).

The batch decision tree learners most related to the work presented in the second part of this thesis are those based on the gradient boosting framework of Friedman (2001). Conventional gradient boosting methods expand an ensemble of weak learners using a sequential update rule that aims to find the weak learner that best corrects the mistakes currently made by the ensemble. The algorithm proposed by Friedman (2001) uses the derivative of a loss function, such as cross entropy or squared error, to determine how much the new weak learner should change the output of the ensemble. This is often thought of as gradient descent in function space. We take inspiration from XGBoost (Chen and Guestrin, 2016), which instead uses a second order approximation to the loss function. This has an analogous interpretation as performing Newton steps in function space. Chen and Guestrin (2016) also design a decision tree method that can learn directly from the gradient and Hessian information provided by the loss function. We reformulate this idea so that a single tree can be optimised using this gradient information, and also take ideas from the Hoeffding tree of Domingos and Hulten (2000) to enable incremental learning, thus enabling joint training with conventional neural network layers.

Chapter 3

The Lipschitz Constant of Neural Networks

Supervised learning is primarily concerned with the problem of approximating a function given examples of what output should be produced for a particular input. In order for the approximation to be of any practical use, it must generalise to unseen data points. Thus, an appropriate space of functions in which the machine should search for a good approximation must be selected, and an algorithm capable of searching through this space must be available. This is typically done by first picking a large family of models, such as support vector machines or decision trees, and applying a suitable search algorithm designed specifically for that model family. Crucially, when performing the search, regularisation techniques specific to the chosen model family must be employed to combat overfitting. For example, one could limit the depth of decision trees considered by a learning algorithm, or impose probabilistic priors on tunable model parameters.

One way to design an effective regularisation scheme is to develop a measure of model complexity that can be penalised during training. Chapter 1 mentions several approaches to quantifying model capacity, such as the VC dimension and Rademacher complexity, that have been developed in the literature on learning theory. However, the bounds derived using these measures do not

suggest novel regularisation approaches, but rather serve as a justification for the weight decay regulariser (Bartlett, 1998). This chapter proposes a measure of model capacity based on an upper bound of the Lipschitz constant of the network. Chapter 4 shows that this measure can be efficiently constrained during training, and that it has favourable regularisation properties.

One of the prevailing themes in the theoretical work surrounding neural networks is that the magnitude of the weights directly impacts the generalisation gap (Bartlett, 1998; Bartlett et al., 2017; Neyshabur, 2017), with larger weights being associated with poorer relative performance on new data. In several of the most recent works (Bartlett et al., 2017; Neyshabur, 2017), some of the dominant terms in these bounds are equal to the upper bound of the Lipschitz constant of neural networks derived in this chapter. While previous works have only considered the Lipschitz continuity of networks with respect to the ℓ_2 norm, we put a particular emphasis on working with ℓ_1 and ℓ_∞ norms. In Chapter 4, the derivations in this chapter are used to construct a practical algorithm for constraining the Lipschitz constant of a network during training.

3.1 Computing the Lipschitz Constant

A function, $f : X \to Y$, is said to be Lipschitz continuous if it satisfies

$$D_Y(f(\vec{x}_1), f(\vec{x}_2)) \leq k D_X(\vec{x}_1, \vec{x}_2) \quad \forall \vec{x}_1, \vec{x}_2 \in X, \tag{3.1}$$

for some real-valued $k \geq 0$, and metrics D_X and D_Y. The value of k is known as the Lipschitz constant, and the function can be referred to as being k-Lipschitz. Generally, we are interested in the smallest possible Lipschitz constant, but it is not always possible to find it. In this section, it is shown how one can compute an upper bound to the Lipschitz constant of a feed-forward neural network with respect to its input features.

Recall the formalisation of deep neural networks, given in Equation 2.1, as a series of function compositions. A particularly useful property of Lip-

schitz functions is how they behave when composed: the composition of a k_1-Lipschitz function, f_1, with a k_2-Lipschitz function, f_2, is a $k_1 k_2$-Lipschitz function. It is important to note that $k_1 k_2$ will not necessarily be the smallest Lipschitz constant of $(f_2 \circ f_1)$, even if k_1 and k_2 are individually the best Lipschitz constants of f_1 and f_2, respectively. Denoting the Lipschitz constant of some function, f, as $L(f)$, repeated application of this composition property yields the following upper bound on the Lipschitz constant for the entire feed-forward network:

$$L(f) \leq \prod_{i=1}^{l} L(\phi_i). \tag{3.2}$$

Using Inequality3.2, the Lipschitz constants of each layer can be computed in isolation and combined in a modular way to establish an upper bound on the constant of the entire network. In the remainder of this section, closed form expressions for the Lipschitz constants of common layer types are derived in the case where D_X and D_Y correspond to the ℓ_1, ℓ_2, or ℓ_∞ norms. As will be shown in Chapter 4, Lipschitz constants with respect to these norms can be constrained efficiently.

3.1.1 Fully Connected Layers

A fully connected layer, $\phi^{fc}(\vec{x})$, implements an affine transformation parameterised by a weight matrix, W, and a bias vector, $\vec{b}$:

$$\phi^{fc}(\vec{x}) = W\vec{x} + \vec{b}. \tag{3.3}$$

Previous work has already established that, under the ℓ_2 norm, the Lipschitz constant of a fully connected layer is given by the spectral norm of the weight matrix (Yoshida and Miyato, 2017; Neyshabur, 2017). This section provides a slightly more general formulation that will prove to be more useful when considering other p-norms. We begin by plugging the definition of a fully

connected layer into the definition of Lipschitz continuity:

$$\|(W\vec{x}_1 + \vec{b}) - (W\vec{x}_2 + \vec{b})\|_p \leq k\|\vec{x}_1 - \vec{x}_2\|_p. \tag{3.4}$$

By setting $\vec{a} = \vec{x}_1 - \vec{x}_2$ and simplifying the expression slightly, we arrive at

$$\|W\vec{a}\|_p \leq k\|\vec{a}\|_p, \tag{3.5}$$

which, assuming $\vec{x}_1 \neq \vec{x}_2$, can be rearranged to

$$\frac{\|W\vec{a}\|_p}{\|\vec{a}\|_p} \leq k, \quad \vec{a} \neq 0. \tag{3.6}$$

The smallest Lipschitz constant is therefore equal to the supremum of the left-hand side of the inequality,

$$L(\phi^{fc}) = \sup_{\vec{a} \neq 0} \frac{\|W\vec{a}\|_p}{\|\vec{a}\|_p}, \tag{3.7}$$

which is the definition of the operator norm.

For the p-norms considered in this chapter, there exist efficient algorithms for computing operator norms on relatively large matrices (Tropp, 2004). Specifically, for $p = 1$, the operator norm is the maximum absolute column sum norm; for $p = \infty$, the operator norm is the maximum absolute row sum norm. The time required to compute both of these norms is linearly related to the number of elements in the weight matrix. When $p = 2$, the operator norm is given by the largest singular value of the weight matrix—the spectral norm—which can be approximated relatively quickly using a small number of iterations of the power method.

3.1.2 Convolutional Layers

Convolutional layers, $\phi^{conv}(X)$, also perform an affine transformation, but it is usually more convenient to express the computation in terms of discrete

convolutions and point-wise additions. For a convolutional layer, the i-th output feature map is given by

$$\phi_i^{conv}(X) = \sum_{j=1}^{M_{l-1}} F_{i,j} * X_j + B_i, \tag{3.8}$$

where each $F_{i,j}$ is a filter, each X_j is an input feature map, B_i is an appropriately shaped bias tensor exhibiting the same value in every element, and the previous layer produced M_{l-1} feature maps.

The convolutions in Equation 3.8 are linear operations, so one can exploit the isomorphism between linear operations and square matrices of the appropriate size to reuse the matrix norms derived in Section 3.1.1. To represent a single convolution operation as a matrix–vector multiplication, the input feature map is serialised into a vector, and the filter coefficients are used to construct a doubly block circulant matrix. Due to the structure of doubly block circulant matrices, each filter coefficient appears in each column and row of this matrix exactly once. Consequently, the ℓ_1 and ℓ_∞ operator norms are the same and given by $\|F_{i,j}\|_1$, the sum of the absolute values of the filter coefficients used to construct the matrix.

Summing over several different convolutions associated with different input feature maps and the same output feature map, as done in Equation 3.8, can be accomplished by horizontally concatenating matrices. For example, suppose $V_{i,j}$ is a matrix that performs a convolution of $F_{i,j}$ with the j-th feature map serialised into a vector. Equation 3.8 can now be rewritten in matrix form as

$$\phi_i^{conv}(\vec{x}) = [V_{1,1}\ V_{1,2}\ \ldots\ V_{1,M_{l-1}}]\vec{x} + \vec{b}_i, \tag{3.9}$$

where the inputs and biases, previously represented by X and B_i, have been serialised into vectors $\vec{x}$ and $\vec{b}_i$, respectively. The complete linear transformation, W, performed by a convolutional layer to generate M_l output feature

maps can be constructed by adding additional rows to the block matrix:

$$W = \begin{bmatrix} V_{1,1} & \dots & V_{1,M_{l-1}} \\ \vdots & \ddots & \\ V_{M_l,1} & & V_{M_l,M_{l-1}} \end{bmatrix}. \tag{3.10}$$

To compute the ℓ_1 and ℓ_∞ operator norms of W, recall that the operator norm of $V_{i,j}$ for $p \in \{1, \infty\}$ is $\|F_{i,j}\|_1$. A second matrix, W', can be constructed from W, where each block, $V_{i,j}$, is replaced with the corresponding operator norm, $\|F_{i,j}\|_1$. Each of these operator norms can be thought of as a partial row or column sum for the original matrix, W. Now, based on the discussion in Section 3.1.1, the ℓ_1 operator norm is given by

$$\|W\|_1 = \max_j \sum_{i=1}^{M_l} \|F_{i,j}\|_1, \tag{3.11}$$

and the ℓ_∞ operator norm is given by

$$\|W\|_\infty = \max_i \sum_{j=1}^{M_{l-1}} \|F_{i,j}\|_1. \tag{3.12}$$

Yoshida and Miyato (2017) and Miyato et al. (2018) both investigate the effect of penalising or constraining the spectral norm of convolutional layers by reinterpreting the weight tensor of a convolutional layer as a matrix,

$$U = \begin{bmatrix} \vec{u}_{1,1} & \dots & \vec{u}_{1,M_{l-1}} \\ \vdots & \ddots & \\ \vec{u}_{M_l,1} & & \vec{u}_{M_l,M_{l-1}} \end{bmatrix}, \tag{3.13}$$

where each $\vec{u}_{i,j}$ contains the elements of the corresponding $F_{i,j}$ serialised into a row vector. They then proceed to compute the spectral norm of U, rather than computing the spectral norm of W, given in Equation 3.10, claiming that this is the spectral norm of a convolutional layer. However, subsequent work by Tsuzuku et al. (2018) has shown that the method of Yoshida and Miyato

(2017) greatly overestimates the true spectral norm.

Here, we consider how to more accurately compute the spectral norm of a convolutional layer, based on W. Explicitly constructing W and applying a conventional singular value decomposition to compute the spectral norm is infeasible, but we show how the power method can be adapted to use standard convolutional network primitives to compute it efficiently. Consider the usual process for computing the largest singular value of a square matrix using the power method, provided in Algorithm 1. The expression of most interest to us is inside the for loop, namely

$$\vec{x}_i = W^T W \vec{x}_{i-1}, \tag{3.14}$$

which, due to the associativity of matrix multiplication, can be broken down into two steps:

$$\vec{x}'_i = W \vec{x}_{i-1} \tag{3.15}$$

and

$$\vec{x}_i = W^T \vec{x}'_i. \tag{3.16}$$

When W is the matrix in Equation 3.10, the expressions given in Equations 3.15 and 3.16 correspond to a forward propagation and a backwards propagation through a convolutional layer, respectively. Thus, if we replace these matrix multiplications with convolution and transposed convolution operations respectively, as implemented in many deep learning frameworks, the spectral norm can be computed efficiently. Note that only a single vector must undergo the forward and backward propagation operations, rather than an entire batch of instances. This means, for most cases, only a small increase in runtime will be incurred by using this method. It also automatically takes into account the padding and stride hyperparameters used by the convolutional layer.

Algorithm 1 Power method for producing the largest singular value, σ_{max}, of a non-square matrix, W.

Randomly initialise $\vec{x}_0$
for $i = 1$ **to** n **do**
  $\vec{x}_i \leftarrow W^T W \vec{x}_{i-1}$
end for
$\sigma_{max} \leftarrow \frac{\|W\vec{x}_n\|_2}{\|\vec{x}_n\|_2}$

3.1.3 Pooling Layers and Activation Functions

Most common activation functions and pooling operations are, at worst, 1-Lipschitz with respect to all p-norms. For example, the maximum absolute subgradient of the ReLU activation function is 1, which means that ReLU operations have a Lipschitz constant of one. A similar argument yields that the Lipschitz constant of max pooling layers is one. The Lipschitz constant of the softmax is one (Gao and Pavel, 2017).

3.1.4 Residual Connections

Recently developed feed-forward architectures often include residual connections between non-adjacent layers (He et al., 2016). These are most commonly used to construct structures known as residual blocks,

$$\phi^{res}(\vec{x}) = \vec{x} + (\phi_{j+n} \circ ... \circ \phi_{j+1})(\vec{x}), \tag{3.17}$$

where the function composition may contain a number of different linear transformations and activation functions. In most cases the composition is formed by two convolutional layers, each preceded by a batch normalisation layer and a ReLU function. While networks that use residual blocks still qualify as feed-forward networks, they no longer conform to the linear chain of function compositions we formalised in Equation 2.1. Fortunately, networks with residual connections are usually built by composing a linear chain of residual blocks of the form given in Equation 3.17. Hence, the Lipschitz constant of a residual network will be the product of Lipschitz constants for each residual

block, and each block is a sum of two functions as stated in Equation 3.17. For a k_1-Lipschitz function, f_1, and a k_2-Lipschitz function, f_2, we are interested in the Lipschitz constant of their sum:

$$\|(f_1(\vec{x}_1) + f_2(\vec{x}_1)) - (f_1(\vec{x}_2) + f_2(\vec{x}_2))\|_p, \tag{3.18}$$

which can be rearranged to

$$\|(f_1(\vec{x}_1) - f_1(\vec{x}_2)) + (f_2(\vec{x}_1) - f_2(\vec{x}_2))\|_p. \tag{3.19}$$

The subadditivity property of norms and the Lipschitz constants of f_1 and f_2 can then be used to bound Equation 3.19 from above:

$$\begin{aligned} \|(f_1(\vec{x}_1) - f_1(\vec{x}_2)) + (f_2(\vec{x}_1) - f_2(\vec{x}_2))\|_p &\leq \|f_1(\vec{x}_1) - f_1(\vec{x}_2)\|_p && (3.20)\\ &+ \|f_2(\vec{x}_1) - f_2(\vec{x}_2)\|_p && (3.21)\\ &\leq k_1\|\vec{x}_1 - \vec{x}_2\|_p + k_2\|\vec{x}_1 - \vec{x}_2\|_p && (3.22)\\ &= (k_1 + k_2)\|\vec{x}_1 - \vec{x}_2\|_p. && (3.23) \end{aligned}$$

From this we can see that the Lipschitz constant of the addition of two functions is bounded from above by the sum of their Lipschitz constants. Setting f_1 to be the identity function and f_2 to be a linear chain of function compositions, we arrive at the definition of a residual block as given in Equation 3.17. Noting that the Lipschitz constant of the identity function is one, we can see that the Lipschitz constant of a residual block is bounded by

$$L(\phi^{res}) \leq 1 + \prod_{i=j+1}^{j+n} L(\phi_i), \tag{3.24}$$

where the property given in Equation 3.2 has been applied to the function compositions.

3.1.5 Batch Normalisation

Batch normalisation is an operation often included in networks because it tends to improve the conditioning of the optimisation process. It can be expressed as

$$\phi^{bn}(\vec{x}) = \text{diag}\left(\frac{\vec{\gamma}}{\sqrt{\text{Var}[\vec{x}]}}\right)(\vec{x} - \text{E}[\vec{x}]) + \vec{\beta}, \tag{3.25}$$

where $\text{diag}(\cdot)$ denotes a diagonal matrix, and $\vec{\gamma}$ and $\vec{\beta}$ are learned parameters. This can be seen as performing an affine transformation with a linear transformation term

$$\text{diag}\left(\frac{\vec{\gamma}}{\sqrt{\text{Var}[\vec{x}]}}\right)\vec{x}. \tag{3.26}$$

Based on the operator norm of this diagonal matrix, the Lipschitz constant of a batch normalisation layer, with respect to p-norms where $p \in \{1, 2, \infty\}$, is given by

$$L(\phi^{bn}) = \max_i \left| \frac{\vec{\gamma}_i}{\sqrt{\text{Var}[\vec{x}_i]}} \right|. \tag{3.27}$$

Thus, if one wishes to measure the Lipschitz constant of a network, the contribution of the $\vec{\gamma}$ parameters in batch normalisation operations must also be taken into account. The other component that is needed is an estimate of the variance of the activations in the previous layer. We use the moving average estimate of the variance when computing the operator norm in Equation 3.27, rather than the variance computed solely on the current minibatch of training examples, because this is the value used at test time.

3.1.6 Dropout

In the standard formulation of dropout, one corrupts the activations of a layer during training by performing pointwise multiplication with a vector, $\vec{\sigma}$, of Bernoulli random variables with a mean of p. As a consequence, when making a prediction at test time—when units are not dropped out—the activations must be scaled by the probability that they remained uncorrupted during

training,

$$\phi^{do}(\vec{x}) = \mathbb{E}_{\vec{\sigma}}[\vec{\sigma} \odot \vec{x}] \tag{3.28}$$

$$= p\vec{x}, \tag{3.29}$$

where $\odot$ indicates pointwise multiplication. This causes the activation magnitudes at both test time and training time to be approximately the same. The majority of modern neural network make extensive use of rectified linear units. From the absolute homogeneity of the ReLU activation function, we have

$$(\phi^{do} \circ \phi^{relu} \circ \phi^{W,\vec{b}})(\vec{x}) = p \max(0, W\vec{x} + \vec{b}) \tag{3.30}$$

$$= \max(0, pW\vec{x} + p\vec{b}), \tag{3.31}$$

where $\phi^{W,\vec{b}}$ is any linear (e.g., fully connected and convolutional) layer parameterised by W and $\vec{b}$. By definition, scaling the weight matrix will also scale the operator norm, and therefore the Lipschitz constant of that layer. This scaling must be taken into account when considering the Lipschitz constant of a layer.

Chapter 4

Constraining the Lipschitz Constant of Neural Networks

Several papers from the literature on statistical learning theory have presented bounds on the worst-case generalisation performance of neural networks that depend to a significant extent on the Lipschitz constant of the network with respect to the input (Neyshabur et al., 2018; Bartlett et al., 2017). Using the composition property of Lipschitz functions, it has been shown in Chapter 3 that the Lipschitz constant of a network is bounded from above by the product of the Lipschitz constants of the layers. Thus, controlling the Lipschitz constant of a network, and therefore influencing the generalisation error, can be accomplished by constraining the Lipschitz constant of each layer in isolation through the use of constrained optimisation methods. This chapter derives an algorithm that is capable of solving such a constrained optimisation problem, and explores the effect of this form of regularisation on the generalisation error of neural network classifiers. It is found that this new algorithm successfully improves generalisation when combined with batch normalisation, and the effectiveness is particularly pronounced when only a small number of training examples are available. Experimental results demonstrating the sensitivity of the newly introduced hyperparameters are also provided.

4.1 Enforcing the Constraint

Typically, when a neural network is used to solve a classification task, it is trained to minimise the negative log likelihood of the labels given the observed data in the training set,

$$nll(\hat{\vec{y}}_i, \vec{y}_i) = -\sum_{j=1}^{C} \vec{y}_{i,j} \log(\hat{\vec{y}}_{i,j}), \tag{4.1}$$

where $\hat{\vec{y}}_i$ is a vector of class membership probabilities produced by the neural network for instance i, $\vec{y}_i$ is the ground truth class distribution (in most cases a one-hot vector) for the same instance, and C is the number of possible classes. There are three common ways to incorporate a regulariser into such an empirical risk minimisation problem: adding a term to the loss function that penalises certain parameter settings, adding hard constraints that force parameters to come from a particular set, and modifying the optimisation algorithm such that a more desirable local minimum is found. An example of the first approach is the weight decay approach commonly used when training neural networks; more modern regularisation techniques like batch normalisation and dropout are examples of the third category. The method presented in this chapter follows the second approach. This approach was partially motivated by the observation that training generative adversarial networks (Goodfellow et al., 2014; Arjovsky et al., 2017) and improving adversarial robustness (Huster et al., 2018) can see benefits from a hard constraint on the Lipschitz constant of the neural network. The constrained optimisation problem that is to be solved is

$$\min_{W_{1:l}} \frac{1}{N} \sum_{i=1}^{N} nll(f(\vec{x}_i, W_{1:l}), \vec{y}_i) \tag{4.2}$$

$$s.t. \forall j \leq l \in \mathbb{N} : \|W_j\|_p \leq \lambda_j,$$

where f is a neural network parameterised by weight matrices, $W_{1:l}$, that each correspond to one of the l layers in the network, and λ_j is a layer-specific

hyperparameter. In practice, for each network trained in the experiments discussed in Section 4.2, a single λ is used for all layers of the same type. That is, λ_{conv} is chosen for all convolutional layers, λ_{fc} is chosen for fully connected layers, and λ_{bn} is selected for batch normalisation operations.

A straightforward and efficient way to adapt deep learning optimisation methods to solve constrained optimisation problems of the type given in Equation 4.2 is to introduce a projection step after each weight update and perform a variant of the projected stochastic gradient method. In this particular problem, because each parameter matrix is constrained in isolation, it is straightforward to project any infeasible parameter values back into the set of feasible matrices. Specifically, after each weight update step, one must check that none of the weight matrices (including the filter banks in the convolutional layers) are violating the Lipschitz constant constraints. If a weight update causes a weight matrix to leave the feasible set, the resulting matrix must be replaced with the closest matrix that does lie in the feasible set. This can be accomplished with the projection function,

$$\pi(W, \lambda) = \frac{1}{\max(1, \frac{\|W\|_p}{\lambda})} W, \tag{4.3}$$

which will leave the matrix untouched if it does not violate the constraint, and project it back to the closest matrix in the feasible set if it does. Closeness is measured by the matrix distance metric induced by taking the operator norm of the difference between two matrices. This will work with any valid operator norm, because all norms are absolutely homogeneous.[1] In particular, it will work with the operator norms with $p \in \{1, 2, \infty\}$, which can be computed using the approaches outlined in Chapter 3.

Pseudocode for this projected gradient method is given in Algorithm 2. In the experiments in Section 4.2, one can observe fast convergence when using the Adam update rule (Kingma and Ba, 2014), but other variants of the stochastic gradient method also work. For example, it is also shown that

[1] $\alpha\|W\|_p = \|\alpha W\|_p, \forall p \in [1, \infty]$

Algorithm 2 Projected stochastic gradient method to optimise a neural network subject to the Lipschitz Constant Constraint (LCC).

$t \leftarrow 0$
while $W_{1:l}^{(t)}$ not converged **do**
  $t \leftarrow t + 1$
  $g_{1:l}^{(t)} \leftarrow \nabla_{W_{1:l}} f(W_{1:l}^{(t-1)})$
  $\widehat{W}_{1:l}^{(t)} \leftarrow update(W_{1:l}^{(t-1)}, g_{1:l}^{(t)})$
  for $i = 1$ **to** l **do**
    $W_i^{(t)} \leftarrow \pi(\widehat{W}_i^{(t)}, \lambda_i)$
  end for
end while

stochastic gradient descent with Nesterov's momentum is compatible with the approach presented in this chapter when training wide residual networks.

A natural question to ask is which p-norm should be chosen when using the training procedure given in Algorithm 2. The Euclidean (i.e., spectral) norm is often seen as the default choice, due to its special status when talking about distances in the real world. Like Yoshida and Miyato (2017), the technique presented in this chapter uses the power method to estimate the spectral norms of the linear operations in deep networks. The convergence rate of the power method is related to the ratio of the two highest singular values, $\frac{\sigma_2}{\sigma_1}$ (Larson, 2016). If the two largest singular values are almost the same, it will converge very slowly. Because each iteration of the power method for computing the spectral norm of a convolutional layer requires both forward propagation and backward propagation, it is only feasible to perform a small number of iterations and maintain acceptable training time. However, regardless of the quality of the approximation, we can be assured that the approximation is an underestimate of the true norm: the expression in the final line of Algorithm 1 is maximised when $\vec{x}_n$ is the first eigenvector of W. Therefore, if the algorithm has not converged $\vec{x}_n$ will not be an eigenvector of W and the approximation of σ_{max} will be an underestimate.

In contrast to the spectral norm, the values of the ℓ_1 and ℓ_∞ norms are computed exactly. This calculation is done in time linear in the number of weights in a layer, so it always comprises a relatively small fraction of the

overall runtime for training the network. However, it may be the case that the ℓ_1 and ℓ_∞ constraints do not provide as suitable an inductive bias as the ℓ_2 constraint. This is something investigated in the experimental evaluation.

4.2 Experiments

The experiments in this section aim to answer several questions about the behaviour of the Lipschitz Constant Constraint (LCC) regularisation scheme presented in this chapter. The question of most interest is how well this regularisation technique compares to the state-of-the-art, in terms of accuracy measured on held-out data. In addition to this, experiments are performed that demonstrate how sensitive the method is to the choice of values of the λ hyperparameters, how it interacts with existing regularisation methods, and how the additional inductive bias imposed on the learning system impacts the sample efficiency.

Several different network architectures are used throughout the experiments. Specifically, multi-layer perceptrons, VGG-style convolutional networks, and networks with residual connections are used. This is to ensure that the regularisation method works for a broad range of feed-forward architectures. SGD with Nesterov momentum is used for training networks with residual connections, and the Adam optimiser (Kingma and Ba, 2014) is used otherwise. All regularisation hyperparameters for the convolutional networks were optimised on a per-layer type basis using the hyperopt package[2] of Bergstra et al. (2015), which uses a tree-structured Parzen estimator in conjunction with Bayesian optimisation. Separate dropout, spectral decay, and λ hyperparameters were optimised for fully connected and convolutional layers. All network weights were initialised using the method of Glorot and Bengio (2010), and the estimated accuracy reported in all tables is the mean of five networks that were each initialised using different seeds, unless stated

[2] `https://github.com/hyperopt/hyperopt`

otherwise. The standard deviation is also reported to give an idea of how robust different regularisers are to different initialisations. The code for running these experiments is available online.[3]

4.2.1 Synthetic Data

The first experiment illustrates several interesting points using a simple synthetic dataset generated using the relation

$$y = \sin(x) + \frac{1}{5}\cos(19x). \tag{4.4}$$

This function was chosen because there are two trends: one with a large amplitude that varies slowly, and another that varies quickly and has a much smaller amplitude. A model with a reasonably small Lipschitz constant should largely ignore the high frequency signal and model only the component with low frequency and high amplitude. Because the function is a scalar function of one variable, it is also very convenient to visualise.

A training dataset was generated by randomly sampling 1,000 points from a uniform distribution covering the range $[-5, 5]$. Each of these points was then labelled by applying the generating function in Equation 4.4. Lipschitz continuous neural networks with several different λ values were trained for each of the three p-norms discussed earlier. Fully connected networks with two hidden layers, each containing 1,000 hidden units, were used. Figures 4.1, 4.2, and 4.3 show the result of training these networks on the synthetic training set using $p = 1$, $p = 2$, and $p = \infty$, respectively.

For all three norms, when λ is set to one, the function approximates the training data very poorly, failing to even capture the general trend provided by the sine function. As λ is increased, the networks approximate the function used to generate the dataset with higher accuracy. As expected, for each norm, there is a window where only the low frequency component of the function is

[3] `https://github.com/henrygouk/keras-lipschitz-networks`

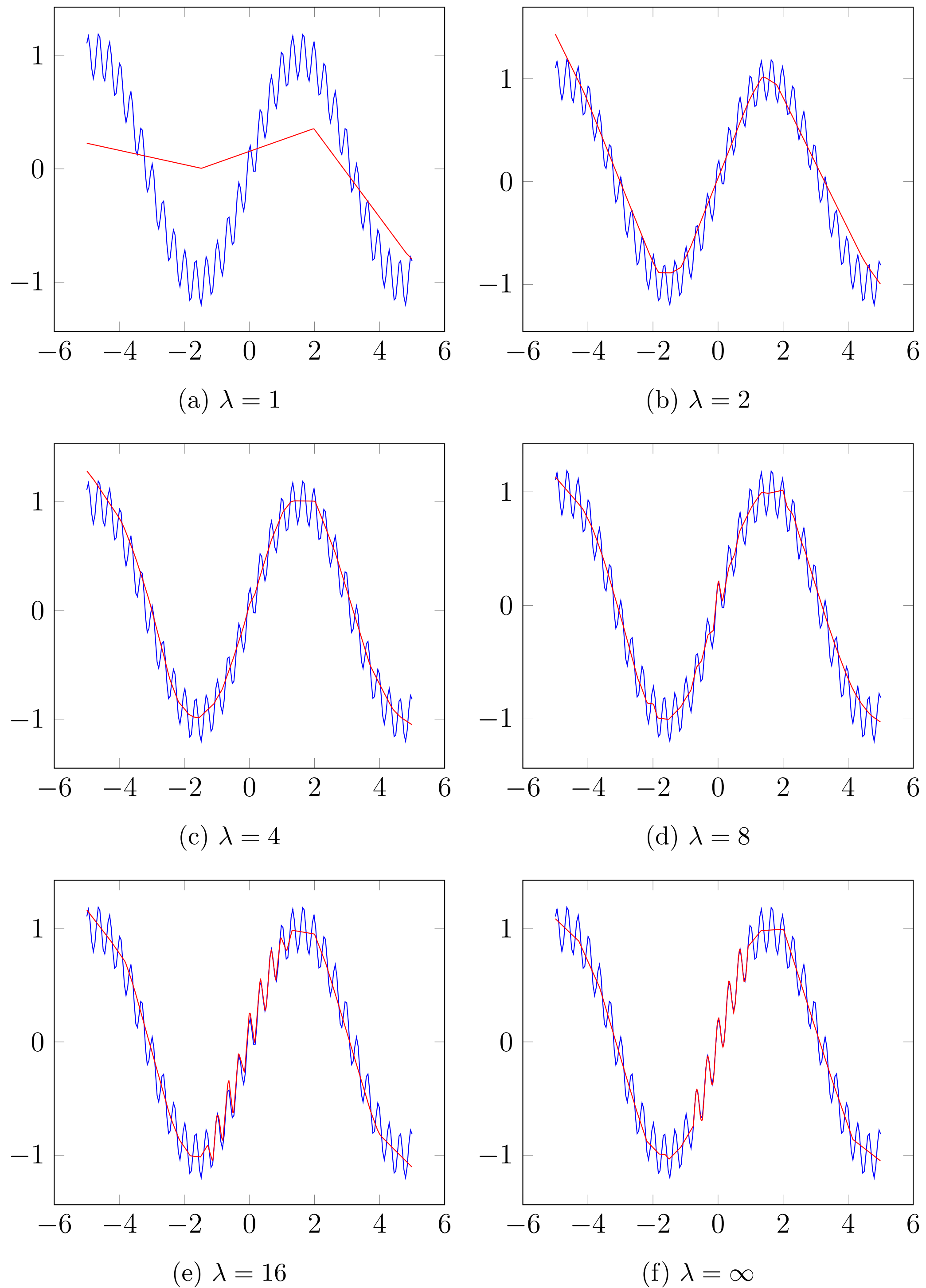
well approximated. This illustrates that the Lipschitz constant can act as a well-behaved measure of model complexity.

One might expect that, because the data generating process is a scalar function, all three norms would behave the same, as they are all equivalent to taking the absolute value in the single dimensional case. Despite this, one can clearly observe varying levels of approximation quality when comparing the use of different norms with the same value for λ. In the case of the ℓ_2 norm, this is fairly easy to explain: the spectral norm of each layer is only an approximation found using the power method, which, as discussed earlier, is liable to underestimate the true spectral norm if only a few iterations are used. It is possible that the difference between the ℓ_1 and ℓ_∞ norm approaches—most noticeable when λ is two—is explained by a less obvious phenomenon. Because of the difference in the operator norms for the ℓ_1 and ℓ_∞ vector norms, the set of permissible weight matrices is different. This is something that is very likely to impact the trajectory of the weights during optimisation, and hence result in quite different solutions. Optimisation dynamics are potentially also the reason the function is poorly approximated when λ is one—even though the sine function is 1-Lipschitz,[4] but this could also be due to the regulariser constraining an upper bound to the Lipschitz constant, rather than the tightest possible constant. The work of Huster et al. (2018) discusses how the particular upper bound used by LCC can suffer from this problem in some situations.

4.2.2 CIFAR-10

The CIFAR-10 dataset (Krizhevsky and Hinton, 2009) contains 60,000 tiny images, each belonging to one of 10 classes. The experiments in this section follow the common protocol of using 10,000 of the images in the 50,000 image training set for tuning the model hyperparameters. Two network architectures are considered for this dataset: a VGG19-style network (Simonyan and Zisserman, 2014), resized to be compatible with the 32×32 pixel images in

[4] $L(\sin) = \max |\delta_x \sin(x)| = \max | \cos(x)| = 1$

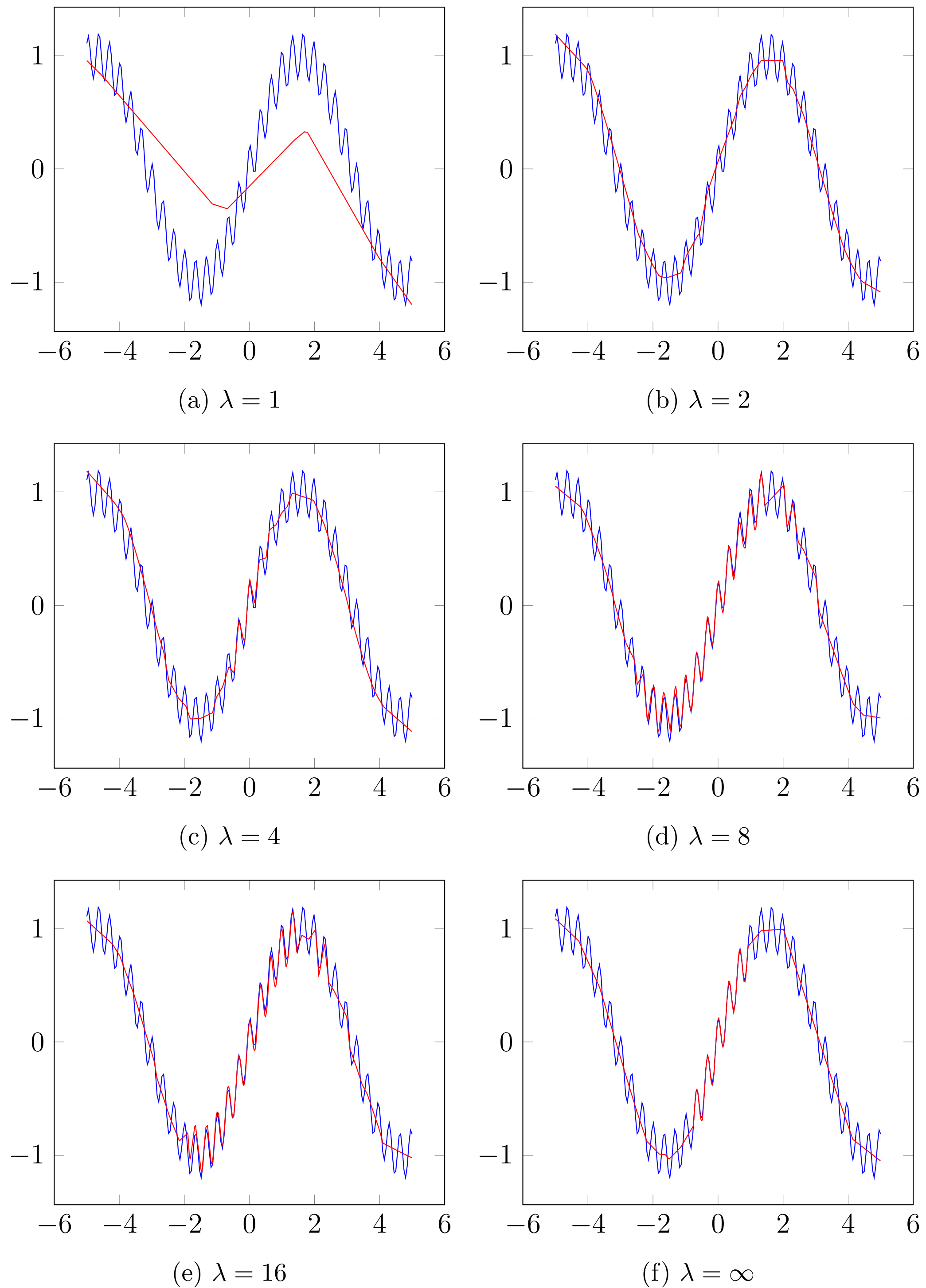


(a) $\lambda = 1$ (b) $\lambda = 2$

(c) $\lambda = 4$ (d) $\lambda = 8$

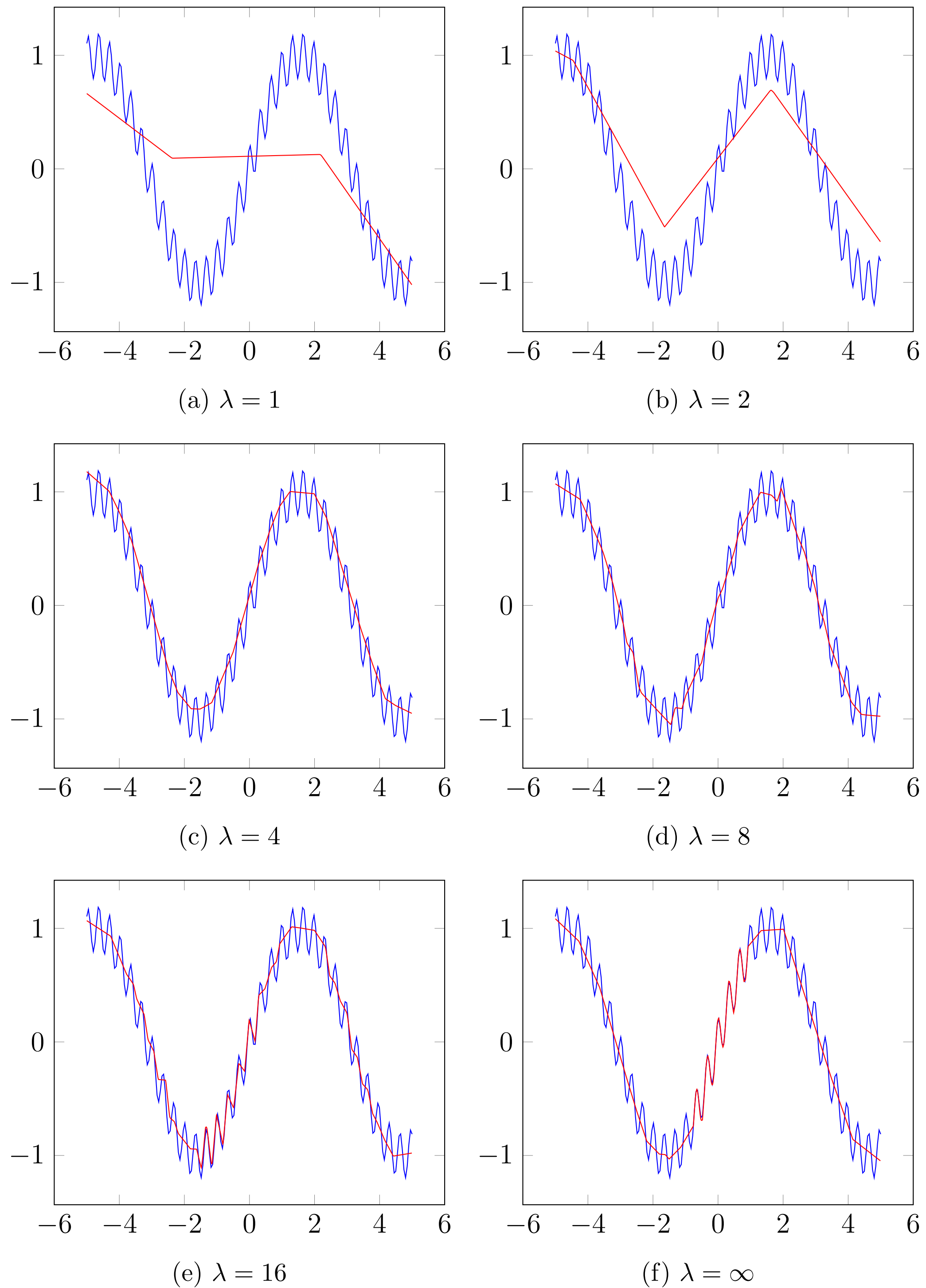
(e) $\lambda = 16$ (f) $\lambda = \infty$

Figure 4.1: Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_1 norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network.



(a) $\lambda = 1$ (b) $\lambda = 2$ (c) $\lambda = 4$ (d) $\lambda = 8$ (e) $\lambda = 16$ (f) $\lambda = \infty$

Figure 4.2: Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_2 norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network.



(a) $\lambda = 1$ (b) $\lambda = 2$

(c) $\lambda = 4$ (d) $\lambda = 8$

(e) $\lambda = 16$ (f) $\lambda = \infty$

Figure 4.3: Visualisation of neural networks trained on a 1-dimensional synthetic dataset with different values of λ. The ℓ_∞ norm was used for regularising these networks. The blue dots are the training data, and the red lines are the predictions made by each network.

CIFAR-10, and a Wide Residual Network (WRN) (Zagoruyko and Komodakis, 2016). All experiments on this dataset utilise data augmentation in the form of random crops and horizontal flips, and the image intensities were rescaled to fall into the $[-1, 1]$ range. Each of the VGG networks were trained for 140 epochs using the Adam optimiser (Kingma and Ba, 2014). The initial learning rate was set to 10^{-4} and decreased by a factor of 10 after epoch 100 and epoch 120. The WRNs were trained for a total of 200 epochs using the stochastic gradient method with Nesterov's momentum. The learning rate was initialised to 0.1, and decreased by a factor of 5 at epochs 60, 120, and 160.

The performance of LCC is compared to dropout, batch normalisation, and the spectral decay method of Yoshida and Miyato (2017). Dropout and batch normalisation are the two most widely used regularisation schemes, often acting as key components of state-of-the-art models (Simonyan and Zisserman, 2014; He et al., 2016; Zagoruyko and Komodakis, 2016), and the spectral decay method has a similar goal to the ℓ_2 instantiation of our method: encouraging the spectral norm of the weight matrices to be small. For this particular experiment, each regulariser is considered in isolation, and we also consider combinations of LCC with dropout and batch normalisation. Results are given in Tables 4.1 and 4.2 for the VGG and WRN architectures, respectively. Interestingly, the performance of the VGG network varies considerably more than that of the Wide Residual Network. VGG models trained with spectral decay, dropout, and LCC exhibit no performance gain over the unregularised baseline; however, when combined with batch normalisation, there is a sizeable increase in accuracy—greater than the improvement provided by batch normalisation alone. In the case of WRNs, LCC performs similarly to dropout, but there is little separation between methods on this dataset.

4.2.3 CIFAR-100

CIFAR-100, like CIFAR-10, is a dataset of 60,000 tiny images, but contains 100 classes rather than 10. The same data augmentation methods used for

Table 4.1: Performance of VGG19 networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-10. LCC-ℓ_p denotes the Lipschitz Constant Constraint method for a given p-norm.

Method	VGG19
None	88.12% ± 0.34
Spectral Decay	88.15% ± 0.34
Batchnorm	90.43% ± 0.17
Dropout	88.14% ± 0.17
Batchnorm + Dropout	90.46% ± 0.10
LCC-ℓ_1	88.33% ± 0.07
LCC-ℓ_2	88.00% ± 0.12
LCC-ℓ_∞	88.03% ± 0.12
Dropout + LCC-ℓ_1	88.26% ± 0.10
Dropout + LCC-ℓ_2	88.29% ± 0.21
Dropout + LCC-ℓ_∞	88.12% ± 0.16
Batchnorm + LCC-ℓ_1	92.48% ± 0.13
Batchnorm + LCC-ℓ_2	92.57% ± 0.28
Batchnorm + LCC-ℓ_∞	91.64% ± 0.20
Dropout + Batchnorm + LCC-ℓ_1	91.72% ± 0.17
Dropout + Batchnorm + LCC-ℓ_2	91.23% ± 0.24
Dropout + Batchnorm + LCC-ℓ_∞	92.71% ± 0.29

CIFAR-10 are also used for training models on CIFAR-100—random crops and horizontal flips. Once again, WRNs and VGG19-style networks ared trained on this dataset. The learning rate schedules used in the CIFAR-10 experiments also worked well on this dataset, which is not surprising given their similarities. However, the regularisation hyperparameters were optimised specifically for CIFAR-100. The results for the VGG networks are given in Table 4.3, and the WRN results can be found in Table 4.4.

It can be seen that combining the Lipschitz-based regularisation scheme with batch normalisation is an effective technique for improving generalisation of networks both with and without residual connections. For the VGG networks, the combination of LCC with batch normalisation provides a sizeable increase in accuracy. For the WRN models, more modest gains are provided by the combination of LCC and dropout. Spectral decay performs consistently worse than LCC-ℓ_2.

Table 4.2: Accuracy of Wide Residual Networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-10. All WRNs are trained with batch normalisation.

Method	WRN-16-10
None	95.13% ± 0.17
Spectral Decay	95.21% ± 0.08
Dropout	95.46% ± 0.20
LCC-ℓ_1	95.34% ± 0.21
LCC-ℓ_2	95.32% ± 0.15
LCC-ℓ_∞	95.82% ± 0.19
Dropout + LCC-ℓ_1	95.47% ± 0.15
Dropout + LCC-ℓ_2	95.57% ± 0.15
Dropout + LCC-ℓ_∞	95.68% ± 0.11

4.2.4 MNIST and Fashion-MNIST

The Fashion-MNIST dataset (Xiao et al., 2017) is designed as a more challenging drop-in replacement for the original MNIST dataset of hand-written digits (LeCun et al., 1998). Both contain $70,000$ greyscale images labelled with one of 10 possible classes. The last 10,000 instances are used as the test set. The final 10,000 instances in the training set are used for measuring performance when optimising the regularisation hyperparameters. In these experiments, small convolutional networks are trained on both of these datasets with different combinations of regularisers. The networks contain only two convolutional layers, each consisting of 5×5 kernels, and both layers are followed by 2×2 max pooling layers. The first layer has 64 feature maps, and the second has 128. These layers feed into a fully connected layer with 128 units, which is followed by the output layer with 10 units. ReLU activations are used for all hidden layers, and each model is trained for 60 epochs using Adam (Kingma and Ba, 2014). The learning rate was started at 10^{-4} and decreased by a factor of 10 at the fiftieth epoch.

The test accuracies for each of the models trained on these datasets are given in Table 4.5. For both datasets, the LCC methods alone provide no gain in performance. When combined with dropout, batchnorm, or a combination of both, there are small gains over the baselines. Once again, spectral decay

Table 4.3: Performance of networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-100. LCC-ℓ_p denotes our Lipschitz Constant Constraint method for some given p-norm.

Method	VGG19
None	57.40% ± 0.70
Spectral Decay	57.04% ± 0.74
Batchnorm	65.46% ± 0.43
Dropout	57.49% ± 0.80
Batchnorm + Dropout	66.75% ± 0.40
LCC-ℓ_1	58.31% ± 0.88
LCC-ℓ_2	59.73% ± 0.65
LCC-ℓ_∞	57.97% ± 0.88
Dropout + LCC-ℓ_1	58.09% ± 0.67
Dropout + LCC-ℓ_2	60.05% ± 0.65
Dropout + LCC-ℓ_∞	58.28% ± 0.62
Batchnorm + LCC-ℓ_1	69.59% ± 0.29
Batchnorm + LCC-ℓ_2	68.25% ± 0.38
Batchnorm + LCC-ℓ_∞	69.16% ± 0.22
Dropout + Batchnorm + LCC-ℓ_1	70.17% ± 0.21
Dropout + Batchnorm + LCC-ℓ_2	71.76% ± 0.26
Dropout + Batchnorm + LCC-ℓ_∞	69.25% ± 0.43

is slightly worse than LCC-ℓ_2.

4.2.5 Street View House Numbers

The Street View House Numbers dataset contains over $600,000$ images of digits extracted from Google's Street View platform. Each image contains three colour channels and has a resolution of 32×32 pixels. As with the previous datasets, the only preprocessing performed is to rescale the input features to the range $[-1, 1]$. However, in contrast to the experiments on CIFAR-10 and CIFAR-100, no data augmentation is performed while training on this dataset. The first network architecture used for this dataset, which follows a VGG-style structure, is comprised of four conv–conv–maxpool blocks with 64, 128, 192, and 256 feature maps, respectively. This is followed by two fully connected layers, each with 512 units, and then the logistic regression layer. Due to the large training set size, it was only necessary to train for 20 epochs. The Adam optimiser (Kingma and Ba, 2014) was used with an initial learning rate

Table 4.4: Accuracy of Wide Residual Networks trained with spectral decay, dropout, LCC, and combinations thereof on CIFAR-100. All WRNs are trained with batch normalisation.

Method	WRN-16-10
None	77.94% ± 0.33
Spectral Decay	77.93% ± 0.20
Dropout	77.98% ± 0.24
LCC-ℓ_1	78.16% ± 0.04
LCC-ℓ_2	79.00% ± 0.33
LCC-ℓ_∞	79.39% ± 0.28
Dropout + LCC-ℓ_1	79.08% ± 0.11
Dropout + LCC-ℓ_2	79.45% ± 0.26
Dropout + LCC-ℓ_∞	78.17% ± 1.86

of 10^{-4}, which was decreased by a factor of 10 at epochs 15 and 18. Results demonstrating the performance of individual regularisers, and the combination of LCC with dropout and batch normalisation, are given in Table 4.6. Small WRN models are also trained on this dataset. Once again, due to the large size of the training set, it is sufficient to only train each network for 20 epochs in total. Therefore, compared to the WRNs trained on CIFAR-10 and CIFAR-100, a compressed learning rate schedule is used. The learning rate is started at 0.1, and is decreased by a factor of 5 at epochs 6, 12, and 16. Test set performance for each of the WRN models trained on SVHN is provided in Table 4.7.

In isolation, LCC provides only a negligible improvement in test accuracy, which is in line with what has been observed on other datasets so far. Also congruent with the other results is that the performance of LCC combined with batch normalisation on the VGG networks is noticeably higher than other approaches—in this case the best VGG network performs almost the same as the baseline WRN model. For this dataset, both the LCC and spectral decay methods provide no benefit to the residual networks considered.

Table 4.5: Test accuracies of the small convolutional networks trained with spectral decay, dropout, LCC, and combinations thereof on the MNIST and Fashion-MNSIT datasets.

Method	MNIST	Fashion-MNIST
None	99.28% $\pm$ 0.01	92.51% $\pm$ 0.28
Spectral Decay	99.27% $\pm$ 0.06	92.39% $\pm$ 0.10
Batchnorm	99.29% $\pm$ 0.03	92.54% $\pm$ 0.10
Dropout	99.36% $\pm$ 0.03	91.82% $\pm$ 0.12
Batchnorm + Dropout	98.93% $\pm$ 0.17	91.68% $\pm$ 0.17
LCC-ℓ_1	99.20% $\pm$ 0.03	91.41% $\pm$ 0.17
LCC-ℓ_2	99.27% $\pm$ 0.06	92.33% $\pm$ 0.09
LCC-ℓ_∞	99.10% $\pm$ 0.07	91.93% $\pm$ 0.16
Dropout + LCC-ℓ_1	99.34% $\pm$ 0.01	91.54% $\pm$ 0.15
Dropout + LCC-ℓ_2	99.39% $\pm$ 0.02	91.85% $\pm$ 0.19
Dropout + LCC-ℓ_∞	99.33% $\pm$ 0.03	91.91% $\pm$ 0.17
Batchnorm + LCC-ℓ_1	99.41% $\pm$ 0.05	93.06% $\pm$ 0.15
Batchnorm + LCC-ℓ_2	99.41% $\pm$ 0.05	92.62% $\pm$ 0.18
Batchnorm + LCC-ℓ_∞	99.32% $\pm$ 0.09	92.87% $\pm$ 0.12
Dropout + Batchnorm + LCC-ℓ_1	99.35% $\pm$ 0.08	93.23% $\pm$ 0.23
Dropout + Batchnorm + LCC-ℓ_2	99.42% $\pm$ 0.10	91.71% $\pm$ 0.38
Dropout + Batchnorm + LCC-ℓ_∞	99.36% $\pm$ 0.04	92.75% $\pm$ 0.25

4.2.6 Scaled ImageNet Subset (SINS-10)

Many datasets used by the deep learning community consist of a single pre-defined training and test split. For example, in the previous experiments on CIFAR-10, one set of 50,000 images was used for training, and another set of 10,000 images was used for testing. In order to perform a significance test, and thus have some degree of confidence in the results and conclusions drawn from experiments, multiple performance measurements of models trained using a particular algorithm configuration must be gathered. To this end, the Scaled ImageNet Subset (SINS-10) dataset has been assembled as part of the work undertaken in Gouk et al. (2019). This is a set of 100,000 colour images retrieved from the ImageNet collection Deng et al. (2009) that are evenly divided into 10 different classes, and each of these classes is associated with multiple related synsets from the ImageNet database. All images were first resized such that their smallest dimension was 96 pixels and their aspect ratio was maintained. Then, the central 96×96 pixel subwindow of the image was

Table 4.6: Prediction accuracy of VGG-style networks trained with spectral decay, dropout, LCC, and combinations thereof on the SVHN dataset.

Method	VGG
None	96.74% ± 0.06
Spectral Decay	96.83% ± 0.07
Batchnorm	96.90% ± 0.05
Dropout	96.99% ± 0.09
Batchnorm + Dropout	96.98% ± 0.10
LCC-ℓ_1	97.07% ± 0.05
LCC-ℓ_2	96.69% ± 0.07
LCC-ℓ_∞	97.15% ± 0.09
Dropout + LCC-ℓ_1	97.47% ± 0.04
Dropout + LCC-ℓ_2	97.15% ± 0.09
Dropout + LCC-ℓ_∞	97.36% ± 0.04
Batchnorm + LCC-ℓ_1	97.17% ± 0.09
Batchnorm + LCC-ℓ_2	96.94% ± 0.04
Batchnorm + LCC-ℓ_∞	97.35% ± 0.03
Dropout + Batchnorm + LCC-ℓ_1	97.30% ± 0.07
Dropout + Batchnorm + LCC-ℓ_2	97.73% ± 0.30
Dropout + Batchnorm + LCC-ℓ_∞	97.32% ± 0.06

extracted to be used as the final instance.

An important difference between currently available benchmark datasets and the proposed dataset is how it has been split into training and testing data. The entire dataset is divided into 10 equal-size predefined folds of 10,000 instances. The first 9,000 images in each fold are intended for training a model, and the remaining 1,000 for testing it. One can then apply a machine learning technique to each fold in the dataset, and repeat the process for techniques one wishes to compare against. This will result in 10 performance measurements for each algorithm. A paired t-test can then be used to determine whether there is a significant difference, with some level of confidence, between the performance of the different techniques.

Note that the protocol for SINS-10 is different to the commonly used cross-validation technique. When performing cross-validation, the training sets overlap significantly, and the measurements for the test fold performance are therefore not independent. To mitigate this, one can use a heuristic for correcting the paired t-test Nadeau and Bengio (2000). Rather than use this heuristic,

Table 4.7: Prediction accuracies of WRN-16-4 networks trained with spectral decay, dropout, LCC, and combinations thereof on the SVHN dataset.

Method	WRN-16-4
None	97.97% ± 0.04
Spectral Decay	98.02% ± 0.04
Dropout	98.23% ± 0.05
LCC-ℓ_1	98.00% ± 0.06
LCC-ℓ_2	97.93% ± 0.07
LCC-ℓ_∞	98.03% ± 0.05
Dropout + LCC-ℓ_1	98.21% ± 0.02
Dropout + LCC-ℓ_2	98.17% ± 0.05
Dropout + LCC-ℓ_∞	98.24% ± 0.06

one can simply avoid fitting models using overlapping training (or test) sets, and use the standard paired t-test to produce more reliable inferences.

The experiments conducted on SINS-10 in this chapter make use of the same VGG-style and WRN network architectures used for the SVHN experiments. However, because each fold of the SINS-10 dataset has many fewer instances than SVHN, the number of epochs and learning rate schedules are changed. The VGG networks are trained for a total of 60 epochs, beginning with a learning rate of 10^{-4} that is decreased by a factor of 10 at epochs 40 and 50. The WRN models are trained for 100 epochs each, with a starting learning rate of 0.1 that is decreased by a factor of five at epochs 30, 60 and 80. The regularisation hyperparameters are optimised on a per-fold basis, and the final 1,000 instances of the training set are repurposed as a validation set to determine the quality of a given hyperparameter setting. The results for these experiments are given in Table 4.8 for the VGG models, and Table 4.9 for the wide residual network models. Hypothesis tests are carried out to determine wether adding LCC to the combination of regularisers improves performance. For example, the test carried out for Batchnorm + LCC-ℓ_1 makes a comparison with the network trained only with batch normalisation.

The results on this dataset exhibit a similar, yet more exaggerated, trend compared to the results on the other datasets. For the VGG networks, LCC and spectral decay in isolation are mostly ineffective. However, when combined

Table 4.8: Prediction accuracies of VGG-style networks trained with spectral decay, dropout, batchnorm, LCC, and combinations thereof on the SINS-10 dataset. The +/−column indicates whether adding LCC to the combination of regularisers results in a statistically significant improvement in performance at the 95% confidence level.

Method	VGG	+/−
None	59.13% ± 1.55	
Spectral Decay	58.96% ± 1.49	
Dropout	64.83% ± 1.15	
Batchnorm	63.73% ± 1.18	
Batchnorm + Dropout	68.65% ± 1.00	
LCC-ℓ_1	58.74% ± 1.15	
LCC-ℓ_2	58.48% ± 1.79	
LCC-ℓ_∞	58.60% ± 1.83	
Dropout + LCC-ℓ_1	64.66% ± 1.23	
Dropout + LCC-ℓ_2	64.23% ± 1.05	
Dropout + LCC-ℓ_∞	63.13% ± 1.07	−
Batchnorm + LCC-ℓ_1	71.24% ± 1.31	+
Batchnorm + LCC-ℓ_2	69.96% ± 2.16	+
Batchnorm + LCC-ℓ_∞	70.92% ± 2.04	+
Batchnorm + Dropout + LCC-ℓ_1	72.18% ± 1.97	+
Batchnorm + Dropout + LCC-ℓ_2	71.15% ± 1.13	+
Batchnorm + Dropout + LCC-ℓ_∞	70.99% ± 1.03	+

with batch normalisation there is a large gain of over 10% percentage points in accuracy—bringing the average performance of LCC-reglarised VGG networks to a greater accuracy than the unregularised WRN baseline. One place where the results on this dataset do not agree with the results observed on the other tasks is the impressive performance of the spectral decay regularisation method when applied to residual networks. Paired t-tests determined no statistically significant different between LCC-ℓ_2 and spectral decay for the VGG-style and WRN models at the 95% confidence level ($p = 0.566$ and $p = 0.053$, respectively).

4.2.7 Fully Connected Networks

Neural networks consisting exclusively of fully connected layers have a long history of being applied to classification problems arising in data mining scenarios. To evaluate how well the LCC regularisers work on tabular data, we

Table 4.9: Prediction accuracies of WRNs trained with spectral decay, dropout, batchnorm, LCC, and combinations thereof on the SINS-10 dataset. The +/−column indicates whether adding LCC to the combination of regularisers results in a statistically significant improvement in performance at the 95% confidence level.

Method	WRN-16-4	+/−
None	68.26% ± 1.89	
Spectral Decay	76.85% ± 1.29	
Dropout	68.14% ± 2.78	
LCC-ℓ_1	72.85% ± 1.63	+
LCC-ℓ_2	75.89% ± 2.02	+
LCC-ℓ_∞	74.09% ± 2.19	+
Dropout + LCC-ℓ_1	73.27% ± 1.19	+
Dropout + LCC-ℓ_2	77.93% ± 1.19	+
Dropout + LCC-ℓ_∞	76.80% ± 1.05	+

have trained fully connected networks on the classification datasets collected by Geurts and Wehenkel (2005). These datasets are primarily from the University California at Irvine dataset repository. The only selection criterion used by Geurts and Wehenkel (2005) was that they contain only numeric features. In these experiments, each network contains two hidden layers consisting of 100 units, and uses the ReLU activation function. Two repetitions of 5-fold cross validation are performed for each dataset. Hyperparameters for each regulariser were tuned on a per-fold basis using grid search. The accuracy of a particular hyperparameter combination tried during the grid search was determined using a hold-out set drawn from the training data in each fold. The values considered for dropping a unit when using dropout were $p \in \{0.2, 0.3, 0.4, 0.5\}$. The values considered for λ when using the ℓ_2 and ℓ_∞ approaches were $\{2, 4, ..., 18, 20\}$, and for the ℓ_1 variant we used $\{5, 10, ..., 45, 50\}$. Once again, the combination of LCC with each of the regularisation methods is also evaluated.

Because multiple estimates of the test set accuracy are available, hypothesis tests can be performed to determine statistically significant differences in performance of each of the regularisers over the unregularised baseline. The hypothesis testing procedure outlined by Bouckaert and Frank (2004) is used

in order to overcome the problem of overlapping testing and training sets introduced by the crossvalidation procedure. The mean test set accuracies on each dataset are given in Table 4.10, with statistically significant improvements and degredations in accuracy at the 95% confidence level indicated by the +and −symbols, respectively.

Several interesting trends can be found in this table. One particularly surprising trend is that the presence of dropout is a very good indicator of a statistically significant degredation in accuracy. Interestingly, the only exceptions to this are the two synthetic datasets, where dropout is associated with an improvement in accuracy. The combination of batch normalisation with LCC is one of the more reliable approaches to regularisation on the tabular classification datasets considered. In particular, the LCC-ℓ_∞ method combined with batch normalisation achieves the highest mean accuracy on seven of the 10 datasets. On the other three datasets, either all regularisers are ineffective, or LCC-ℓ_∞ is still present in the best combination of regularisation schemes. This provides strong evidence that LCC-ℓ_∞ is a good choice for regularisation of neural network models trained on tabular data.

These results can also be visualised using a critical difference diagram (Demšar, 2006), as shown in Figure 4.4. The average rank of LCC-ℓ_∞ combined with batch normalisation is 2.6, which is considerably higher than the next best method, batch normalisation, with an average rank of 5.9. However, there is insufficient evidence to be able to state that LCC-ℓ_∞ significantly outperforms batch normalisation. Nevertheless, it can also be seen from this diagram that LCC-ℓ_∞ with batch normalisation is statistically significantly better than most of the combinations of regularisers that include dropout.

4.2.8 Sensitivity to λ

The ability to easily tune the hyperparameters of a regularisation method is important. The previous experiments have primarily taken advantage of automated hyperparameter tuning through the use of the hyperopt pack-

Table 4.10: Mean test set accuracies obtained using two repetitions of 5-fold cross validation. Statistically significant improvements and degredations (95% confidence) are marked with the +and −symbols, respectively. The highest mean accuracy achieved on each dataset is bolded.

Method	dig44	letter	pendigits	sat	segment	spambase	twonorm	vehicle	vowel	waveform
None	96.27	93.92	99.41	90.34	95.84	**94.53**	97.15	76.89	82.27	85.48
BN	96.96+	95.37+	99.44	90.82	95.37	94.11	97.16	78.02	86.21+	85.40
DO	95.79−	90.28−	99.14−	89.18−	93.70−	93.88−	97.64+	72.52−	68.18−	86.16
LCC-ℓ_1	96.28	93.74	99.43	90.29	96.04	94.29	97.30	77.48	79.95	86.25
LCC-ℓ_2	96.33	93.83	99.42	90.30	96.04	94.29	97.29	77.48	80.20	85.97
LCC-ℓ_∞	96.42	92.88−	99.37	90.24	95.95	94.46	97.66+	76.24	76.41−	86.90+
BN + LCC-ℓ_1	96.83+	95.29+	99.45	90.75	95.91	94.06	97.10	77.84	82.98	86.00
BN + LCC-ℓ_2	96.85+	95.34+	99.45	90.73	95.89	93.86	97.05	78.07	83.13	85.82
BN + LCC-ℓ_∞	**97.11+**	**96.42+**	**99.52**	**91.00+**	**96.52**	94.37	97.41	**80.14**	**90.86+**	86.51+
DO + LCC-ℓ_1	95.66−	90.27−	99.19	89.21−	93.74−	94.14	97.72+	73.52	68.89−	86.36+
DO + LCC-ℓ_2	95.65−	90.23−	99.19−	89.22−	93.92−	93.96	97.70+	73.58	68.38−	86.53+
DO + LCC-ℓ_∞	95.69−	88.90−	99.04−	89.30	93.42−	94.20	**97.74+**	75.65	65.71−	**86.99+**
DO + BN + LCC-ℓ_1	96.04	91.44−	99.21−	90.08	93.90−	93.68−	97.71+	74.94	70.61	86.71+
DO + BN + LCC-ℓ_2	96.04	91.37−	99.25	89.84	93.85−	93.68	97.68+	74.65	71.21	86.54+
DO + BN + LCC-ℓ_∞	96.81+	93.24−	99.41	90.06	95.52	94.06	97.69+	77.60	77.07	86.59

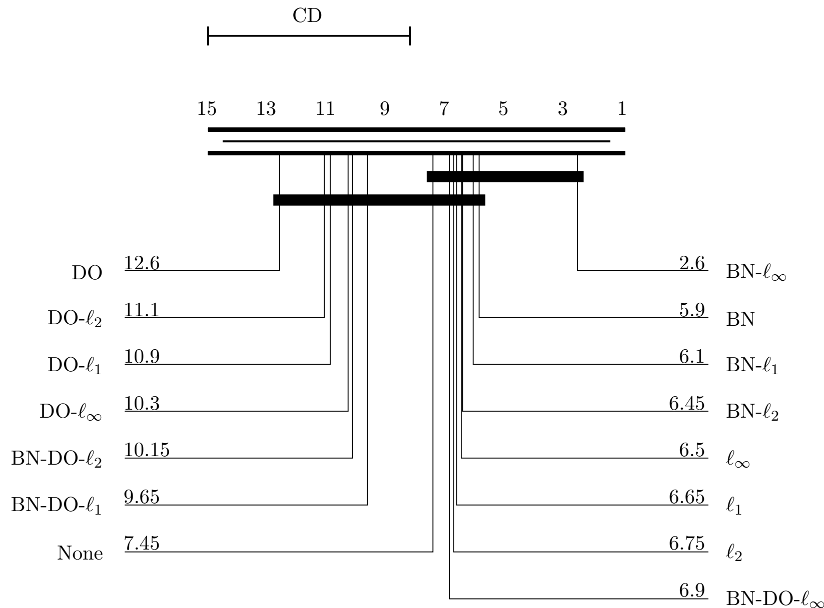


Figure 4.4: A critical difference diagram showing the statistically significant (95% confidence) differences between the average rank of each method. The number beside each method is the average rank of that method across all datasets. The thick black bars overlaid on groups of thin black lines indicate a clique of methods that have not been found to be statistically significantly different.

age (Bergstra et al., 2015), but investigating how sensitive the algorithm is to the choice of λ could lead to useful intuition for both manual hyperparameter tuning and automated methods. The networks that have been trained with LCC regularisation thus far have required up to three different λ hyperparameters—one for each parameterised layer type. Therefore, one cannot simply plot the model accuracy for given values of λ: it is not a scalar quantity. However, one can multiply all three of these hyperparameters by a single scalar value, and vary this scalar quantity to investigate the relationship between hyperparameter magnitude and generalisation performance. Figure 4.5 visualises this relationship using the CIFAR-100 dataset and several models with the VGG19-style architecture. Because previous experiments have demonstrated that using LCC in conjunction with batch normalisation yields

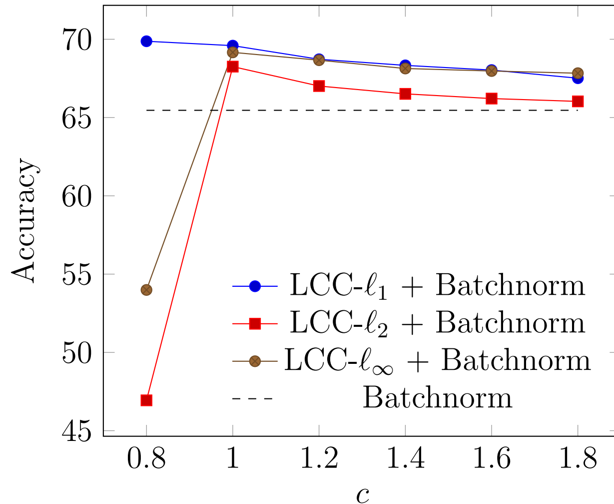


Figure 4.5: This figure demonstrates the sensitivity of the algorithm to the choice of λ for each of the three p-norms when used to regularise VGG19 networks trained on the CIFAR-100 dataset. Because a different hyperparameter was optimised for each layer type, the horizontal axis represents the value of a single constant that is used to scale the three different λ hyperparameters associated with each curve. Note that when $c = 0.6$, the LCC-ℓ_1 network fails to converge.

the best results, that is the only set of configurations that are considered. This plot was generated by defining a hyperparameter vector, $\vec{\lambda} = [\lambda_{conv}, \lambda_{fc}, \lambda_{bn}]$, where each component is set to the value found during the hyperparameter optimisation procedure performed as part of the experiments carried out in Section 4.2.3. Each data point in the plot is created by training a network with hyperparameters specified by $c\vec{\lambda}$, where c is a user-provided scalar value, and plotting the resulting test set accuracy for different values of c.

One trend that is particularly salient in Figure 4.5 is that choosing values for the hyperparameters that are even slightly too small results in a massive degradation in performance. Conversely, when c is set above the optimimal value, each method exhibits a slow decline in performance until the accuracy is comparable to that of a network trained with only batch normalisation. Although this is the type of behaviour one might expect from a sensible means for controlling model capacity, this second phenomenon can cause difficulty during hyperparameter tuning. It is easy to determine when the hyperparameters have been assigned values that are too small, as the model fails to

converge. However, it is not easy to determine by how much the hyperparameters should be increased. It was found that for each dataset, network architecture, and p-norm choice, vastly different hyperparameter settings were chosen by the automated tuning process. This means there is no typical range one should expect the optimal hyperparameters to lie in, and one must use a very uninformative prior when performing hyperparameter optimisation.

4.2.9 Sample Efficiency

Imposing on a learning algorithm additional inductive biases that accurately reflect the underlying relationship between the input and output variables should result in a method that can produce well-performing models with fewer training examples than an algorithm without such inductive biases: more informative inductive biases should result better sample efficiency. To determine if LCC improves the sample efficiency of training neural networks on image data, a series of networks are trained on progressively larger subsets of the CIFAR-10 training set. Informed by the results already presented, only models trained with a combination of the LCC regulariser and batch normalisation are compared to the baselines. The full test set is still used for computing estimates of the accuracy of the resulting models. The learning curves for the VGG-style networks are given in Figure 4.6.

In the VGG plot, there is a difference of approximately 10 percentage points between the performance of the networks trained with batch normalisation and LCC and those trained with only one of the weaker baselines, for the case where only 5,000 instances were used during training. As the number of available training instances is increased, the gap between the performance of all methods becomes smaller because each method must rely less on the prior knowledge built into the learning algorithm and more on the evidence provided by the examples in the training set. The performance of networks trained only with batch normalisation consistently falls between that of the baselines and the networks trained with the methods proposed in this chapter. A similar, but

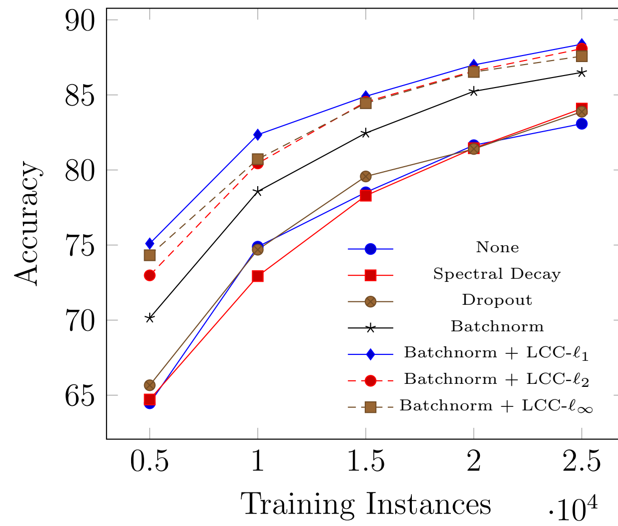


Figure 4.6: Learning curves for VGG-style networks trained with each of the regularisation methods.

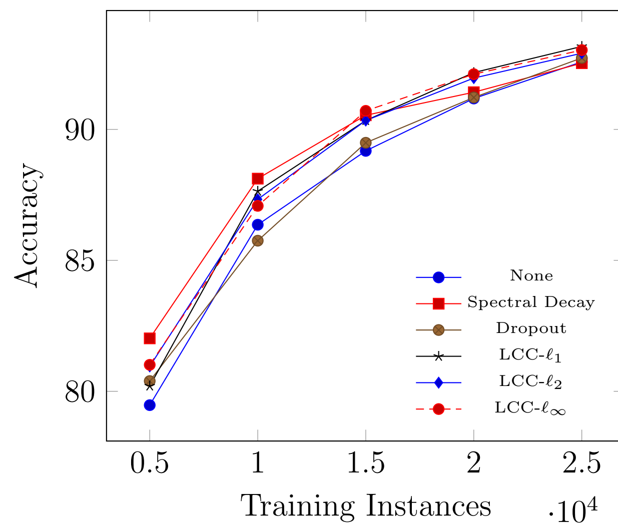


Figure 4.7: Learning curves for wide residual networks trained with each of the regularisation methods. Note that batch normalisation is used when training all of these models.

less exaggerated, trend is visible in the plot corresponding to the wide residual networks in Figure 4.7, with one exception: the spectral decay method exhibits very good performance when only a small amount of training data is available. This agrees with the previous results on the SINS-10 dataset. However, it is interesting to note that as the number of available training examples grows, this advantage is lost and the performance of networks regularised with the spectral decay method tend towards the performance of the unregularised baseline—a trend that is also noticeable in the experiments on other datasets.

4.3 Conclusion

This chapter has presented a simple and effective regularisation technique for deep feed-forward neural networks, shown that it is applicable to a variety of feed-forward neural network architectures and is particularly suited to situations where only a small amount of training data is available. It has also been demonstrated that the technique presented in this chapter can be used in conjunction with both batch normalisation and dropout, which is where one sees the most gain in performance. The investigation into the differences between the three p-norms ($p \in \{1, 2, \infty\}$) considered has provided some useful information about which one might be best-suited to the problem at hand. In particular, the ℓ_∞ norm appears particularly suitable for tabular data, and the ℓ_2 norm showed the most consistently competitive performance when used as a regulariser on natural image datasets. However, given that LCC-ℓ_2 is only approximately constraining the norm, if one wants a guarantee that the Lipschitz constant of the trained network is bounded below some user-specified value, then using the ℓ_1 or ℓ_∞ norm would be more appropriate.

Lastly, recent and concurrent work suggests that the utility of constraining the Lipschitz constant of neural networks is limited not only to improving classification accuracy. There is already evidence that constraining the Lipschitz constant of the discriminator networks in GANs is useful (Arjovsky et al., 2017;

Miyato et al., 2018). Given the shortcomings in previous approaches to constraining Lipschitz constants we have outlined (cf. Section 3.1.2), one might expect improvements training GANs that are k-Lipschitz with respect to the ℓ_1 or ℓ_∞ norms, and approximately 1-Lipschitz with respect to the ℓ_2 norm, by applying the methods presented in this chapter. Exploring how well the technique presented in this chapter works with recurrent neural networks would also be of great interest, but the theoretical basis for the method discussed in Chapter 3 does not apply in the context of recurrent neural networks. Finally, the experiments carried out in this chapter forced all layers of the same type to have the same Lipschitz constant. This is likely an inappropriate assumption in practice, and a more sophisticated hyperparameter tuning mechanism that allows for selecting a different value of λ for each layer could provide a further improvement to performance. It should be noted, however, that devising a means for efficiently allocating modelling capacity on a per-layer basis is an open research problem.

Chapter 5

MaxGain Regularisation of Neural Networks

Regularisation is a crucial component of neural network systems, where the huge number of parameters can lead to extreme overfitting, such as memorising the training set—even in the case where the labels have been randomised (Zhang et al., 2016). In this chapter, a regularisation technique inspired by the LCC method from Chapter 4 is presented. Most work in machine learning that deals with the concept of Lipschitz continuity assumes, often implicitly, that the input domain of the function of interest is $\mathbb{R}^d$—sometimes with the additional assumption that each component in this vector space is bounded in, for example, the range $[-1, 1]$. When working with unstructured data—a task at which neural networks excel—a common assumption is that the data lie near a low-dimensional manifold embedded in a high-dimensional space. This is known as the manifold hypothesis (Cayton, 2005). This chapter explores the idea of constraining the Lipschitz continuity of neural network models when they are viewed in this light: as mappings from the subset of $\mathbb{R}^d$ that contains the low-dimensional manifold to some meaningful vector space, such as the distribution over possible classes. The precise structure of the manifold is unknown to us, which makes constraining a function that operates on this manifold difficult. To circumvent this problem, the concept of *gain*

is introduced—an empirical analogue to the operator norm technique used in Chapters 3 and 4 to compute the Lipschitz constant of a neural network layer.

More specifically, we consider a regularisation scheme that improves the generalisation performance of neural networks by constraining the maximum gain of each layer. This is accomplished using a simple modification to conventional neural network optimisers that applies a stochastic projection function in addition to the usual stochastic estimate of the gradient. The effectiveness of this regularisation algorithm is demonstrated on several popular image classification datasets, and the SINS-10 dataset introduced in Chapter 4. Additionally, it is shown how the technique performs when used in conjunction with other regularisation methods such as dropout (Srivastava et al., 2014) and batch normalisation (Ioffe and Szegedy, 2015). Empirical evidence is also provided that constraining the gain on the training set results in observing lower gain on the test set. Details of how the hyperparameters impact the performance of models trained with this regularisation technique are also provided.

5.1 Regularisation by Constraining Gain

A common assumption in machine learning is that many types of unstructured data, such as images and audio, lie near a low-dimensional manifold embedded in a high dimensional vector space. This is known as the manifold hypothesis. Assuming that the manifold hypothesis holds, a network will only be supplied with elements of some strict subset $\mathcal{X} \subset \mathbb{R}^d$. As a consequence, the training procedure need only ensure that the network is Lipschitz continuous on $\mathcal{X}$ in order to construct a network with a slowly varying decision boundary. In practice, the exact structure of $\mathcal{X}$ is unknown, but a finite sample of instances, $X \subset \mathcal{X}$, is available, and can be used to empirically estimate various characteristics of $\mathcal{X}$.

Lipschitz continuity is not something that can be established empirically,

as it is a property that must be enforced for every possible input—including those have not been seen during training. However, one can find a lower bound for k by sampling pairs of points from the training set and determining the smallest value of k that satisfies the Lipschitz condition, given in Equation 3.1. This solution, while conceptually simple, has a number of finer details that can greatly impact the result. For example, how should pairs be sampled? If they are chosen randomly, then a very large number of pairs will be required to provide a good estimate of k. On the other hand, if a hard-negative mining approach were employed, fewer pairs would be required, but the amount of computation per pair would be greatly increased.

By restricting the analysis to feed-forward neural networks, a simpler and more computationally efficient approach is derived. Recall that the Lipschitz constant of a feed-forward network is given by the product of the Lipschitz constants associated with each activation function—which are usually less than or equal to one and cannot be changed during training—and the operator norms associated with the linear transformations in the learned layers. One can define a quantity, which in this chapter is referred to as gain, by adapting the argument of the supremum in the operator norm,

$$Gain_p(W, \vec{x}) = \frac{\|W\vec{x}\|_p}{\|\vec{x}\|_p}, \tag{5.1}$$

for some input instance $\vec{x}$, and use the maximum gain observed over some set of input vectors from the manifold of interest as an approximation of the operator norm. This empirical estimate of the operator norm of a matrix has several advantages over computing the true operator norm. Firstly, it fulfills the desire to approximately compute the Lipschitz constant of an affine function on $\mathcal{X}$. It is also well behaved, in the sense that $X = \mathcal{X} \implies \sup_{\vec{x}} Gain(W, \vec{x}) = \|W\|_p$. Some more practical advantages include not having to explicitly construct W, but merely requiring a means of computing $W\vec{x}$—a property that is extremely useful when computing the operator norm of a convolutional layer. Also,

because one need not compute a matrix norm directly, it is possible to compute the gain with respect to a p-norm for which it would be NP-hard to compute the induced matrix operator norm.

5.1.1 MaxGain Regularisation

The crux of our regularisation technique is to limit the gain of each layer in a feed-forward neural network. To achieve this, each layer is constrained, in isolation, to have a gain less than or equal to a user specified hyperparameter, γ. Put formally, we wish to solve the following optimisation problem:

$$W_{1..l} = \arg\min_{W_{1..l}} \sum_{\vec{x}_i^1 \in X} \mathcal{L}(\vec{x}_i^1, \vec{y}_i) \tag{5.2}$$

$$s.t. \max_{\vec{x}_i^j} Gain_p(W_j, \vec{x}_i^j) \leq \gamma \qquad \forall j \in \{1 \dots l\}, \tag{5.3}$$

where $\vec{x}_i^j$ represents the input to the jth layer for instance i, $\vec{y}_i$ is a label vector associated with instance i, W_j is the weight matrix for layer j, and $\mathcal{L}(\cdot)$ is some task-specific loss function. Note that if $\|\vec{x}_i^j\|_p$ is zero, the gain for that particular measurement is set to zero rather than leaving it undefined.

The conventional approach to solving Equation 5.2 without the constraint in Equation 5.3 is to use some variant of the stochastic gradient method. For simple constraints, such as requiring W_j to lie in some known convex set, a projection function can be used to enforce the constraint after each parameter update. In the application considered here, applying the projection function after each parameter update would involve propagating the entire training set through the network to measure the maximum gain for each layer. Even for modest sized datasets this is completely infeasible, and it defeats the purpose of using a stochastic optimiser. Instead, it is proposed that one uses a stochastic projection function, where the maximum in Equation 5.3 is taken over the same minibatch used to compute an estimate of the gradient of the loss function. The "stale" activations computed before the weight update are reused in order

to avoid the extra computation required for propagating all of the instances through the network again. The following projection function is used:

$$\pi(W, \hat{\gamma}, \gamma) = \frac{1}{\max(1, \frac{\hat{\gamma}}{\gamma})} W, \tag{5.4}$$

where $\hat{\gamma}$ is the estimate of the maximum gain for layer j. If the MaxGain constraint is not violated, then W will be left untouched. If the constraint is violated, W will be rescaled to fix the violation. In the case where the maximum gain is computed exactly, this function will rescale the weight matrix such that the maximum gain is less than or equal to γ. However, because the maximum gain is only approximated, this constraint may not be perfectly satisfied on the training set.

During training, batch normalisation applies a transformation to the activations of a minibatch using statistics computed using only the instances contained in that minibatch. Thus, the gain measured for a particular instance is dependent on the other instances in the batch. Specifically, the activations, $\vec{x}$, produced by some layer, are standardised:

$$\phi^{bn}(\vec{x}) = \text{diag}(\frac{\vec{\alpha}}{\sqrt{\text{Var}[\vec{x}]}})(\vec{x} - \text{E}[\vec{x}]) + \vec{\beta}, \tag{5.5}$$

where diag$(\cdot)$ denotes a diagonal matrix, $\vec{\alpha}$ and $\vec{\beta}$ are learned parameters, and the Var$[\cdot]$ and E$[\cdot]$ operations are computed over only the instances in the current minibatch. If the estimated mean and variance values are particularly unstable, then the gain values will also be very unstable and the training procedure will converge very slowly—or possibly not at all. Indeed, one can observe empirically that the high dimensionality of hidden layer activation vectors, and their sparse nature when using the ReLU activation function, coupled with a relatively small batch size, tends leads to unstable measurements when using MaxGain in conjunction with batch normalisation. This is remedied by recomputing the batch normalisation output in the projection function using the running averages of the standard deviation estimates that are kept for

performing test-time predictions. By standardising the minibatch activations using these more stable estimates of the activation statistics, considerably more reliable convergence is achieved. Note that the stochastic estimates of the mean and standard deviation of activations are still used for computing the gradient—it is only the projection function that uses the running averages of these values.

Pseudocode for the constrained optimisation algorithm based on stochastic projection is provided in Algorithm 3. The inputs to each layer for each minibatch, $X^{(t)}_{1:l}$, and the results of transforming these by the linear term of the affine transformations, $Z^{(t)}_{1:l}$, are cached during the gradient computation to be reused in the projection function. A single hyperparameter, γ, is used to control the allowed gain of all layers. There is no fundamental reason that a different γ cannot be selected for each layer other than the added difficulty in optimising more hyperparameters. The *update*$(\cdot, \cdot)$ function can be any stochastic optimisation algorithm commonly used with neural networks. This chapters considers both Adam (Kingma and Ba, 2014) and SGD with Nesterov momentum.

Algorithm 3 This algorithm makes use of the stochastic gradient method (or some variation thereof) and a stochastic projection function to approximately solve the constrained optimisation problem outlined in Equations 5.2 and 5.3. The *zip* function converts a tuple of arrays into an array of tuples.

$t \leftarrow 0$
while $W^{(t)}_{1:l}$ not converged **do**
  $t \leftarrow t + 1$
  $(g^{(t)}_{1:l}, X^{(t)}_{1:l}, Z^{(t)}_{1:l}) \leftarrow \nabla_{W_{1:l}} f(W^{(t-1)}_{1:l})$
  $\widehat{W}^{(t)}_{1:l} \leftarrow \mathit{update}(W^{(t-1)}_{1:l}, g^{(t)}_{1:l})$
  for $i = 1$ **to** l **do**
    $\hat{\gamma} \leftarrow 0$
    for $(\vec{x}_j, W^{(t)}_i \vec{x}_j)$ **in** $\mathit{zip}(X^{(t)}_i, Z^{(t)}_i)$ **do**
      $\hat{\gamma} \leftarrow \mathit{max}(\hat{\gamma}, \frac{\|W^{(t)}_i \vec{x}_j\|_p}{\|x_j\|_p})$
    end for
    $W^{(t)}_i \leftarrow \pi(\widehat{W}^{(t)}_i, \hat{\gamma}, \gamma)$
  end for
end while

5.1.2 Compatibility with Dropout

For practical applications, we need to consider how the maxgain approach interacts with dropout when training neural networks. There are two parts to applying dropout regularisation to a network. Firstly, during training, one must stochastically corrupt the activations of some hidden layers, usually by multiplying them with vectors of Bernoulli random variables. Secondly, during test time, the activations are scaled such that the expected magnitude of each activation is the same as what it would have been during training. In the case of standard Bernoulli dropout, this just means multiplying each activation by the probability that it was not corrupted during training. As discussed in Chapter 3, this scaling changes the Lipschitz constant of a network over $\mathbb{R}^d$, and the same argument applies to the Lipschitz constant on $\mathcal{X}$. Because many commonly used activation functions are homogeneous, namely ReLU and its many variants, scaling the output activations is equivalent to scaling the output of the affine transformation. This, in turn, has an identical effect to scaling both the weight matrix and bias vector. Due to the homogeneity of norms, this scaling also directly affects the gain. Therefore, one might expect that one needs to increase γ when using MaxGain in conjunction with dropout. This issue is considered in the experiments below.

5.2 Experiments

The experiments reported in this section aim to demonstrate several aspects of the proposed regularisation method. The primary question is whether the technique for constraining the maximum gain of each learned layer in a network is an effective regularisation method. It is also considered whether constraining the gain on training instances results in observing lower gain on the test. All networks trained with MaxGain regularisation use the same γ parameter for each layer in order to simplify hyperparameter optimisation. While the method can be used in conjunction with any vector norm, in this chapter only the ℓ_2

Table 5.1: Accuracy of a VGG-19 network trained in CIFAR-10 with different regularisation techniques.

Regulariser	Accuracy
None	88.29%
Dropout	89.71%
Batchnorm	90.80%
Batchnorm + Dropout	90.90%
Batchnorm + LCC-ℓ_2	92.68%
MaxGain	90.75%
MaxGain + Dropout	90.95%
MaxGain + Batchnorm	91.76%
MaxGain + Batchnorm + Dropout	91.52%

vector norm is considered.

5.2.1 CIFAR-10

The experiments in this chapter follow the standard CIFAR-10 protocol of using 50,000 images for training and 10,000 images for testing. Additionally, a 10,000 image subset of the training set is used to tune the hyperparameters. VGG-19 networks (Simonyan and Zisserman, 2014) trained using the Adam optimiser (Kingma and Ba, 2014) are used for this dataset. The models are trained for 140 epochs, starting with a learning rate of 10^{-4}, which is decreased to 10^{-5} at epoch 100 and 10^{-6} at epoch 120. Data augmentation in the form of horizontal flips, and padding training images to 40×40 pixels and cropping out a random 32×32 patch is used.

Results demonstrating how MaxGain compares with other common regularisation techniques are given in Table 5.1. Several trends stand out in this table. Firstly, when comparing with each other technique in isolation, the proposed method performs well. It performs noticeably better than dropout and similarly to batch normalisation. When used in conjunction with batch normalisation, the resulting test accuracy improves further. Interestingly, combining the use of dropout with both other regularisation approaches does not seem to have a noticeable cumulative effect.

Table 5.2: Accuracy of a Wide Residual Network with a depth of 16 and a width factor of four trained on CIFAR-100 with different regularisation techniques.

Regulariser	Accuracy
Batchnorm	75.34%
Batchnorm + Dropout	75.72%
Batchnorm + LCC-ℓ_2	76.01%
MaxGain + Batchnorm	75.89%
MaxGain + Batchnorm + Dropout	76.44%

5.2.2 CIFAR-100

Wide Residual Networks (Zagoruyko and Komodakis, 2016) are trained using this dataset, as this allows investigation of how well MaxGain works on networks with residual connections. Batch normalisation is applied to all models trained on this dataset, as convergence is unreliable when training WRNs without batch normalisation. Stochastic gradient descent with Nesterov's momentum is used to train each model for a total of 200 epochs. The initial learning rate is set to 10^{-1} and decreased by a factor of 5 at epochs 60, 120, and 160. The data augmentation used for the CIFAR-10 models is also used when training the CIFAR-100 models.

Results for experiments run on CIFAR-100 are given in Table 5.2. In this case, one can see that MaxGain performs comparably to dropout when both techniques are used in conjunction with batch normalisation. The combination of all three regularisation schemes performs the best.

5.2.3 Street View House Numbers (SVHN)

VGG-style networks are trained on this dataset using the Adam (Kingma and Ba, 2014) optimiser. Likely due to the large size of the dataset, the networks only need to be trained for 17 epochs. An initial learning rate of 10^{-4} is used, and it is reduced it by a factor of 10 for the last two epochs. The number of epochs and the learning rate schedule were determined manually using a validation set of 10,000 examples randomly selected from the more difficult

Table 5.3: Accuracy of a VGG-style network on the SVHN dataset when trained with various regularisation techniques.

Regulariser	Accuracy
None	96.99%
Dropout	97.72%
Batchnorm	96.97%
Batchnorm + Dropout	97.86%
Batchnorm + LCC-ℓ_2	96.94%
MaxGain	97.22%
MaxGain + Dropout	97.89%
MaxGain + Batchnorm	97.31%
MaxGain + Batchnorm + Dropout	97.98%

section of the training set.

Table 5.3 shows how the different models performed on SVHN. An interesting result here is that, in isolation, dropout outperforms both MaxGain and batch normalisation in terms of accuracy improvement over the baseline. This is potentially due to the mismatch between the distributions of the training and testing datasets. Despite the lackluster performance of these methods in isolation, they do still provide a benefit when combined with each other and dropout, which is consistent with the results of the other experiments.

5.2.4 SINS-10

The SINS-10 dataset, which is described in more detail in Chapter 4, consists of 10 folds of 10,000 colour images, all of which are 96×96 pixels. Wide Residual Network with a depth of 16 and a width factor of four are trained on this dataset. No data augmentation is used and each model is trained for 90 epochs using stochastic gradient descent with Nesterov momentum. The learning rate is started at 10^{-1} and decreased by a factor of five at epochs 60 and 80. For each regularisation scheme, a model is trained on each fold of the dataset. Regularisation hyperparameters, such as γ and the dropout rate, are determined on a per-fold basis using a validation set of 1,000 instances drawn from the training set of the fold under consideration.

Results for the different regularisation schemes trained on this dataset are

Table 5.4: Performance of the Wide Residual Network on the Scaled ImageNet Subset dataset using various combinations of regularisation techniques. The figures in this table are the mean accuracy ± the standard error, as measured across the 10 different folds.

Regulariser	Accuracy
Batchnorm	70.13% (±0.27)
Batchnorm + Dropout	74.81% (±0.49)
Batchnorm + LCC-ℓ_2	75.89% (±0.64)
MaxGain + Batchnorm	70.65% (±0.54)
MaxGain + Batchnorm + Dropout	74.80% (±0.51)

given in Table 5.4. The mean accuracy across each of the 10 folds, as well as the standard error, are reported. Paired t-tests were performed to compare Batchnorm to MaxGain + Batchnorm, and also Batchnorm + Dropout versus MaxGain + Batchnorm + Dropout. Neither of the tests resulted in a statistically significant difference ($p = 0.332$ and $p = 0.976$, respectively).

5.2.5 Gain on the Test Set

Due to the stochastic nature of the projection function, the technique used to constrain the gain on the training set is only approximate. Therefore, it is important to verify whether the constraint is fulfilled in practice. Moreover, even if the constraint is satisfied on the training set, that does not necessarily mean it will be satisfied on data not seen during training. To investigate this, plots showing the distribution of gains in each layer in the VGG-19 network trained using MaxGain on the CIFAR-10 dataset are given in Figure 5.1. It can be seen that the gain distributions between the train and test sets are virtually identical, and are never significantly above 2—the value selected for γ when training this network.

In addition to demonstrating that the stochastic projection function does effectively limit the maximum gain on the test set, it is interesting to visualise gain measurements taken from each layer in a network trained without the MaxGain regulariser. This visualisation is given in Figure 5.2. Once again, the distributions of gains measured on the training versus test data are almost

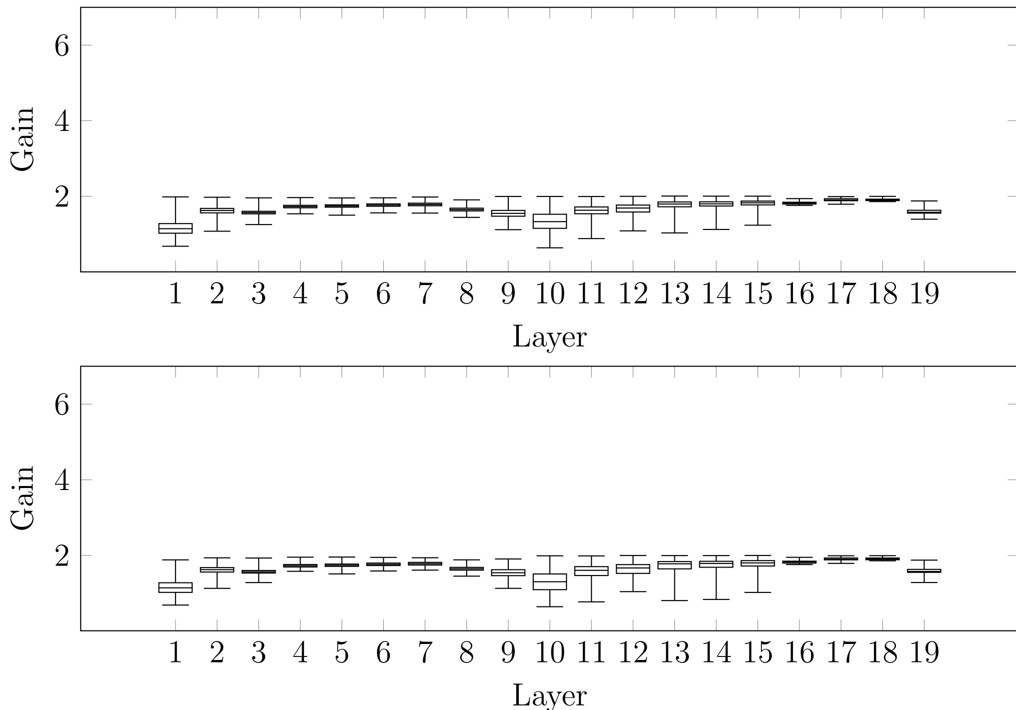


Figure 5.1: Boxplots showing the distributions of gains measured on each layer of the MaxGain-regularised VGG-19 network trained on CIFAR-10. The top plot shows the distributions on the training set, and the bottom plot on the test set.

identical. Comparing the distributions given in Figure 5.2 with those provided in Figure 5.1 show that the MaxGain regulariser has a substantial effect on the activation magnitudes produced by each layer.

If there is no constraint on the magnitude of the weights, then once the network can almost perfectly classify the training data, the optimiser can easily decrease the log loss by making the weights larger. This results in an "exploding activation" effect, similar to the exploding/vanishing gradient phenomenon, which is only curbed when the cost of the small number of instances in the training set that are very confidently classified incorrectly begin to outweigh the increase in confidence on the correct classifications. Because MaxGain constrains the weight sizes of each layer, those that would have had large weights no longer do, and those that would have had small weights will now need larger weights in order to increase the confidence of the model. This results in the far more uniform changes in activation magnitude in Figure 5.1 compared to those in Figure 5.2.

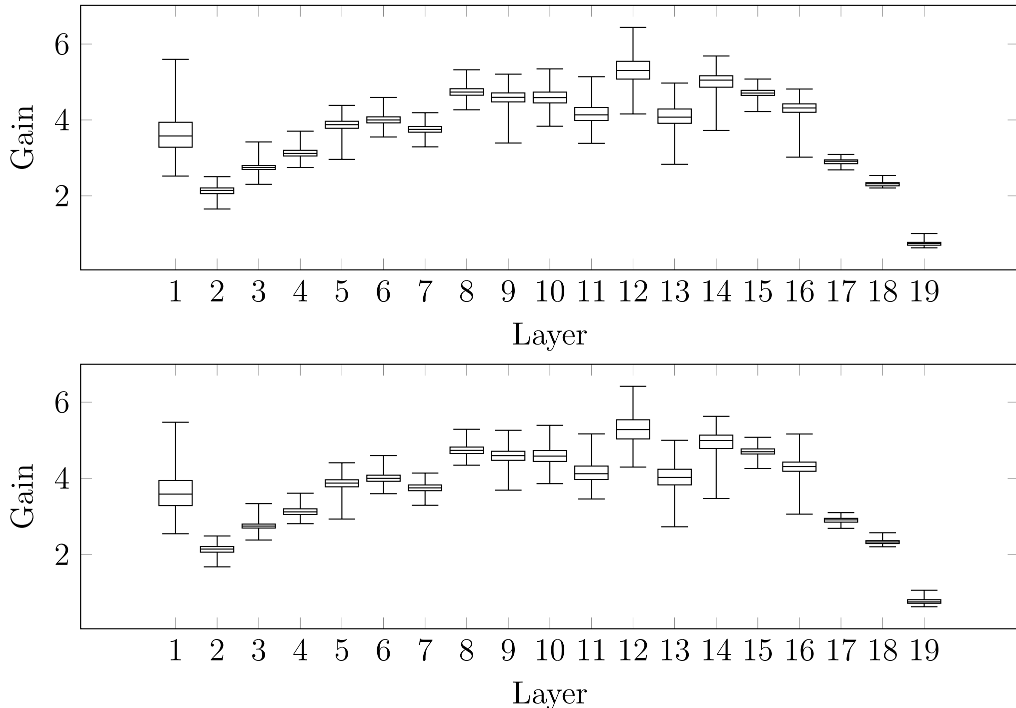


Figure 5.2: Boxplots showing the distributions of gains measured on each layer of the unregularised VGG-19 network trained on CIFAR-10. The top plot shows the distributions on the training set, and the bottom plot on the test set.

5.2.6 Sensitivity to γ

The single hyperparameter, γ, that is used to control the capacity of MaxGain-regularised networks should behave similarly to the λ hyperparameter used in Chapter 4, which is used to precisely bound the Lipschitz constant. In particular, when γ is set to a small value the model should underfit, and when it is set to a large value one should observe overfitting. This is explored empirically in the context of the VGG-style network trained on SVHN. Figure 5.3 shows how the performance on the training and test sets of SVHN varies as γ is changed. This plot shows that γ behaves somewhat similarly to the previously mentioned λ hyperparameter. Specifically, for very low values of γ, the network exhibits low accuracy and high loss for both the train and test splits of the dataset. As the value of γ is increased, the training accuracy goes towards 100% and the loss goes towards zero. The test accuracy peaks and then plateaus, but the loss on the test set continues to increase, indicating that the

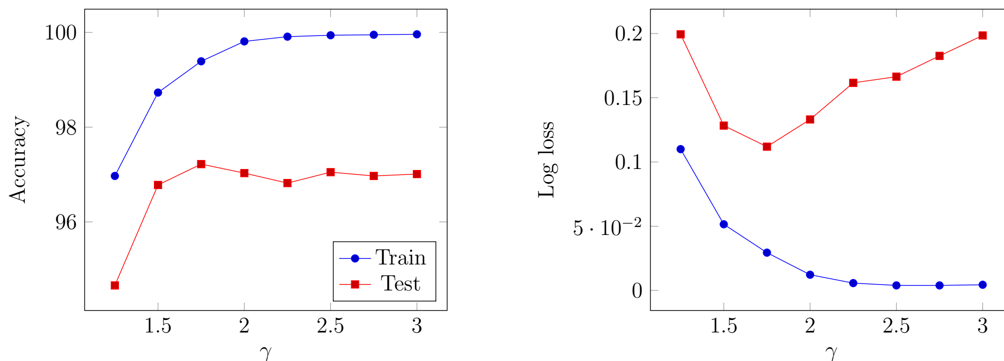


Figure 5.3: Accuracy (left) and log loss (right) of the VGG-style model on both the train and test splits of the SVHN dataset as the γ hyperparameter is varied. The legend is shared between both plots.

network is more confidently misclassifying instances rather than misclassifying more instances. However, the behaviour of γ is quite different to λ is the sense that the peak is smoother, and γ is much more intuitive to train: it was set to two for all experiments in this chapter, except on SVHN, where it was set to 1.75.

5.3 Discussion

This chapter introduced MaxGain, a method for regularising neural networks by constraining how the magnitudes of activation vectors can vary across layers. It was shown how this method can be seen as an approximation to constraining the Lipschitz constant of a network, with the advantage of being usable for any vector norm. The technique is conceptually simple and easy to implement efficiently, thus making it a very practical approach to controlling the capacity of neural networks. It has been shown that MaxGain performs competitively with other common regularisation schemes, such as batch normalisation and dropout, when compared in isolation. It was also demonstrated that when these techniques are combined together, further performance gains can be achieved. Results on classification tasks have shown that MaxGain often exhibits performance competitive with LCC-ℓ_2, however γ requires much less optimisation effort than λ in order to achieve acceptable performance gains.

Chapter 6

A Problem Transformation Approach to Embedding

In the previous chapters, the primary motivation for the presented techniques has been to improve results on multiclass classification problems. This chapter considers the problem of multi-label classification, where additional structure in the label space is present. This can be advantageous, because correlations between labels provide useful additional information about the relationships between different groups of instances. However, it can also be a disadvantage because many common constructs in conventional classification problems, such as the notion of accuracy, do not have an obvious equivalent in the multi-label setting. The methods presented so far have made use of the standard framework of applying gradient-based optimisation methods to find good settings for the parameters in end-to-end differentiable models. This chapter introduces a method for decoupling the optimisation of representations from the training of embedding models, thus enabling the embedding functions to come from hypothesis spaces that cannot be searched using gradient-based optimisers. This approach allows one to use models such as decision trees to transform data into a more amenable representation. This is accomplished using a two step process. The first step is conceptually similar to multidimensional scaling, and finds an idealised target embedding for each instance in the training set. That

is, we compute a good vectorial representation for each training instance without considering how we might actually construct a general model that can map from arbitrary points in the original feature space to this new representation. The process is similar to multidimensional scaling in the sense that we use a pairwise distance matrix as the source of supervision, but it differs in that the pairwise distances are not simply the Euclidean distances between feature vectors: they are a measure of semantic similarity between label sets. Once these vectors are obtained, the second stage of the pipeline is to construct an embedding function by training a multi-target regression model to map from the original feature space to this new target space.

There is a potential disadvantage to learning embedding models in this way: the multi-target regression learner may not be able to find an accurate mapping from the original feature space to the target vectors learned during the first phase. This is a problem that is not encountered in the end-to-end optimisation paradigm because the learned representations are determined to a significant extent by the inductive bias of the class of embedding models selected by the user. For an example of how this might be a problem, consider how one would define a semantic similarity metric for multiclass classification. The obvious choice is to say two instances are related if they belong to the same class and unrelated otherwise. An end-to-end learning system with the appropriate inductive bias would organise the learned target vectors in such a way that instances of the same class are clustered together, and the clusters associated with semantically related classes are adjacent. In contrast, the two stage approach described in this chapter would be unable to group semantically related clusters together because the inductive bias of the multi-target regression model, which could potentially effect such a grouping, has no opportunity to influence. However, if the measure of semantic similarity provides more information about how the target vectors should be arranged, then the proposed problem transformation approach is a suitable method for extracting useable representations from data. We hypothesise that the structure in the

label space in multi-label classification tasks often contains the information required to learn a globally coherent latent space using our technique.

We approach the task of learning an embedding function using pairwise distance supervision as a metric learning problem, as this leads to a well defined set of rules for eventually making predictions using the resulting representations: the k-nearest neighbours (k-NN) approach can be applied. Multiclass classification is one of the most ubiquitous tasks found in the field of machine learning, so it should come as no surprise that the majority of methods for learning distance metrics are designed to be applied to multiclass data. These techniques generally involve optimising a loss function that considers pairwise equality constraints between similar instances: two instances are considered similar if they both belong to the same class. However, in the case of multi-label classification, each instance can be assigned multiple labels, and it is unclear how to define equality. Should two instances be considered equal only if they have exactly the same set of labels? Or should equality be indicated by non-empty overlap between the label sets of the two instances? Should a threshold be selected for the number of shared labels required for two instances to be considered equal? To sidestep this issue, we investigate a method for learning distance metrics that avoids labelling pairs as equal or not equal, and instead assigns a real valued similarity score.

Perhaps the most important aspect of the proposed two-step approach is that it enables representation learning using decision trees by applying multivariate predictions obtained from decision trees in the second stage of the pipeline. Neural networks tend to exhibit inferior performance on tabular datasets compared to decision tree methods (see Section 7.4) so alternative approaches for representation learning on such data are desirable. For completeness, we also show how a linear embedding function can be trained with the objective that is used to find the target vectors in the first-step of the pipeline. This linear method, which couples the two stages by using a common loss function, is also more closely related to the extensive prior literature

on metric learning for the multiclass classification setting than the problem transformation approach.

This chapter first summarises related work in Section 6.1 and then describes the method for learning linear multi-label distance metrics in Section 6.2, which we refer to as Linear Jaccard Embedding (LJE). This is followed by a generalisation to nonlinear metrics using problem transformation in Section 6.3, resulting in the Nonlinear Jaccard Embedding (NJE) technique. Section 6.4 demonstrates that the linear metrics work well on high-dimensional data and classification accuracy is substantially improved in general when using the nonlinear method. Section 6.5 summarises the contributions of this chapter and speculates on future directions.

6.1 Background

Many distance metric learning algorithms are based on the Mahalanobis distance,

$$D(\vec{x}_i, \vec{x}_j) = \sqrt{(\vec{x}_i - \vec{x}_j)^T \mathbf{M} (\vec{x}_i - \vec{x}_j)}, \tag{6.1}$$

where $\vec{x}_i$ and $\vec{x}_j$ are feature vectors and $\mathbf{M}$ is the matrix of model parameters found during the training process.[1]

Prior work addressing metric learning for improving the performance of k-NN classifiers has almost exclusively focused on the multiclass classification setting. Large Margin Nearest Neighbours (Weinberger et al., 2005) learns a metric by optimising a loss function that considers triples of instances: a seed instance, an instance similar to the seed, and an instance dissimilar to the seed. Its aims to make the distance between the similar instances smaller than the distance between the dissimilar instances. Another technique is Information Theoretic Metric Learning (Davis et al., 2007). The objective for this approach aims to minimise—subject to instance equality constraints—the Kullback-Leibler divergence between a prior Gaussian distribution and the

[1] In the original formulation of Mahalanobis distance, $\mathbf{M}$ is the inverse covariance matrix of the training data.

Gaussian distribution with a covariance matrix parameterised by the inverse of $\mathbf{M}$. The linear method discussed first in this chapter is similar to these approaches in the sense that it applies a metric with the same functional form, but instead of binary equality constraints it uses real-valued ground truth measures of similarity derived from the label sets of the training instances. The most similar approach is Large Margin k-NN (Liu and Tsang, 2015). The work of Liu and Tsang (2015) appears to be the only paper to directly address the problem of metric learning for multi-label data, but several other works consider scenarios with additional constraints (Jin et al., 2009; Guillaumin et al., 2010).

Metric learning alone is not capable of provding predictions, but it can be combined with a nearest neighbour-based classification rule to provide a complete classification system. The conventional k-NN classification scheme is inherently a multiclass method, and not immediately capable of performing multi-label classification. A popular adaptation of k-NN to the multi-label problem is the Multi-label k-Nearest Neighbours (MLkNN) algorithm (Zhang and Zhou, 2007). This method finds the k nearest neighbours of a test instance and uses a maximum likelihood technique to predict the labels.

An alternative to adapting the classification algorithm, as in the case of MLkNN, is to reformulate the problem such that an ensemble of binary or multiclass classifiers can be used to perform multi-label classification; so-called problem transformation methods do just that. The most basic method is the Binary Relevance (BR) scheme, which simply trains a binary classifier for each label in the dataset, ignoring any information that can be gleaned from considering label correlations. Another popular method that fits into this framework is the Ensemble of Classifier Chains (ECC) method proposed by Read et al. (2011). Each node in the classifier chain predicts the presence of a single label using a binary classifier, and each prediction is then appended to the feature vector before continuing along the chain to the next binary classifier. The order that the labels are predicted is generated randomly during

the process of training the classifier chain. Creating ensembles of these chains via bagging results in a classifier that is able to take advantage of correlations between labels when making predictions.

6.2 Learning Linear Metrics

We start by defining a loss function that can be applied to Mahalanobis metrics—metrics of the form given in Equation 6.1. To do this, we must first decide what this loss function should accomplish. As discussed above, there is no single appropriate definition for accuracy when performing multi-label classification. However, evaluation measures such as the Jaccard index,

$$J(Y_i, Y_j) = \frac{|Y_i \cap Y_j|}{|Y_i \cup Y_j|}, \tag{6.2}$$

can be used to quantify the similarity of two label sets. Interestingly, one can transform the Jaccard index into a distance metric, called the Jaccard distance, in a trivial manner:

$$D_J(Y_i, Y_j) = 1 - J(Y_i, Y_j). \tag{6.3}$$

We propose the use of the Jaccard distance between the label sets of training instances to determine how similar two instances are when learning a distance metric. For this reason, we refer to this linear metric learning approach as Linear Jaccard Embedding (LJE). This provides a fine-grained measure of similarity between the examples in the training data. We use this information to train a linear distance metric that can estimate the Jaccard distance between two unknown label sets by considering only the features associated with the label sets.

More formally, suppose we have a training set, $\mathcal{X}$, with elements of the form $(\vec{x}_i, Y_i)$, where $\vec{x}_i \in \mathbb{R}^d$ is the d-dimensional column vector of features for instance i, and Y_i is the label set. We define a set, $\mathcal{Z} = \mathcal{X} \times \mathcal{X}$, that contains

all possible pairings of instances in $\mathcal{X}$. The parameters for our Mahalanobis distance metric are learned by solving an optimisation problem,

$$\mathbf{M}^* = \arg\min_{\mathbf{M}} \frac{1}{|\mathcal{Z}|} \sum_{((\vec{x}_i, Y_i),(\vec{x}_j, Y_j)) \in \mathcal{Z}} L(\vec{x}_i, \vec{x}_j, Y_i, Y_j) \tag{6.4}$$

$$L(\vec{x}_i, \vec{x}_j, Y_i, Y_j) = \begin{cases} (D_J(Y_i, Y_j) - D(\vec{x}_i, \vec{x}_j)^2)^2, & D_J(Y_i, Y_j) < 1 \\ -\min(1, D(\vec{x}_i, \vec{x}_j)^2), & \text{otherwise} \end{cases} \tag{6.5}$$

where $D(\cdot, \cdot)$ is a Mahalanobis metric parameterised by the positive-semidefinite matrix $\mathbf{M}$, and $D_J(\cdot, \cdot)$ is the Jaccard distance. This function learns a clamped Mahalanobis metric that is capable of estimating the Jaccard distance between the unknown label sets of two new data points. The min function prevents the metric from being disproportionally rewarded when large values are predicted for completely dissimilar instances.

With the formulation given in Equation 6.4, the parameter matrix, $\mathbb{M}$, must be constrained to be positive semi-definite. To avoid solving a constrained optimisation problem, we reparameterise the Mahalanobis distance metric as $\mathbf{M} = \mathbf{G}^T\mathbf{G}$. In this formulation $\mathbf{G}$ is a matrix with the same number of rows as $\mathbf{M}$, but the number of columns is specified by a new hyper-parameter, t. That is, $\mathbf{M}$ is $d \times d$ and $\mathbf{G}$ is $t \times d$. This guarantees that $\mathbf{M}$ is positive-semidefinite, and thus $D(\cdot, \cdot)$ behaves as a proper distance metric. We can now equivalently express the canonical formula for Mahalanobis metrics as

$$D(\vec{x}_i, \vec{x}_j) = \sqrt{(\mathbf{G} \cdot (\vec{x}_i - \vec{x}_j))^T (\mathbf{G} \cdot (\vec{x}_i - \vec{x}_j))}. \tag{6.6}$$

One can think of this alternative formulation as embedding the instances into a t dimensional vector space and then applying Euclidean distance to this new representation. An attractive property of explicitly embedding the data is that it can be stored in a data structure, such as a k-d tree, capable of performing efficient nearest neighbour searches if the dimensionality of the embedding is sufficiently small. This greatly reduces the computational cost of performing

neighbourhood queries on new instances during k-NN classification. Based on this reparametrisation, we solve for the optimal value of $\mathbf{G}$ instead of $\mathbf{M}$.

Because $\mathcal{Z}$ is of size $|\mathcal{X}|^2$, moderate growth in the training set size will lead to a great increase of $|\mathcal{Z}|$, causing exact optimisation methods, such as batch gradient descent or Newton-based methods, to become very slow. The conditional expression and $\min(\cdot, \cdot)$ function also present problems, rendering the objective nonsmooth and also resulting in a very large plateau on the error surface—something exact optimisation methods tend to struggle with. Hence, we apply approximate optimisation using the Adam optimisation algorithm (Kingma and Ba, 2014), a variant of stochastic gradient descent (SGD). An additional advantage of using Adam is that it is designed for training deep neural networks, where local minima are abundant. Our objective is not convex with respect to $\mathbf{G}$, so our choice of optimiser has a great impact on the final accuracy of the learned metrics.

Adam requires gradient information to optimise functions, but our objective is nonsmooth and we must settle for having a means of computing subgradients. This is accomplished by rewriting our reparameterised objective function in a form more amenable to differentiation. Expanding the min in the loss function yields

$$L(\vec{x}_i, \vec{x}_j, Y_i, Y_j) = \begin{cases} (D_J(Y_i, Y_j) - D(\vec{x}_i, \vec{x}_j)^2)^2, & D_J(Y_i, Y_j) < 1 \\ -D(\vec{x}_i, \vec{x}_j)^2, & D_J(Y_i, Y_j) = 1 \text{ and } D(\vec{x}_i, \vec{x}_j)^2 < 1 \\ -1, & \text{otherwise,} \end{cases} \tag{6.7}$$

and differentiating with respect to the elements of $\mathbf{G}$ results in

$$\nabla_{\mathbf{G}} L = 2\mathbf{G}(\vec{x}_i - \vec{x}_j)(\vec{x}_i - \vec{x}_j)^T \times \begin{cases} 2(D(\vec{x}_i, \vec{x}_j)^2 - D_J(Y_i, Y_j)), & D_J(Y_i, Y_j) < 1 \\ -1, & D_J(Y_i, Y_j) = 1 \text{ and } D(\vec{x}_i, \vec{x}_j)^2 < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{6.8}$$

Equation 6.8 is used by Adam to compute the required gradients. Algorithm 4 describes the entire metric learning procedure, including parameter initialisation, optimisation, and training pair generation. We use the parameters suggested by Kingma and Ba (2014) for Adam: $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$. The convergence tolerance, e, is the number of iterations through the dataset that will be tolerated without observing an improvement in the loss function. We set this value to 5 in all of our experiments.

Algorithm 4 Optimisation procedure for finding $\mathbf{G}$.

```
epochs ← 0
bestloss ← ∞
loss ← 0
for all i, j ∈ {1, ..., T} × {1, ..., D} do
    G_ij ← RandomSample(N(0, 1/(D+T)))
end for
while epochs < e do
    for all (x_i, Y_i) ∈ X do
        (x_j, Y_j) ← RandomElement(X)
        loss ← loss + L(x_i, x_j, Y_i, Y_j)
        G ← Adam(∇_G L, G, x_i, x_j, Y_i, Y_j)
    end for
    if loss < bestloss then
        bestloss ← loss
        epochs ← 0
    else
        epochs ← epochs + 1
    end if
end while
```

6.3 Extension to Nonlinear Models

As mentioned in Section 6.2, the formulation of Mahalanobis distance given in Equation 6.6 can be thought of as a transformation of two instances into a t-dimensional vector space, followed by a comparison between the two embeddings using Euclidean distance. In this section, we consider how the linear transformation can be swapped out for a nonlinear embedding model. The resulting technique is referred to as Nonlinear Jaccard Embedding (NJE). The method is composed of two phases: learning the target vectors that provide a suitable representation for each instance, and then learning an embedding function that can map arbitrary points from the feature space to the target space. The target vectors can be found by minimising any loss function for metric learning that involves embedding instances into a vector space with a predefined metric (such as Euclidean distance) that is a useful indicator of similarity.

Note that rather than actually learning a function that can perform the embedding, the proposed approach simply computes what the ideal embedding for each instance is for the supplied loss function. This method also requires the hyperparameter, t, that determines the size of these embeddings. The first phase can be concisely described as

$$E^* = \arg\min_{E} \frac{1}{|\mathcal{Z}|} \sum_{((\vec{x}_i,Y_i),(\vec{x}_j,Y_j))\in\mathcal{Z}} L(\vec{x}_i, \vec{x}_j, Y_i, Y_j) \tag{6.9}$$

$$L(\vec{x}_i, \vec{x}_j, Y_i, Y_j) = \begin{cases} (D_J(Y_i, Y_j) - D(\vec{x}_i, \vec{x}_j)^2)^2, & D_J(Y_i, Y_j) < 1 \\ -\min(1, D(\vec{x}_i, \vec{x}_j)^2), & \text{otherwise} \end{cases} \tag{6.10}$$

$$D(\vec{x}_i, \vec{x}_j) = \|\vec{e}_i - \vec{e}_j\|_2, \tag{6.11}$$

where $\vec{e}_i, \vec{e}_j \in E$ are the t-dimensional target vectors associated with $\vec{x}_i, \vec{x}_j \in \mathcal{X}$. We use the Adam optimiser to find the set, E^*, of locally optimal target vectors. The linear approach discussed above required optimising G, but now we must take the derivative with respect to the target vectors we are

attempting to learn,

$$(\frac{\partial L}{\partial \vec{e}_i}, \frac{\partial L}{\partial \vec{e}_j}) = 2(\vec{e}_i - \vec{e}_j, \vec{e}_j - \vec{e}_i) \times \begin{cases} 2(D(\vec{x}_i, \vec{x}_j)^2 - D_J(Y_i, Y_j)), & D_J(Y_i, Y_j) < 1 \\ -1, & D_J(Y_i, Y_j) = 1 \text{ and } D(\vec{x}_i, \vec{x}_j)^2 < 1 \\ 0, & \text{otherwise.} \end{cases} \tag{6.12}$$

The optimisation is performed by adapting Algorithm 4 to use the appropriate derivatives and proceeding to sample pairs of instances for a fixed number of iterations. 10,000 epochs were sufficient for all problems considered in the experiments. Experiments showed that the simple stopping criterion used for the linear metric learning—monitoring the value of the loss function—is not a good heuristic for determining when the optimisation is complete when training the target vectors for the problem transformation approach.

Once the target vectors are obtained, the next stage of the nonlinear two-step modeling approach requires a model that can perform multi-target regression—a problem similar to multi-label classification but with a vector of real valued response variables rather than binary labels. We investigate how well the framework performs when the multi-target regression model is a set of random forests. That is, a separate random forest is trained to make predictions for each component in the target vector space. This can be seen as an analogue to the binary relevance method for regression. Figure 6.1 outlines the proposed two-stage procedure for constructing nonlinear distance metrics in the context of k-NN classification.

Once a metric has been learned, it can be applied to novel instances in a similar manner to other embedding-based metrics. Firstly, two instances are embedded into the t-dimensional vector space by the random forests. Secondly, these embeddings are compared using squared Euclidean distance, resulting in an estimate of the Jaccard distance between the two unknown label sets.

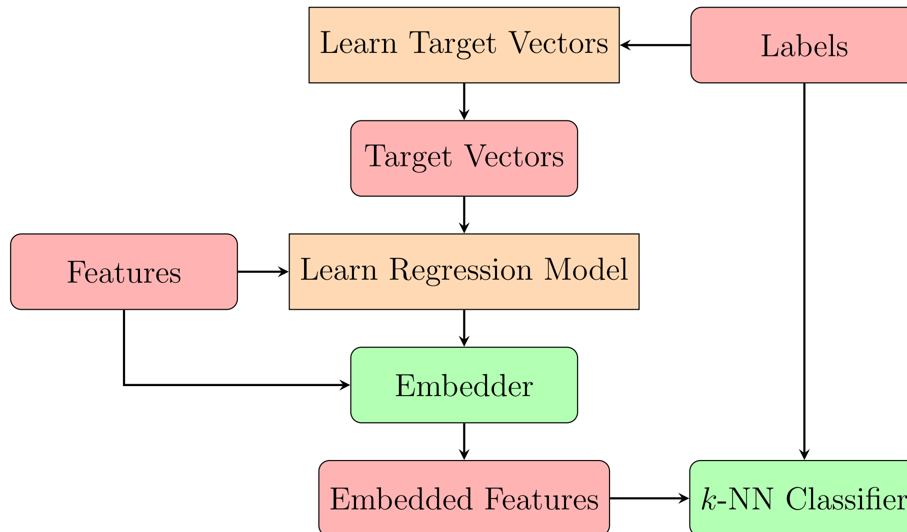


Figure 6.1: This diagram provides an overview of how nonlinear metrics are constructed using the proposed method. The first step in the process is to learn the target vectors for which squared Euclidean distance is an accurate estimate of the Jaccard distance over the original label sets. Once the target vectors have been learned, they, along with the original features, are used to create a multi-target regression model. We apply a separate random forest for each target vector component. Once the multi-target regression model has been learned, it can be used to embed the features into the same space as the target vectors. Finally, the embeddeds and the original labels can be used to construct a k-NN classifier.

6.4 Experiments

We evaluate the above approaches using two problems for which embedding-based metric learning is appropriate: the improvement of k-NN type classifiers, and visualising datasets. We take advantage of several freely available datasets that are summarised in Table 6.1. The algorithms were implemented in the MEKA framework (Read et al., 2016) and the source code has been released publicly.[2]

6.4.1 Classification

The proposed approaches are designed to improve the performance of multi-label k-NN classifiers. Hence, we compare our approaches to plain Euclidean distance by evaluating them in conjunction with several standard k-NN multi-

[2] `http://github.com/henrygouk/meka-metric-learning`

Table 6.1: A summary of the datasets that we use in our experiments. The label cardinality is the average number of labels assigned to each example.

Dataset	Instances	Features	Labels	Cardinality
scene	2,407	294	6	1.074
yeast	2,417	103	14	4.237
enron	1,702	1,001	53	3.378
emotions	592	72	6	1.869
genbase	662	1,186	27	3.392
medical	978	1,449	45	1.245

label classification schemes. In particular, we consider the binary relevance method and ensembles of classifier chains (both with k-NN base learners) as implemented by the MEKA framework. We also compare against the MLkNN scheme implemented in the MULAN framework (Tsoumakas et al., 2011). We set $k = 10$ for all models. For all the models taking advantage of the nonlinear metric learning procedure, we set $t = 16$. In the case of the linear models, we set $t = 32$. All ECC models use an ensemble size of 50.

Because there is no single evaluation metric that is suitable for multi-label classification, we report results using the Jaccard index, the log loss averaged over each label, the Hamming loss, and the F1 score averaged over each example.

We first investigate the performance of LJE compared to other common approaches for performing k-NN classification on multi-label data. The results for these experiments are summarised in Table 6.2. On average, performance is competitive with two of the three baselines. MLkNN is the clear winner according to all evaluation metrics, but the performance of all the linear metric learning approaches is significantly better on enron and medical, the high-dimensional natural language datasets.

Results for the k-NN models taking advantage of NJE can be found in Table 6.3. In this experiment, we also evaluate the MANIAC method (Wicker et al., 2016), which is similar to our technique in the sense that a nonlinear transformation of the labels is used in an attempt to improve predictive accu-

Table 6.2: Multi-label classification performance. LJE indicates the linear Jaccard Embedding method.

(a) F1 score (macro averaged over examples, higher is better)

Datasets	BR	MLkNN	ECC	LJE-BR	LJE-MLkNN	LJE-ECC
yeast	**0.660 (6)**	0.652 (4)	0.656 (5)	0.641 (2)	0.640 (1)	0.649 (3)
enron	0.364 (2)	0.456 (3)	0.339 (1)	0.507 (4)	**0.533 (6)**	0.518 (5)
medical	0.610 (2)	0.662 (3)	0.583 (1)	0.686 (4)	0.689 (5)	**0.696 (6)**
emotions	0.648 (5)	0.638 (4)	**0.657 (6)**	0.611 (2)	0.608 (1)	0.624 (3)
genbase	0.945 (5)	**0.961 (6)**	0.914 (2)	0.919 (3)	0.890 (1)	0.941 (4)
scene	0.642 (3)	**0.663 (6)**	0.634 (1)	0.652 (5)	0.647 (4)	0.641 (2)
Avg. Rank	3.833	**4.333**	2.667	3.333	3.000	3.833

(b) Hamming loss (lower is better)

Datasets	BR	MLkNN	ECC	LJE-BR	LJE-MLkNN	LJE-ECC
yeast	0.217 (5)	0.202 (2)	**0.201 (1)**	0.229 (6)	0.210 (4)	0.206 (3)
enron	0.079 (6)	0.067 (3)	0.071 (5)	0.068 (4)	**0.059 (1)**	0.063 (2)
medical	0.020 (5)	0.018 (3)	0.022 (6)	0.018 (4)	**0.017 (1)**	0.017 (2)
emotions	0.209 (3)	0.207 (2)	**0.200 (1)**	0.233 (6)	0.226 (5)	0.217 (4)
genbase	0.007 (2)	**0.006 (1)**	0.011 (4)	0.043 (6)	0.014 (5)	0.008 (3)
scene	0.136 (6)	**0.119 (1)**	0.131 (4)	0.134 (5)	0.126 (2)	0.131 (3)
Avg. Rank	4.500	**2.000**	3.500	5.167	3.000	2.833

(c) Jaccard index (higher is better)

Datasets	BR	MLkNN	ECC	LJE-BR	LJE-MLkNN	LJE-ECC
yeast	0.551 (5)	0.546 (4)	**0.556 (6)**	0.529 (1)	0.531 (2)	0.546 (3)
enron	0.260 (2)	0.324 (3)	0.260 (1)	0.375 (4)	**0.405 (6)**	0.399 (5)
medical	0.562 (2)	0.616 (3)	0.536 (1)	0.634 (4)	0.645 (5)	**0.653 (6)**
emotions	0.559 (5)	0.551 (4)	**0.575 (6)**	0.523 (2)	0.518 (1)	0.542 (3)
genbase	0.930 (5)	**0.946 (6)**	0.894 (2)	0.902 (3)	0.841 (1)	0.922 (4)
scene	0.606 (1)	**0.631 (6)**	0.611 (2)	0.616 (3)	0.620 (5)	0.619 (4)
Avg. Rank	3.333	**4.333**	3.000	2.833	3.333	4.167

(d) Log loss (averaged over labels, lower is better)

Datasets	BR	MLkNN	ECC	LJE-BR	LJE-MLkNN	LJE-ECC
yeast	**0.422 (1)**	0.431 (2)	0.435 (3)	0.443 (4)	0.446 (5)	0.458 (6)
enron	0.179 (5)	0.157 (3)	0.226 (6)	0.151 (2)	**0.149 (1)**	0.160 (4)
medical	0.049 (5)	0.046 (3)	0.057 (6)	0.045 (2)	**0.044 (1)**	0.048 (4)
emotions	0.399 (3)	0.416 (4)	**0.372 (1)**	0.428 (5)	0.444 (6)	0.391 (2)
genbase	0.015 (3)	0.014 (2)	0.023 (6)	**0.013 (1)**	0.017 (4)	0.018 (5)
scene	0.245 (6)	0.232 (3)	**0.227 (1)**	0.239 (4)	0.240 (5)	0.228 (2)
Avg. Rank	3.833	**2.833**	3.833	3.000	3.667	3.833

racy. When considering only the average rank of each method for each metric, we can see that all the models taking advantage of the Jaccard embedding have superior performance to all models that simply use Euclidean distance. We observe that the performance of MANIAC is positively correlated with the number of labels in the dataset, which is congruent with conclusions made by Wicker et al. (2016). In contrast, our nonlinear method performs well irrespective of the number of labels in the dataset. Investigating further, the BR models that use the Jaccard embedding perform the best on three of the four metrics, and on the other metric they are ranked second best. We see this as strong evidence that the nonlinear Jaccard embedding method coupled with the BR problem transformation is a good choice for k-NN classification on multi-label data.

6.4.2 Target Vector Size

The size of the target vectors, controlled by t, has a substantial effect on the final performance of a k-NN based multi-label classification scheme. The dimensionality of the vector space the features are embedded into must be large enough to accurately capture all the interactions between the different labels. However, setting t to be too large could make the optimisation problem unnecessarily complex and the process of training the multi-target regression model more time consuming. To shed some light on the effect of the dimensionality, we investigate how t impacts the performance of k-NN trained under the BR scheme when using our nonlinear metric learning method. The results of these experiments are summarised in Figure 6.2. It can be seen that the performance of each algorithm on each dataset converges towards a constant value under all metrics as the dimensionality grows. In most cases the best performing models are obtained when $t \geq 16$, but for genbase this is not the case. A possible cause for this is that the embedder is overfitting due to the small number of instances and large number of attributes. Another potential concern is that because the loss function has multiple minima, and we are only performing

Table 6.3: Multi-label classification performance. NJE indicates the nonlinear Jaccard Embedding method.

(a) F1 score (macro averaged over examples, higher is better)

Datasets	MANIAC	BR	MLkNN	ECC	NJE-BR	NJE-MLkNN	NJE-ECC
yeast	0.567 (1)	0.660 (4)	0.652 (2)	0.656 (3)	**0.667 (7)**	0.663 (5)	0.664 (6)
enron	0.441 (3)	0.364 (2)	0.456 (4)	0.339 (1)	0.534 (5)	**0.540 (7)**	0.535 (6)
medical	**0.745 (7)**	0.610 (2)	0.662 (3)	0.583 (1)	0.736 (6)	0.693 (4)	0.728 (5)
emotions	0.472 (1)	0.648 (3)	0.638 (2)	0.657 (5)	**0.673 (7)**	0.668 (6)	0.656 (4)
genbase	**0.985 (7)**	0.945 (4)	0.961 (6)	0.914 (2)	0.914 (3)	0.849 (1)	0.960 (5)
scene	0.435 (1)	0.642 (3)	0.663 (4)	0.634 (2)	**0.727 (7)**	0.689 (5)	0.721 (6)
Avg. Rank	3.333	3.000	3.500	2.333	**5.833**	4.667	5.333

(b) Hamming loss (lower is better)

Datasets	MANIAC	BR	MLkNN	ECC	NJE-BR	NJE-MLkNN	NJE-ECC
yeast	0.216 (6)	0.217 (7)	0.202 (5)	0.201 (4)	0.196 (2)	0.197 (3)	**0.195 (1)**
enron	**0.051 (1)**	0.079 (7)	0.067 (5)	0.071 (6)	0.060 (4)	0.058 (2)	0.059 (3)
medical	**0.013 (1)**	0.020 (6)	0.018 (5)	0.022 (7)	0.015 (2)	0.017 (4)	0.016 (3)
emotions	0.334 (7)	0.209 (6)	0.207 (5)	0.200 (4)	0.188 (2)	**0.186 (1)**	0.191 (3)
genbase	**0.002 (1)**	0.007 (4)	0.006 (2)	0.011 (5)	0.065 (7)	0.019 (6)	0.006 (3)
scene	0.255 (7)	0.136 (6)	0.119 (4)	0.131 (5)	**0.102 (1)**	0.113 (3)	0.102 (2)
Avg. Rank	3.833	6.000	4.333	5.167	3.000	3.167	**2.500**

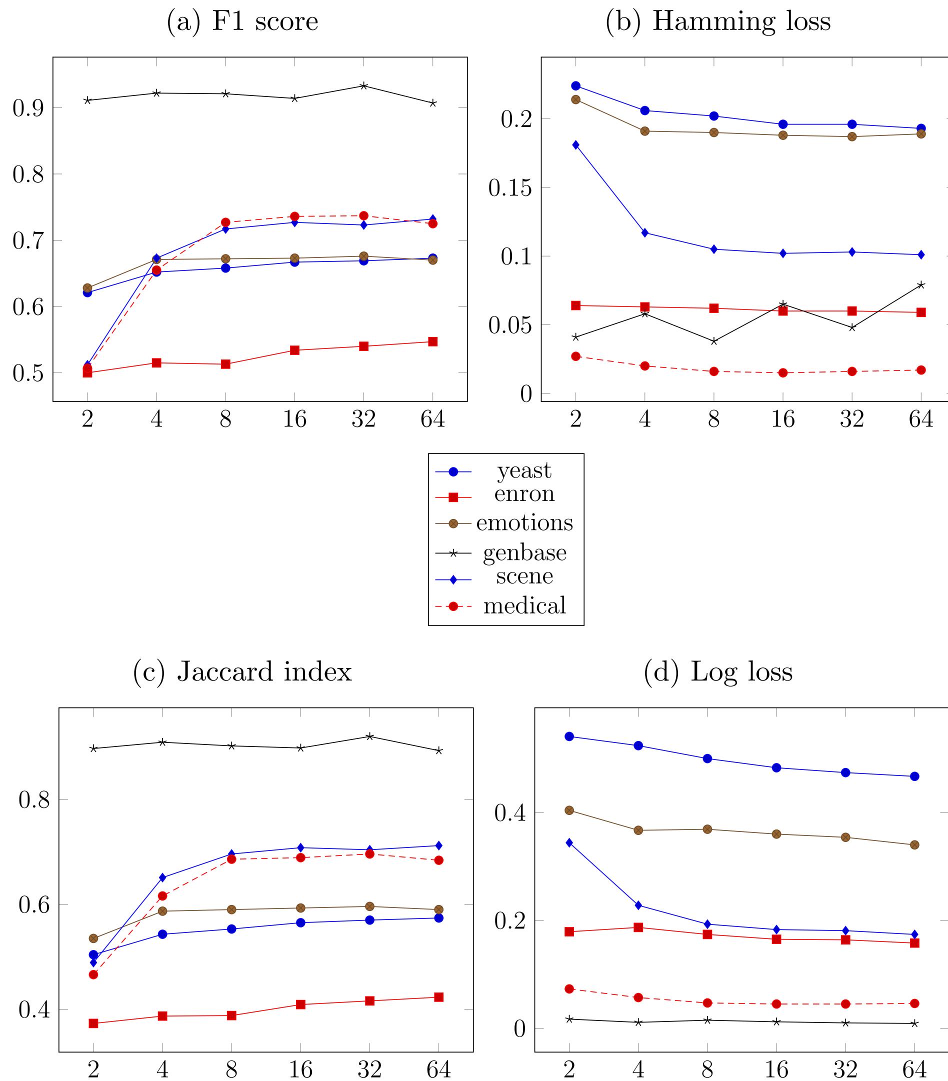
(c) Jaccard index (higher is better)

Datasets	MANIAC	BR	MLkNN	ECC	NJE-BR	NJE-MLkNN	NJE-ECC
yeast	0.450 (1)	0.551 (3)	0.546 (2)	0.556 (4)	**0.565 (7)**	0.561 (5)	0.564 (6)
enron	0.339 (4)	0.260 (2)	0.324 (3)	0.260 (1)	0.409 (5)	0.413 (6)	**0.414 (7)**
medical	**0.704 (7)**	0.562 (2)	0.616 (3)	0.536 (1)	0.698 (6)	0.656 (4)	0.691 (5)
emotions	0.385 (1)	0.559 (3)	0.551 (2)	0.575 (4)	**0.593 (7)**	0.589 (6)	0.576 (5)
genbase	**0.979 (7)**	0.930 (4)	0.946 (6)	0.894 (2)	0.898 (3)	0.775 (1)	0.942 (5)
scene	0.366 (1)	0.606 (2)	0.631 (4)	0.611 (3)	**0.708 (7)**	0.668 (5)	0.704 (6)
Avg. Rank	3.500	2.667	3.333	2.500	**5.833**	4.500	5.667

(d) Log loss (averaged over labels, lower is better)

Datasets	MANIAC	BR	MLkNN	ECC	NJE-BR	NJE-MLkNN	NJE-ECC
yeast	0.552 (7)	**0.422 (1)**	0.431 (2)	0.435 (3)	0.483 (4)	0.502 (5)	0.503 (6)
enron	0.194 (6)	0.179 (5)	**0.157 (1)**	0.226 (7)	0.165 (2)	0.169 (3)	0.174 (4)
medical	0.047 (4)	0.049 (5)	0.046 (2)	0.057 (7)	**0.045 (1)**	0.047 (3)	0.050 (6)
emotions	0.597 (7)	0.399 (5)	0.416 (6)	0.372 (4)	0.360 (3)	**0.358 (1)**	0.358 (2)
genbase	**0.006 (1)**	0.015 (5)	0.014 (3)	0.023 (7)	0.012 (2)	0.015 (4)	0.016 (6)
scene	0.456 (7)	0.245 (6)	0.232 (5)	0.227 (4)	0.183 (2)	0.196 (3)	**0.182 (1)**
Avg. Rank	5.333	4.500	3.167	5.333	**2.333**	3.167	4.167

Figure 6.2: A demonstration of the impact that the target vector size has on the performance of the nonlinear models, as measured by several standard multi-label evaluation measures. The same legend applies to all plots.



local optimisation, we may find low quality local minima. In practice, it can be observed that the local minima reached during optimisation are of similar quality across multiple runs.

6.4.3 Visualisation

An interesting aspect of the proposed framework is the ability to reduce the dimensionality of the data to an arbitrary size, providing the ability to create two dimensional visualisations of multi-label data. Figure 6.3 uses the scene dataset to demonstrate how this can be an effective visualisation method.

These images are the result of plotting the output of a nonlinear metric learned using NJE, with $t = 2$. An ensemble of random forests trained using the BR scheme was used to perform multi-target regression. A separate plot is used for each label and, possibly because the label cardinality of this dataset is quite low, it can be seen that each label is mapped primarily to one cluster. There are several small groups of instances that appear between clusters, and one can see that the instances in these groups have multiple labels and generally belong to the classes associated with both nearby clusters.

Figure 6.4 shows a similar visualisation for the emotions dataset. This dataset has a noticeably higher label cardinality, which has resulted in a more interesting visualisation. Each label is associated with multiple clusters, and each cluster is also associated with multiple labels, as one would expect from multi-label data. This requirement that certain clusters must be adjacent, as enforced by the correlations between labels, is something that is absent from distance metric learning for multiclass classification.

6.5 Conclusion

This chapter has introduced methods that learn linear and nonlinear distance metrics aimed at improving the performance of k-NN applied to multi-label data. The linear metric learning approach appears to add little benefit in most scenarios, but can improve performance for problems with high dimensional data. The proposed nonlinear extension is more beneficial to predictive performance. The nonlinear extension enables use of random forests for performing a wider class of transformations, relative to the linear method, when computing distances between instances. In addition to improving classification performance, the embedding components of the nonlinear metrics are effective tools for multi-label data visualisation.

In the future, it may be interesting to explore other ways in which the proposed loss function can be applied to nonlinear models. For example, a deep

Figure 6.3: Visualisation of a random sample of the scene dataset. Each plot indicates the presence (red) or absence (blue) of a label for each instance.

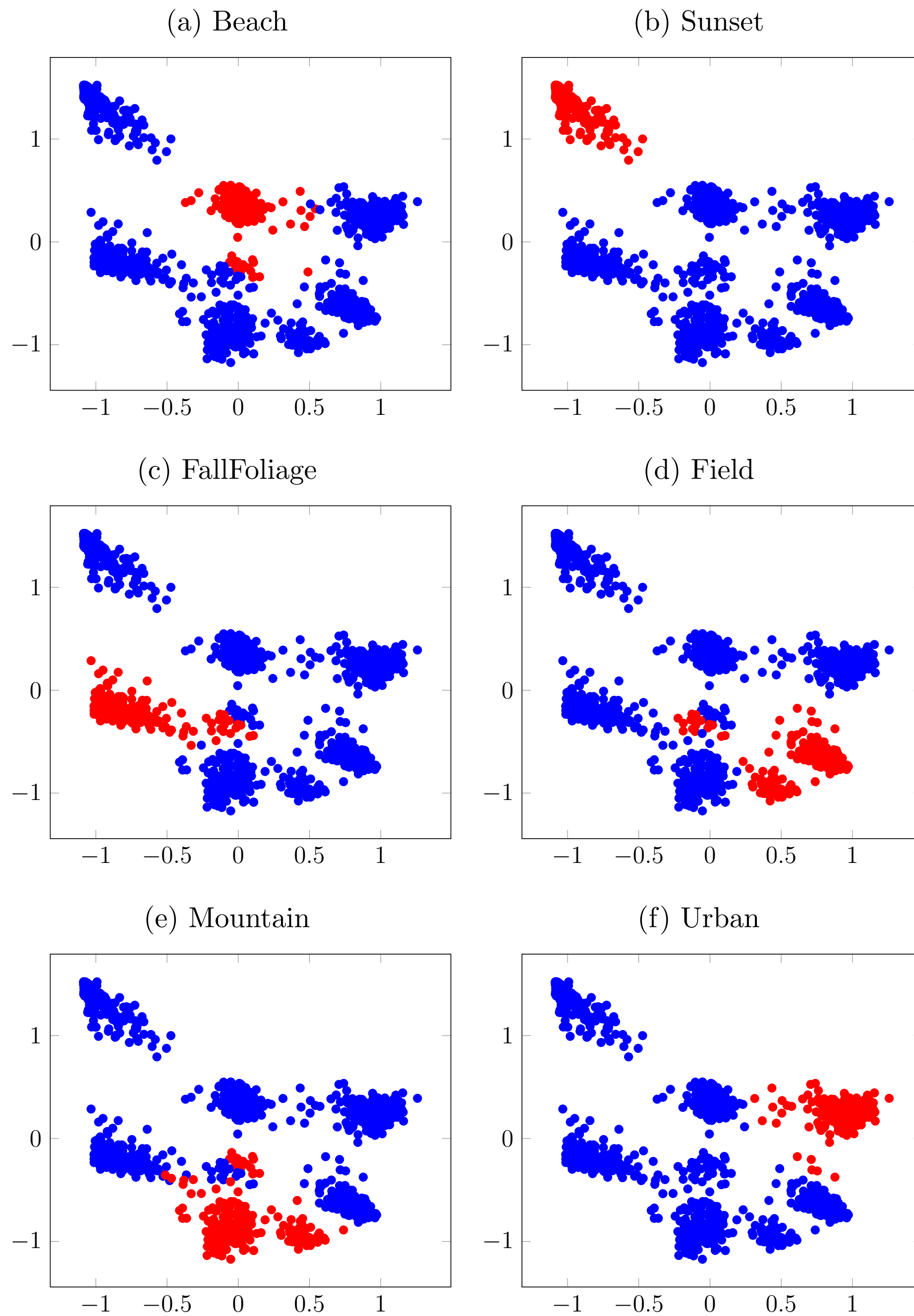
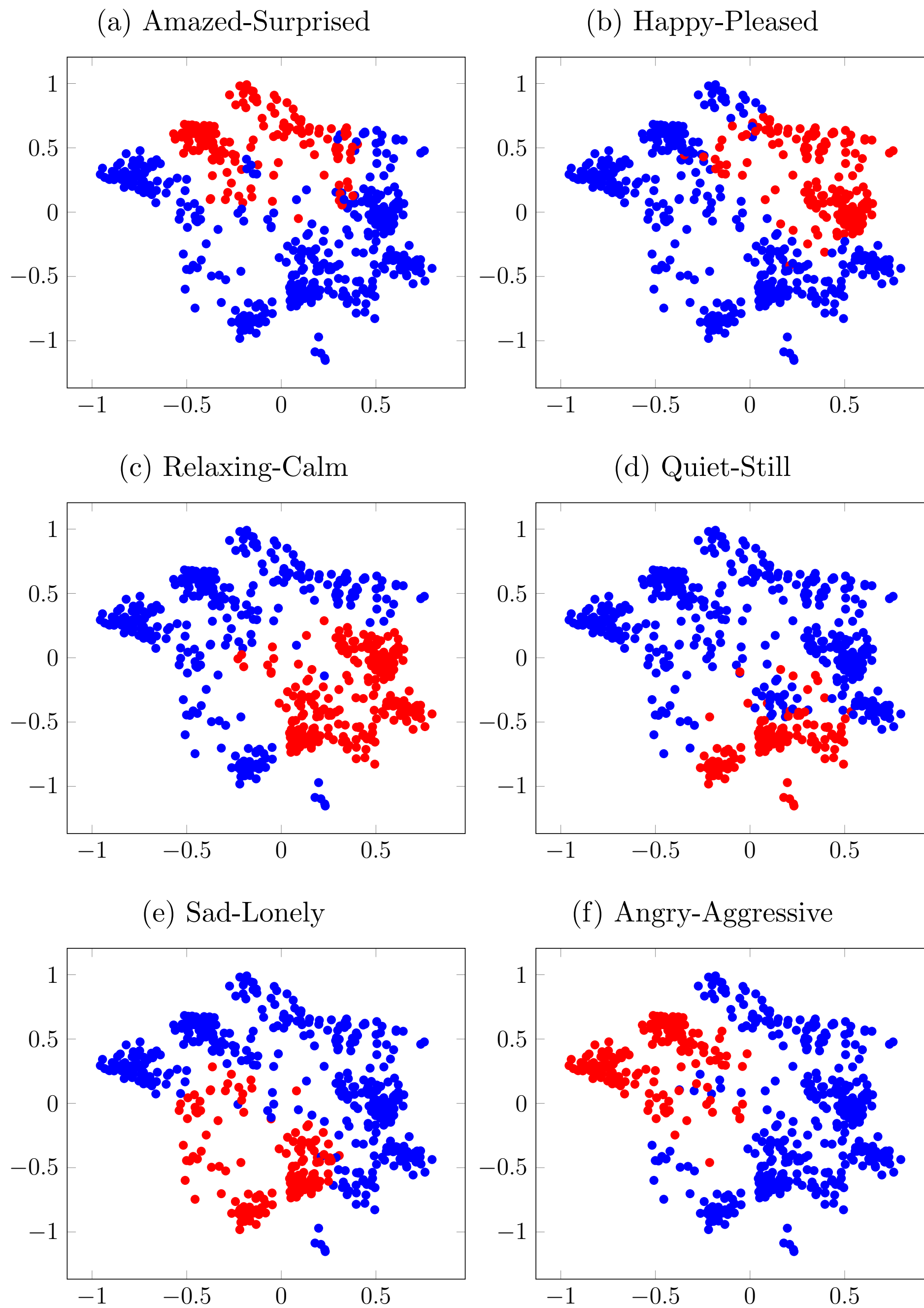


Figure 6.4: Visualisation of the emotions dataset. Each plot indicates the presence (red) or absence (blue) of a label for each instance.



convolutional neural network trained with this loss function could be used to perform web image tag prediction. It would also be interesting to investigate the scalability of this approach to large datasets—likely taking advantage of an algorithm specialised for multi-target regression instead of using a problem transformation method that requires one to train a large number of single-target regression models.

Chapter 7

Online Learning of Discrete Representations

Deep learning has received a lot of attention in a wide number of application domains—particularly computer vision and natural language processing (LeCun et al., 2015). To a significant extent, this is due to the convenient framework supplied by automatic differentiation coupled with gradient-based optimisation methods. This allows different inductive biases to be combined with very little technical effort. For example, a convolutional network can be trained to perform many different tasks by simply changing the loss function. Similar remarks can be made for recurrent neural networks trained and text data. However, on tabular data, neural networks are usually outperformed by decision tree induction methods, which in standard implementations that employ "hard" splits, do not fit into the gradient-based optimisation framework that makes deep learning so ubiquitous. Chapter 6 presented a problem transformation approach to representation learning with nondifferentiable models. In this chapter, we aim to bridge the gap between decision tree induction and gradient-based representation learning, thus providing a principled way to backpropagate from a neural network into a conventional decision tree with hard splits. We hypothesise that this will result in a class of models that have an inductive bias suited to modelling tabular data, but with the flexibility of

reusing this same inductive bias for a variety of tasks.

The online nature of the optimisation algorithms used for training neural networks mandates that any potential decision tree component of the model must also be learned in an incremental manner to enable the joint optimisation desideratum to be met. To this end, we develop an incremental algorithm for constructing decision trees that uses gradient information as a supervision signal. We adapt the Taylor series approximation method used in gradient boosting techniques but reformulate it in such a way that one can train a single decision tree to optimise a given loss function, rather than requiring an ensemble of trees.

Previous approaches to incremental training of decision trees are primarily based on the Hoeffding tree algorithm of Domingos and Hulten (2000), which uses a hypothesis test based on the Hoeffding concentration inequality to determine whether a split should be made. We show how this type of hypothesis test is cumbersome to perform when learning from gradients, and instead make use of one-sample t-tests. Experiments in Section 7.4.1 show that our method performs favourably when compared to state of the art incremental data mining methods.

After introducing the incremental decision tree learner that can be trained with gradients, we show how an ensemble of these trees can be used to construct a novel neural network layer. This tree-based layer can be used as a feature extractor for tabular data, similar to how a block of convolutional layers is often used to extract features from images. We compare neural networks with this novel feature extractor to conventional neural networks and common decision tree ensemble methods. The results of these experiments show that the network with the tree layer outperforms a conventional neural network on a variety of datasets, and approaches the performance of state of the art decision tree ensembles.

7.1 Related Work

Hoeffding trees (Domingos and Hulten, 2000) are commonly used decision trees for incremental learning. In each leaf node, they maintain a co-occurrence histogram reflecting the association between feature values and classes and apply the Hoeffding concentration inequality to determine whether there is enough evidence to identify the best split. Various extensions for Hoeffding trees exist: mechanisms for adapting to concept drift (Bifet and Gavaldà, 2009), specialised ensembling approaches (Bifet et al., 2010a; Pfahringer et al., 2007), and hybrid approaches that build simpler models in each leaf node (Gama et al., 2003). Recent work has also investigated alternative strategies for selecting splits, such as splitting on the first attribute that proves to be a statistically significant improvement over the current tree, rather than trying to find the best possible split before modifying the tree (Manapragada et al., 2018). We also employ this strategy but do not attempt to revisit the split if it becomes apparent that another attribute would have provided better gain in predictive performance. There are also extensions that can perform regression: FIMT-DD (Ikonomovska et al., 2011b), which can build model trees, and ORTO (Ikonomovska et al., 2011a), which takes advantage of option nodes.

The batch decision tree learners most related to our work are those based on gradient boosting Friedman (2001). Conventional gradient tree boosting expands an ensemble of trees using a sequential update rule to find the tree that maximally improves the current ensemble. The algorithm in Friedman (2001) uses the derivative of a loss function, such as cross entropy or squared error, to determine how much the new tree should change the output of the ensemble, implementing a form of gradient descent in function space. We take inspiration from XGBoost (Chen and Guestrin, 2016), which instead uses a second order approximation to the loss function (thus performing Newton steps in function space). XGBoost learns directly from the gradient and Hessian of the loss function. We adapt this to modify a single tree instead of an ensemble and also take ideas from Hoeffding trees to enable incremental learning. To deal

with numeric attributes, we apply the approach used in LightGBM (Ke et al., 2017), which discretizes numeric attributes into ordinal quantities (Ranka and Singh, 1998; Frank and Witten, 1999) to speed up learning.

The literature also contains work on learning representations using decision trees. Kontschieder et al. (2015) replace the fully connected layers often found at the end of convolutional networks with a soft decision tree that has a predefined structure. Each binary decision node defines a differentiable oblique split (Murthy et al., 1994), giving a probability distribution over which child node an instance should be assigned to. The splits are jointly optimised with the convolutional feature extractor preceding the soft tree, resulting in a model that performs comparably to a conventional convolutional network. Yang et al. (2018) focus on learning interpretable decision trees by training soft decision trees using backpropagation. They use a soft binning function to learn how features should be discretized. The depth of the tree is determined by the number of features in the dataset so the method does not scale to datasets with many features. This is mitigated by training an ensemble where each tree is supplied with a subset of the attributes. This approach yields interpretable models on small tabular datasets, providing accuracy similar to conventional networks. Zhou and Feng (2017) show that an ensemble of ensembles of conventional decision trees can be used to learn useful representations for classification, but their method cannot be generalised to other tasks and requires a very large number of decision models to be competitive with XGBoost(Chen and Guestrin, 2016).

7.2 Stochastic Gradient Trees

In supervised incremental learning, data is of the form $(\vec{x}_t, y_t) \in \mathcal{S}$, a new pair arrives at every time step, t, and the aim is to predict the value of y_t given $\vec{x}_t$. Algorithms for this setting must be incremental and enable prediction at any time step—they cannot wait until all instances have arrived and then train a

model. In this section, we describe our method for incrementally constructing decision trees that can be trained to optimise arbitrary twice-differentiable loss functions. We refer to such models as stochastic gradient trees (SGTs). The first key ingredient is a technique for evaluating splits and computing leaf node predictions using only gradient information, which we adapt from the gradient boosting literature (Friedman, 2001; Chen and Guestrin, 2016). Secondly, to enable loss functions that have unbounded gradients, we employ standard one-sample t-tests rather than hypothesis tests based on the Hoeffding inequality.

7.2.1 Leveraging Gradient Information

We assume a loss function, $l(y, \hat{y})$, that measures how well our predictions, $\hat{y}$, match the true values, y. Training should minimize its expected value, as estimated from the available data at a particular time step, t. Assuming i.i.d. data,

$$\mathbb{E}[l(y, \hat{y})] \approx \frac{1}{t} \sum_{i=1}^{t} l(y_i, \hat{y}_i). \tag{7.1}$$

The predictions $\hat{y}_i$ are obtained from the SGT, f_t. At each time step, we aim to find a modification, u, to the tree that takes a step towards minimising the expected loss:

$$f_{t+1} = f_t + \arg\min_{u} [\mathcal{L}_t(u) + \Omega(u)], \tag{7.2}$$

where

$$\mathcal{L}_t(u) = \sum_{i=1}^{t} l(y_i, f_t(\vec{x}_i) + u(\vec{x}_i)), \tag{7.3}$$

and

$$\Omega(u) = \gamma |Q_u| + \frac{\lambda}{2} \sum_{j \in Q_u} v_u^2(j). \tag{7.4}$$

The Ω term is a regularizer, Q_u is the set of new leaf nodes associated with u, and v_u maps these new leaf nodes to the difference between their predictions and the prediction made by their parent, or how an existing leaf node should be updated. The first term in Ω imposes a cost for each new node added to the tree, and the second term can be interpreted as a prior that encourages the

leaf prediction values to be small. In our experiments, we set λ to 0.1 and γ to 1. Because f_t is a decision tree, u will be a function that represents a possible split of one of its leaf nodes, or an update to the prediction made by a leaf: the addition of f_t and u is actually the act of splitting a node in f_t, or changing the value predicted by an existing leaf node. In the case of Hoeffding trees, and also SGTs, only the leaf that contains $\vec{x}_t$ will be considered for splitting at time t, and information from all previous instances that have arrived in that leaf will be used to determine the quality of potential splits. The algorithm also has the option to leave the tree unmodified if there is insufficient evidence to determine the best split.

There are two obstacles to incrementally training a tree using an arbitrary loss function. Firstly, the splitting criterion must be designed to be consistent with the loss to be minimised. Secondly, the leaf nodes' predictions must be chosen in a manner that is consistent with the loss. Both problems can be overcome by adapting a trick used in gradient boosting (Friedman, 2001) that expands an ensemble of trees by applying a Taylor expansion of the loss function around the current state of the ensemble. We only consider modification of a single tree, therefore the empirical expectation of the loss function can be approximated using a Taylor expansion around the unmodified tree at time t:

$$\mathcal{L}_t(u) \approx \sum_{i=1}^{t} [l(y_i, f_t(\vec{x}_i)) + g_i u(\vec{x}_i) + \frac{1}{2} h_i u^2(\vec{x}_i)], \tag{7.5}$$

where g_i and h_i are the first and second derivatives, respectively, of l with respect to each $\hat{y}_i$ produced so far. Optimisation can be further simplified by eliminating the constant first term inside the summation, resulting in

$$\begin{aligned} \Delta\mathcal{L}_t(u) &= \sum_{i=1}^{t} [g_i u(\vec{x}_i) + \frac{1}{2} h_i u^2(\vec{x}_i)] \\ &= \sum_{i=1}^{t} \Delta l_i(u), \end{aligned} \tag{7.6}$$

which now describes the change in loss due to the split, u.

This function is evaluated for each possible split to find the one that yields the maximum reduction in loss. As in the Hoeffding tree algorithm, at time t, we only attempt to split the leaf node into which $\vec{x}_t$ falls, and we consider splitting on each attribute. For each potential split, we need to decide what values should be assigned to the corresponding leaf nodes. Note that we also consider the option of not performing a split at all, and only updating the prediction made by the existing leaf node.

We introduce some notation to explain our procedure. Firstly, we define what a potential split looks like:

$$u(\vec{x}) = \begin{cases} v_u(q_u(\vec{x})), & \text{if } \vec{x} \in \text{Domain}(q_u) \\ 0, & \text{otherwise} \end{cases} \tag{7.7}$$

where q_u maps an instance in the current leaf node to a leaf node that would be created if the split were performed. We denote the codomain of q_u—the set of leaf nodes that would result from this split—as Q_u. We define the set I_u^j as the set of indices of the instances that would reach the new leaf node, j. The objective can then be rewritten as

$$\Delta\mathcal{L}_t(u) = \sum_{j \in Q_u} \sum_{i \in I_u^j} [g_i v_u(j) + \frac{1}{2} h_i v_u^2(j)], \tag{7.8}$$

which can be rearranged to

$$\Delta\mathcal{L}_t(u) = \sum_{j \in Q_u} [(\sum_{i \in I_u^j} g_i) v_u(j) + \frac{1}{2} (\sum_{i \in I_u^j} h_i) v_u^2(j)], \tag{7.9}$$

which uses the sums of the gradient and Hessian values that have been seen thus far by the node being considered for splitting. The optimal $v_u(j)$ for each candidate leaf can be found by taking the relevant term in Equation 7.9 and

adding the corresponding term from Ω,

$$(\sum_{i\in I_u^j} g_i)v_u(j) + \frac{1}{2}(\sum_{i\in I_u^j} h_i)v_u^2(j) + \frac{\lambda}{2}v_u^2(j), \tag{7.10}$$

then setting the derivative to zero,

$$0 = (\sum_{i\in I_u^j} g_i) + (\lambda + \sum_{i\in I_u^j} h_i)v_u(j), \tag{7.11}$$

and solving for $v_u(j)$, yielding

$$v_u^*(j) = -\frac{\sum_{i\in I_u^j} g_i}{\lambda + \sum_{i\in I_u^j} h_i}. \tag{7.12}$$

Viewing the expected loss as a functional, this induction procedure can be thought of as performing Newton's method in function space. In gradient boosting, the addition of each new tree to the ensemble performs a Newton step in function space. The difference in our approach is that each Newton step consists of modifying a prediction value or performing a single split, rather than constructing an entire tree. For loss functionals that cannot be perfectly represented with the quadratic approximation in Newton-type methods, an SGT can potentially take advantage of gradient information more effectively than trees trained using conventional gradient tree boosting (Chen and Guestrin, 2016; Ke et al., 2017).

7.2.2 Splitting on Numeric Attributes

When splitting on nominal attributes, we create a branch for each value of the attribute, yielding a multi-way split. We deal with numeric attributes by discretizing them using simple equal width binning, similar to approaches used in recent gradient boosting techniques (Ke et al., 2017). A sample of instances from the incoming data is used to estimate the minimum and maximum values of each numeric attribute—if these are not already known in advance. Any

future values that do not lie in the estimated range are clipped. The number of bins and the number of instances used to estimate the range of attribute values are user-provided hyperparameters. In our experiments, we set them to 64 and 1,000, respectively. Given a discretized attribute, we consider all possible binary splits that can be made based on the bin boundaries, thus treating it as ordinal (Frank and Witten, 1999).

7.2.3 Determining when to Split

Equation 7.9 estimates the quality of a split but does not indicate whether a split should be made. Hoeffding trees use the Hoeffding concentration inequality to make this decision. It states that, with some probability $1 - \delta$,

$$\mathbb{E}[\overline{X}] > \overline{X} - \epsilon, \tag{7.13}$$

with

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}, \tag{7.14}$$

where $\overline{X}$ is the sample mean of a sequence of random variables X_i, R is the range of values each X_i can take, and n is the sample size used to calculate $\overline{X}$. Suppose the best split considered at time t is u^a. Let $\overline{L} = \frac{1}{n}\hat{\mathcal{L}}_t(u^a)$ be the mean change in loss if the split were applied, as measured on a sample of $n \leq t$ instances. Thus, if $-\overline{L} > \epsilon$, we know, with $1 - \delta$ confidence, that applying this split will result in a reduction of loss on future instances.

In order to apply the Hoeffding bound, we must know the range, R, of values that can be taken by the n terms, $\Delta\hat{l}_i$, in $\Delta\hat{\mathcal{L}}$. In our application, this would require proving upper and lower bounds of the first and second derivatives of the loss function, and constraining the output of the tree to lie within some prespecified range, thus preventing rapid experimentation with different loss functions for novel tasks—one of the properties that has made deep learning on image and language data so widespread. To circumvent this problem, we instead use Student's t-test to determine whether a split should

be made. The t statistic is computed by

$$t = \frac{\overline{L} - \mathbb{E}[\overline{L}]}{s/\sqrt{n}}, \tag{7.15}$$

where s is the sample standard deviation of L_i and, under the null hypothesis, $\mathbb{E}[\overline{L}]$ is assumed to be zero—i.e., it is assumed that the split does not result in a change in loss. A p value can be computed using the inverse cumulative distribution function of the t distribution and, if p is less than δ, the split can be applied.

This test assumes that $\overline{L}$ follows a normal distribution. Although it cannot be assumed that each L_i will be normally distributed, due to the central limit theorem, we are justified in assuming $\overline{L}$ will be normally distributed for sufficiently large n. Computing s requires estimating the sample variance of L_i, which is made easier by initially considering each of the new leaf nodes, $j \in Q_u$, in isolation:

$$\mathrm{Var}(L_i) = \mathrm{Var}(G_i v_u(j) + \frac{1}{2} H_i v_u^2(j)), \tag{7.16}$$

where G_i and H_i are the random variables representing the gradient and Hessian values, respectively. We intentionally treat $v_u(j)$ as a constant, even though this ignores the correlation between the updates that have been applied to the leaf node predictions and the gradient and Hessian values. Empirically, this does not appear to matter, and it eliminates the need to compute the variance of a quotient of random variables—an expression for which there is no distribution-free solution.

Equation 7.16 cannot be computed incrementally because the $v_u(j)$ are not known until all the data has been seen. It is also infeasible to store all of the gradient and Hessian pairs because this could lead to unbounded memory usage. Instead, the equation can be rearranged using some fundamental

properties of variances to yield

$$\mathrm{Var}(L_i) = v_u^2 \mathrm{Var}(G_i) + \frac{1}{4} v_u^4 \mathrm{Var}(H_i) + v_u^3 \mathrm{Cov}(G_i, H_i), \tag{7.17}$$

where we have dropped the "(j)" for compactness. The variances and covariances associated with each feature value can be incrementally estimated using the method of Welford (1962) and one of the algorithms in Bennett et al. (2009), respectively. When considering numeric splits, the sample statistics collected in all the histogram entries must be combined into two bins: one corresponding to feature values below the split point, and the other to values above the split point. This can be done using the concurrent statistical estimation algorithms given by Bennett et al. (2009), resulting in the sample variance, s^2.

The process used to determine whether enough evidence has been collected to justify a split would be prohibitively expensive to carry out every time a new instance arrives. In practice, we follow a common trend in online decision tree induction and only check whether enough evidence exists to perform a split when the number of instances that have fallen into a leaf node is a multiple of some user specified parameter. As with many incremental decision tree induction implementations, this value is set to 200 by default.

7.3 Ensembles of Stochastic Gradient Trees

One of the advantages of deep networks is their ability to output a vector of predictions. This is useful for tasks such as multiclass classification, multi-label classification, multi-target regression, and learning embeddings. SGTs, as formulated so far, output only a single value and are thus less flexible. However, multiple SGTs, combined in an ensemble, can be trained to produce a vector of outputs. The first type of ensemble we consider is a committee of trees where each component in the output vector is produced by an SGT. The second approach backpropagates from a neural network into a committee of

SGTs, decoupling the number of trees from the number of output components.

7.3.1 Training Committees of SGTs

To produce multiple outputs, a different SGT can be trained to estimate each component of the prediction vector, yielding a simple method for multiclass classification. For an m-class classification problem, $m-1$ SGTs can be trained to produce margin values that are fed into a softmax activation function composed with the cross entropy loss function—the mth margin value can be hard-coded to zero. Similarly, a committee of m SGTs can be trained to produce a label vector for multi-label classification, where more (or less) than one of the m binary labels can be associated with a single instance.

While conceptually simple, this method has some drawbacks. Firstly, if there is a large number of outputs, the computational resources required to construct the model could make training a committee of SGTs impractical. Moreover, for non-convex loss functions, care must be taken to ensure the trees are not initialised in such a way that they lie at a saddle point on the loss surface. This is a very real concern when training embedding models using objectives such as the contrastive loss (Hadsell et al., 2006) or a triplet loss (Chechik et al., 2010). This second issue is not a problem for neural networks, due to their random initialisation.

7.3.2 Training SGT Networks

Our solution to the problems associated with training a committee of SGTs is to use an ensemble of trees as a layer in a neural network. This allows the output dimensionality to be dictated by the number of units in the final layer of the network, rather than the number of trees that will be trained. Typical neural networks are composed of a series of linear transformations interleaved with nonlinear activation functions. The conventional wisdom in the deep learning community is that a large number of layers will result in higher accuracy. We propose using an ensemble of SGTs as the first hidden

layer and conventional fully connected layers for the rest of the network, with all but the output layer also making use of the rectified linear unit activation function. We refer to such networks as SGT networks.

SGTs require second order information to evaluate splits and compute leaf prediction values. Therefore, we must backpropagate these second derivatives through the fully connected layers along with the gradient information usually produced by backpropagation. More formally, let $\phi_{sgt}(\vec{x})$ be a function that produces a vector, where each component is determined by a different SGT. We can write down the equation for an SGT network, f, as

$$f(\vec{x}) = (\phi_{fc}^{(m)} \circ \phi_{relu}^{(m-1)} ... \phi_{relu} \circ \phi_{fc}^{(1)} \circ \phi_{sgt})(\vec{x}), \tag{7.18}$$

where each $\phi_{fc}^{(i)}$ is a fully connected layer with a randomly initialised weight matrix, $W^{(i)}$, and bias vector, $\vec{b}^{(i)}$.

Suppose now that the loss function, l, accepts a vector of predictions, rather than a scalar prediction, and denote the input of layer i as $\vec{x}^{(i)}$. We must compute the first and second derivatives of the loss function with respect to each of the SGTs in the first layer of the network. It is important to note that we do not need to compute the full Hessian matrix for the neural network parameters or activations—we need only compute elements that lie on the diagonal and are associated with the activations of each layer. The first derivatives can be computed using repeated application of the chain rule (i.e., backpropagation). Faà di Bruno's formula (Arbogast, 1800; Faà di Bruno, 1855) can be used in place of the chain rule when dealing with higher order derivatives, resulting in rules for backpropagating second derivatives through fully connected layers,

$$\frac{\partial^2 l}{\partial \vec{z}_j^{(i)^2}} = \sum_{k=1}^{d_{(i+1)}} \frac{\partial^2 l}{\partial \vec{z}_j^{(i+1)^2}} W_{kj}^{(i)^2}, \tag{7.19}$$

and rectified linear units,

$$\frac{\partial^2 l}{\partial \vec{z}_j^{(i)2}} = \mathbb{I}(\vec{z}_j^{(i)} > 0) \frac{\partial^2 l}{\partial \vec{z}_j^{(i+1)2}}. \tag{7.20}$$

In these equations $\vec{z}^{(i)}$ is the output of $\phi^{(i)}$ during a forward propagation, d_i represents the number of components in $\vec{z}^{(i)}$, and $\mathbb{I}$ is the indicator function that evaluates to one when the expression in parentheses is true. The first and second derivatives of l with respect to the output of ϕ_{sgt} can be used to train the committee of SGTs, and the first order derivatives calculated by the conventional backpropagation process can be used for training the fully connected layers using any of the usual optimisation techniques. In our experiments, we use the Adam optimiser (Kingma and Ba, 2014) to learn the weights in the fully connected layers.

The weights in the fully connected layers are initialised using the method of Glorot and Bengio (2010). We also randomly initialise the trees in the SGT layer: a randomly generated split is applied to each tree in the layer before training begins. This split is generated by sampling a random feature, and in the case of numeric attributes, a random boundary between quantiles; the resulting leaf nodes are assigned random prediction values.

7.4 Experiments

In this section, we investigate the performance of the SGT on incremental learning tasks, and also demonstrate how well the SGT layer improves neural network accuracy on tabular data in the batch setting. We implemented our approaches in Java, making use of the MOA framework (Bifet et al., 2010b) for the experiments with incremental learning, WEKA (Hall et al., 2009) for the batch multiclass classification experiments, and MEKA (Read et al., 2016) for the multi-label classification experiments. The implementation is available online.[1]

[1] https://github.com/henrygouk/stochastic-gradient-trees

Table 7.1: Details of the classification and regression datasets used for evaluating incremental decision tree learners. No entry in the # Classes column indicates that the dataset is associated with a regression task.

Dataset	# Features	# Instances	# Classes
Higgs	27	11,000,000	2
HEPMASS	27	10,500,000	2
KDD'99	41	4,898,430	40
Covertype	55	581,012	7
AWS Prices	7	27,410,309	-
Airline	13	5,810,462	-
Zurich	13	5,465,575	-
MSD Year	90	515,345	-

Table 7.2: Mean classification error, model size (number of nodes), and runtime (seconds) of the trees produced by the classification methods on 10 random shuffles of each dataset.

		Higgs	HEPMASS	KDD'99	Covertype
Error	SGT	30.10	14.66	0.27	26.94
	VFDT	30.25	14.81	0.73	32.54
	EFDT	31.46	15.22	0.07	22.05
Model size	SGT	1,933.2	1,289.0	446.3	620.8
	VFDT	8,081.4	7,256.0	160.0	91.8
	EFDT	38,535.1	24,760.2	913.8	3.261.4
Runtime	SGT	384.13	290.46	466.26	32.56
	VFDT	115.68	108.11	33.15	3.04
	EFDT	512.32	425.49	97.61	47.23

7.4.1 Incremental Learning

To gain insight into how well SGTs perform in an incremental learning setting, we evaluate their performance on a collection of large classification and regression datasets summarised in Table 7.1. For each dataset, we report the mean across 10 runs, where the data is randomly shuffled for each run. Absolute error is used to evaluate performance on regression problems and classification error is used otherwise. The standard Hoeffding tree algorithm (VFDT) and the faster extension of Manapragada et al. (2018) (EFDT) are used as baselines for the classification tasks, and the FIMT-DD (Ikonomovska et al., 2011b) and ORTO (Ikonomovska et al., 2011a) methods are used as points of reference for regression. The squared error loss function was used for training the regression

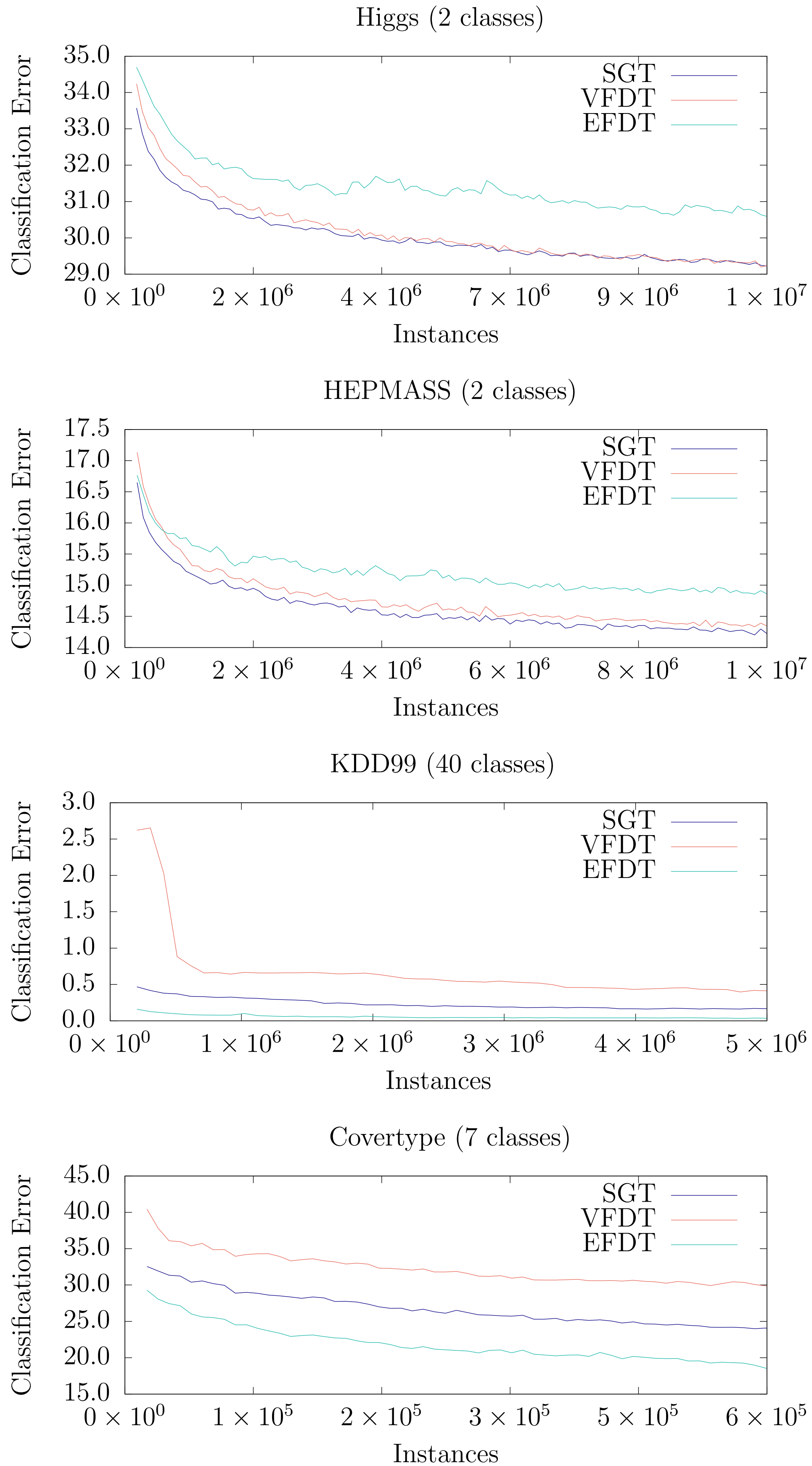


Figure 7.1: Learning curves for the incremental classification problems.

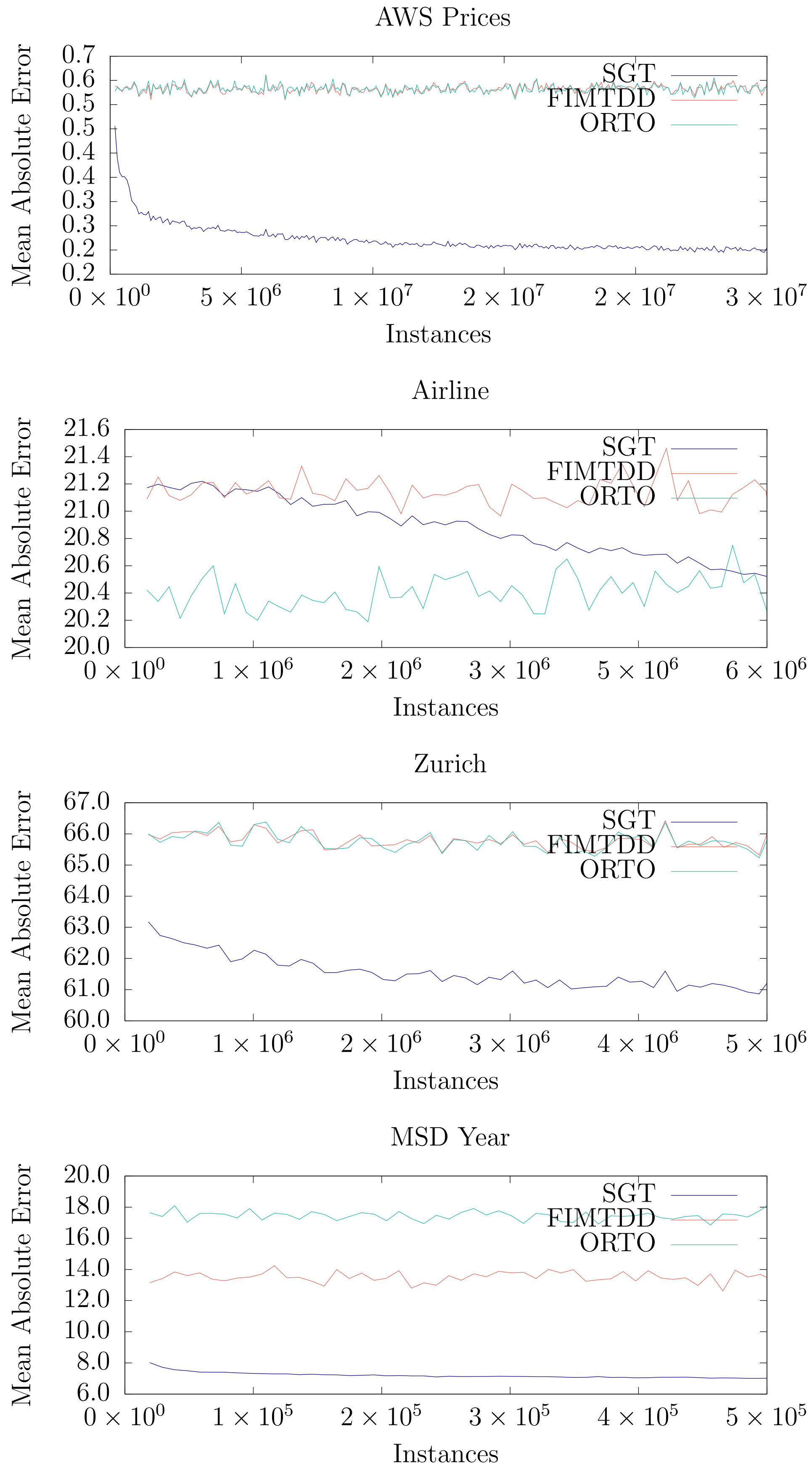


Figure 7.2: Learning curves for the incremental regression problems.

Table 7.3: Mean absolute error, model size (number of nodes), and runtime (seconds) of the trees produced by the regression methods on 10 random shuffles of each dataset.

		Airline	AWS Prices	Zurich	MSD Year
Error	SGT	20.86	0.27	61.59	7.20
	FIMT-DD	21.14	0.58	65.80	13.52
	ORTO	20.41	0.58	65.78	21.76
Model size	SGT	39,287.0	3,316.3	763.1	235.8
	FIMT-DD	30,060.4	165,660.6	21,756.2	2,264.4
	ORTO	33,584.8	177,929.0	23,769.0	2,466.8
Runtime	SGT	22.76	79.96	33.18	58.23
	FIMT-DD	22.43	80.89	29.71	70.75
	ORTO	50.16	42.66	22.16	37.17

trees, and committees of SGTs were trained using the cross entropy loss function, composed with a softmax, for the classification tasks. Learning curves for these experiments are given in Figures 7.1 and 7.2. Statistics on overall performance, model size, and runtime for models trained on the classification and regression datasets are given in Tables 7.2 and 7.3, respectively. From this table and the figure, we can see that SGTs perform similarly to state of the art methods on classification problems, uniformly outperforming VFDT and exhibiting two wins and two losses each compared to EFDT, and exhibit an advantage over existing incremental regression tree methods. On most problems, SGTs also result in smaller model sizes, and take a similar length of time to train compared to the baseline methods. An exception to this is in the multiclass classification scenario, where the model size and training time grows proportionally to the number of classes because a different tree is trained for each class.

7.4.2 Batch Learning

The SGT layer presented in Section 7.3.2 is intended to be used as a feature extractor for tabular data. The first experiment we conduct to characterise the performance of this novel layer is a comparison with standard multiclass classification methods. The datasets used in these experiments were collected from

Table 7.4: 10-fold cross validation results for a collection of datasets found on OpenML (Vanschoren et al., 2013).

Dataset	SGT Network	Neural Network		Random Forest		XGBoost	
adult	86.69±0.48	85.03±0.36	•	85.13±0.69	•	87.27±0.51	◦
bank-marketing	89.10±0.93	89.60±1.45		89.67±0.87		89.80±0.81	
churn	95.18±1.33	92.84±1.27	•	95.10±1.11		95.40±0.79	
electricity	89.00±0.34	84.19±0.69	•	92.47±0.64	◦	91.89±0.37	◦
eye-movements	65.20±1.18	57.53±2.12	•	70.60±1.88	◦	73.85±1.64	◦
internet-usage	88.27±0.71	86.10±1.61	•	87.84±0.83		89.69±1.19	◦
nomao	96.27±0.56	95.87±0.59	•	96.95±0.34	◦	97.31±0.35	◦
phishing	96.47±0.34	96.60±0.61		97.26±0.40	◦	97.11±0.35	◦
speed-dating	100.00±0.00	98.87±0.38	•	92.87±0.75	•	100.00±0.00	
thyroid	98.62±0.62	97.30±0.98	•	98.49±0.75		99.05±0.52	

◦, • statistically significant improvement or degradation, respectively, at 95% confidence.

Table 7.5: Results of 10-fold cross validation on multi-label datasets.

	Average Precision				Macro F1				Accuracy			
Dataset	BR	ECC	NN	SGT	BR	ECC	NN	SGT	BR	ECC	NN	SGT
CAL500	0.345	0.427	0.497	0.504	0.411	0.444	0.313	0.460	0.263	0.291	0.189	0.305
music	0.658	0.778	0.810	0.793	0.562	0.661	0.577	0.634	0.465	0.566	0.503	0.552
scene	0.690	0.833	0.873	0.851	0.583	0.730	0.702	0.725	0.548	0.693	0.686	0.705
yeast	0.582	0.723	0.772	0.727	0.561	0.637	0.631	0.607	0.436	0.523	0.582	0.500
birds	0.713	0.775	0.650	0.735	0.188	0.255	0.078	0.153	0.512	0.598	0.481	0.550
Avg. rank	3.8	2.6	1.8	1.8	3.4	1.2	3.2	2.2	3.6	1.6	3	1.8

OpenML (Vanschoren et al., 2013) by taking the 10 largest datasets containing a mixture of nominal and numeric attributes in the “Categorical Columns Analysis” study, making sure not to include duplicate datasets labelled with different names. The SGT network trained on these datasets contains an SGT layer with 100 trees and two fully connected hidden layers, each with 100 units. The neural network without the SGT layer contains three fully connected hidden layers with 100 units each. Adam (Kingma and Ba, 2014) is used to optimise the neural networks and a validation set consisting of 10% of the training data is used to determine how many epochs should be used for training. We have included random forest (Breiman, 2001) and XGBoost (Chen and Guestrin, 2016) baselines to give an indication of how well state of the art tree based methods perform on tabular data. Each of these tree ensemble approaches is trained with 100 trees and default parameters otherwise. The results of performing 10-fold cross validation are given in Table 7.4, where statistically significant differences have been tested with a corrected resampled t-test (Nadeau and Bengio, 2000). They show that the network with the SGT layer exhibits superior performance on tabular data compared to a conventional neural network. The SGT network also comes close to the performance of random forests, but it is outperformed by the gradient boosted decision tree models produced by XGBoost.

Neural networks are commonly applied in settings where model outputs have a more complicated structure than a single category or numeric value. For example, convolutional networks are often applied to segmentation problems, where a segmentation mask is generated. A analogous task on tabular data is multi-label prediction, where a set of labels must be assigned to each input instance. To demonstrate that the representations learned by SGT networks are well-suited to such structured prediction problems, we compare our method to a standard neural network and two common multi-label classification methods. We use the same SGT network and neural network as was used for the multiclass classification experiments, but we instead use the logistic

activation function and a sum of binary cross entropy loss functions. The two multi-label classification methods we use are the binary relevance method and the ensemble of classifier chains technique of Read et al. (2011). Both of these are problem transformation methods that create an ensemble of classifiers in order to produce the set of predicted labels. The binary relevance approach simply trains a classifier for each potential label, whereas the classifier chain method attempts to exploit correlations between labels. We use the J48 algorithm in WEKA as the base classifier for these two ensemble methods.

The results of running 10-fold cross validation on a collection of multi-label datasets is given in Table 7.5. Wu and Zhou (2017) show that many multi-label classifiation evaluation measures fit into one of two groups: those that favour instance-wise performance, and those that favour label-wise performance. We report both average precision (emphasising label-wise performance) and macro F1 (emphasising instance-wise performance). We also report the Jaccard index (also known as multi-label accuracy), because it is another common measure. It was not considered by Wu and Zhou (2017). Looking at the average rank of each method, we can see that the SGT network consistently performs better than, or similarly to, the conventional neural network. The binary relevance method performs the worst, and ensembles of classifier chains perform slightly better than the SGT network.

7.5 Conclusion

This chapter presents an incremental algorithm for constructing decision trees that exclusively uses gradient information as the source of supervision. This algorithm is inspired by Hoeffding trees but uses the t-test instead of the Hoeffding inequality because the former does not require upper bounds on the gradient and Hessian of the loss function. This chapter also shows how to use a neural network as the source of gradient information required for supervision, resulting in a new tree-based network layer that can be used for extracting

features from tabular data. The results given in Section 7.4 demonstrate that the proposed technique is competitive with standard online classification trees, and outperforms state of the art incremental regression trees. It is also shown that the SGT neural network layer provides a clear advantage over conventional neural networks when learning from tabular data.

It is conceivable that the proposed method will allow a more diverse range of incremental learning tasks to be addressed using decision tree models. The reformulation of the Taylor expansion-based loss function given in Section 7.2.1 can also be adapted to work in the batch setting, which has the potential to result in smaller ensemble sizes when training gradient boosted decision tree models. The SGT layer presented in Section 7.3.2 should also enable construction of neural network architectures for complex multi-relational tabular data mining problems, such as click prediction and designing content-based recommendation systems.

Chapter 8

Conclusion

This thesis began by presenting an argument for exploring several new hypothesis spaces in representation learning—in particular, the investigation of two contrasting lines of research with similar goals:

- Constraining the set of functions that a neural network can represent in order to enforce that the output changes slowly, motivated by the hypothesis that a function that changes slowly will have better generalisation performance;

- Adapting decision tree methods to fit into the gradient-based optimisation framework that has enabled deep learning to become so widespread, thus providing a more suitable inductive bias for representation learning models in domains where decision trees perform better.

In regards to the first point, Chapters 4 and 5 have presented techniques for constraining the Lipschitz constant of a network and an empirical estimate of the Lipschitz constant, respectively. Both of these methods result in improved performance on unseen data when used in isolation, and also when they are used in conjunction with existing regularisation methods. The Lipschitz constant constraint (LCC) regulariser presented in Chapter 4 has the attractive property of fitting directly into the projected stochastic gradient descent optimisation framework. However, its hyperparameter can be difficult to tune and the method exhibited a reliance on being used in conjunction

with batch normalisation to achieve the best results. The MaxGain regularisation method presented in Chapter 5 does not fit into the projected stochastic gradient descent framework, but demonstrates better performance than LCC in isolation; moreover, the optimal hyperparameter setting was stable across different network architectures and datasets.

The second line of enquiry has been addressed by the problem transformation method presented in Chapter 6, and the gradient-based decision incremental tree learning approach introduced in Chapter 7. The problem transformation approach has the advantage that it is able to reuse any batch regression tree learner to construct the model. This implies that future advances in learning regression trees will translate to gains in tree-based representation learning using this framework. The approach was evaluated using multi-label classification as a case study, and resulted in improvements in performance over similar methods on a variety of datasets. It has the downside that it is not based on an end-to-end differentiable architecture, and is therefore much less flexible than neural network approaches to representation learning.

The stochastic gradient tree (SGT) algorithm introduced in Chapter 7 alleviates this problem by providing a means to train an ensemble of decision trees using gradient information as the sole source of supervision. Because the method operates in an incremental manner, the trees can be optimised jointly with subsequent fully connected layers in the neural network. The experimental results show that significant gains in accuracy over conventional neural networks can be achieved when using an SGT layer as a feature extractor specialised for tabular data. Futhermore, in the data stream setting, stochastic gradient trees perform similarly to standard incremental classification trees, and outperform state of the art incremental regression trees.

8.1 Future Research Directions

There is a wealth of potential future work that could be based on the research presented in this thesis. Several avenues of research naturally follow from the Lipschitz-based regularisation techniques introduced in Chapters 4 and 5:

- The methods presented in this thesis only consider the idea of constraining each layer of a network in isolation. More holistic approaches to enforcing constraints on the Lipschitz constant—or empirical approximations thereof—that consider multiple layers simultaneously have the potential to behave quite differently and may also result in tighter upper bounds on the true Lipschitz constant of the resulting model.

- There are also applications of Lipschitz constant constraints beyond achieving better classification accuracy. Recent work has shown that the Lipschitz constant of a convolutional network plays a role in determining how robust the model is to adversarially crafted input images (Cisse et al., 2017). The simple method presented in Chapter 4 could be used as a basis for training provably robust models.

- Many distributed training methods require a measure for how much the local copy of a model has diverged from the copy stored on a central parameter server (McMahan et al., 2017). The operator norm that corresponds to the Lipschitz constant of each layer, and therefore the entire network, naturally gives rise to a distance measure between two different sets of parameters. This distance measure directly relates to how much the output of the layer will change.

- The methods used to enforce the constraints during optimisation make use of projection functions, which, while simple, may not be the most efficient way to impose the constraint. Alternative optimisation algorithms, such as the alternating direction method of multipliers, can be harder to apply to a particular problem, but often result in a more efficient solver.

The approaches based on training ensembles of decision trees to represent data can also be built upon:

- The problem transformation method presented in Chapter 6 used multi-label classification as an illustrative example. Exploring more general embedding loss functions, like the triplet loss of Chechik et al. (2010), would enable relative similarity of instances to be specified in a more general way.

- Because the problem transformation method only takes advantage of the relationships between labels, the intermediate target vectors are not influenced at all by the inductive bias of the model that actually learns the mapping from the feature space to the target vector space. Using a different optimisation procedure that alternates between optimising the target vectors and training the embedding model could resolve this issue.

- Investigating the efficacy of training SGT networks using embedding loss functions would be of interest. In Chapter 7, only classification and multi-label classification tasks were considered, but learning to embed instances would enable one to solve tasks involving similarity searches. Examples of potential applications include content-based recommender systems and information retrieval.

- SGTs, as formulated in Chapter 7, can accept gradients as a form of supervision but lack the ability to produce gradients that can be back-propagated to earlier layers in a network. This means they are locked into the role of a feature extractor for tabular data. Developing a model tree variant of SGTs is one way to potentially solve this issue.

- Removing the dependence of SGTs on second derivative information would make integration with existing deep learning frameworks easier. It would also have the benefit of making SGTs compatible with policy gradient methods, which could be of great interest to the interpretable reinforcement learning community.

Finally, both of the research directions pursued in this thesis are based on a single idea: using hypothesis spaces that have an associated measure of model complexity enables induction of models that generalise well. Decision trees contain a finite number of leaf nodes, each of which predict a single constant value for all instances that fall into it. The number of nodes in a tree acts as a sensible measure of model comlexity. The Lipschitz constant was investigated as a measure of model complexity for continuous functions, however it is not the only sensible metric to use in this context. Investigating alternative measures of the complexity of continuous functions would be of great interest. For example, in the variational computer vision literature, where images are often treated as functions with continuous domains, the total variation of a function is often used to measure its complexity. Repurposing this quantity to serve as a measure of model capacity in machine learning is one of logical continuations of the work presented in this thesis.

References

Martin Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, 2016.

Louis François Antoine Arbogast. *Du Calcul Des Dérivations*. Levrault, Strasbourg, 1800.

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein GAN. January 2017.

Peter L. Bartlett and Shahar Mendelson. Rademacher and Gaussian Complexities: Risk Bounds and Structural Results. In *Computational Learning Theory*, volume 2111, pages 224–240. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-42343-0 978-3-540-44581-4.

Peter L Bartlett, Dylan J Foster, and Matus J Telgarsky. Spectrally-normalized margin bounds for neural networks. In *Advances in Neural Information Processing Systems 30*, pages 6240–6249, 2017.

P.L. Bartlett. The sample complexity of pattern classification with neural networks: The size of the weights is more important than the size of the network. *IEEE Transactions on Information Theory*, 44(2):525–536, March 1998.

Christopher Beckham and Christopher Pal. Unimodal Probability Distributions for Deep Ordinal Classification. In *International Conference on Machine Learning*, pages 411–419, July 2017.

J. Bennett, R. Grout, P. Pebay, D. Roe, and D. Thompson. Numerically stable, single-pass, parallel statistics algorithms. In *2009 IEEE International Conference on Cluster Computing and Workshops*, pages 1–8, August 2009.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: A CPU and GPU Math Compiler in Python. In *Proceedings of the 9th Python in Science Conference*, pages 3–10, 2010.

James Bergstra, Brent Komer, Chris Eliasmith, Dan Yamins, and David D. Cox. Hyperopt: A Python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008, July 2015.

Albert Bifet and Ricard Gavaldà. Adaptive Learning from Evolving Data Streams. In *Advances in Intelligent Data Analysis VIII*, Lecture Notes in Computer Science, pages 249–260. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-03915-7.

Albert Bifet, Eibe Frank, Geoffrey Holmes, and Bernhard Pfahringer. Accurate Ensembles for Data Streams: Combining Restricted Hoeffding Trees using Stacking. In *JMLR Workshop and Conference Proceedings 13*, page 16, Tokyo, Japan, 2010a.

Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. MOA: Massive Online Analysis. *Journal of Machine Learning Research*, 11(May): 1601–1604, 2010b.

Remco R. Bouckaert and Eibe Frank. Evaluating the Replicability of Significance Tests for Comparing Learning Algorithms. In *Advances in Knowledge Discovery and Data Mining*, Lecture Notes in Computer Science, pages 3–12. Springer Berlin Heidelberg, 2004. ISBN 978-3-540-24775-3.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, August 1996.

Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, October 2001.

Carla E. Brodley and Paul E. Utgoff. Multivariate Decision Trees. *Machine Learning*, 19(1):45–77, April 1995.

Lawrence Cayton. Algorithms for manifold learning. *University of California at San Diego Technical Report*, 12(1-17):1, 2005.

Gal Chechik, Varun Sharma, Uri Shalit, and Samy Bengio. Large Scale Online Learning of Image Similarity through Ranking. *Journal of Machine Learning Research*, 11:1109–1135, 2010.

Tianqi Chen and Carlos Guestrin. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 785–794, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4232-2.

Moustapha Cisse, Piotr Bojanowski, Edouard Grave, Yann Dauphin, and Nicolas Usunier. Parseval Networks: Improving Robustness to Adversarial Examples. In *International Conference on Machine Learning*, pages 854–863, July 2017.

Jason Cong and Bingjun Xiao. Minimizing Computation in Convolutional Neural Networks. In *Artificial Neural Networks and Machine Learning – ICANN 2014*, Lecture Notes in Computer Science, pages 281–290. Springer International Publishing, 2014. ISBN 978-3-319-11179-7.

Jason V Davis, Brian Kulis, Prateek Jain, Suvrit Sra, and Inderjit S Dhillon. Information-theoretic metric learning. In *Proc. of the 24th ICML*, pages 209–216. ACM, 2007.

Janez Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, 7(Jan):1–30, 2006.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference On*, pages 248–255. IEEE, 2009.

L. Deng, G. Hinton, and B. Kingsbury. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603, May 2013.

Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '00*, pages 71–80, Boston, Massachusetts, United States, 2000. ACM Press. ISBN 978-1-58113-233-5.

John Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

Francesco Faà di Bruno. Sullo sviluppo delle funzioni. *Annali di scienze matematiche e fisiche*, 6:479–480, 1855.

Eibe Frank and Ian H. Witten. Selecting multiway splits in decision trees. Working Paper, December 1996.

Eibe Frank and Ian H Witten. Making Better Use of Global Discretization. In *16th International Conference on Machine Learning*, page 9, 1999.

Jerome H. Friedman. Greedy Function Approximation: A Gradient Boosting Machine. *The Annals of Statistics*, 29(5):1189–1232, 2001.

Nicholas Frosst and Geoffrey Hinton. Distilling a Neural Network Into a Soft Decision Tree. *arXiv:1711.09784 [cs, stat]*, November 2017.

Kunihiko Fukushima. Neocognitron: A self-organizing neural network model

for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36(4):193–202, April 1980.

Yarin Gal, Jiri Hron, and Alex Kendall. Concrete dropout. In *Advances in Neural Information Processing Systems*, pages 3584–3593, 2017.

João Gama, Ricardo Rocha, and Pedro Medas. Accurate Decision Trees for Mining High-speed Data Streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, pages 523–528, New York, NY, USA, 2003. ACM. ISBN 978-1-58113-737-8.

Todor Ganchev, Nikos Fakotakis, and George Kokkinakis. Comparative evaluation of various MFCC implementations on the speaker verification task. In *In Proc. of the SPECOM-2005*, pages 191–194, 2005.

Bolin Gao and Lacra Pavel. On the Properties of the Softmax Function with Application in Game Theory and Reinforcement Learning. *arXiv:1704.00805 [cs, math]*, April 2017.

Pierre Geurts and Louis Wehenkel. Closed-form Dual Perturb and Combine for Tree-based Models. In *Proceedings of the 22Nd International Conference on Machine Learning*, ICML '05, pages 233–240, New York, NY, USA, 2005. ACM. ISBN 978-1-59593-180-1.

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256, March 2010.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

Henry Gouk, Bernhard Pfahringer, Eibe Frank, and Michael J. Cree. MaxGain: Regularisation of Neural Networks by Constraining Activation Magnitudes. In *Machine Learning and Knowledge Discovery in Databases. ECML PKDD 2018.*, Lecture Notes in Computer Science, pages 541–556. Springer International Publishing, 2019. ISBN 978-3-030-10925-7.

Matthieu Guillaumin, Jakob Verbeek, and Cordelia Schmid. Multiple instance metric learning from automatically labeled bags of faces. In *European Conference on Computer Vision*, pages 634–647. Springer, 2010.

Chuan Guo, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. On Calibration of Modern Neural Networks. In *International Conference on Machine Learning*, pages 1321–1330, July 2017.

R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality Reduction by Learning an Invariant Mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742, June 2006.

Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H. Witten. The WEKA Data Mining Software: An Update. *SIGKDD Explor. Newsl.*, 11(1):10–18, November 2009.

Moritz Hardt, Ben Recht, and Yoram Singer. Train faster, generalize better: Stability of stochastic gradient descent. In *International Conference on Machine Learning*, pages 1225–1234, June 2016.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the Knowledge in a Neural Network. *arXiv:1503.02531 [cs, stat]*, March 2015.

Geoffrey E. Hinton. Learning translation invariant recognition in a massively parallel networks. In *PARLE Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science, pages 1–13. Springer Berlin Heidelberg, 1987. ISBN 978-3-540-47144-8.

Sepp Hochreiter. *Untersuchungen Zu Dynamischen Neuronalen Netzen.* Diploma, Technische Universität München, 1991.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997.

Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, January 1989.

Todd Huster, Cho-Yu Jason Chiang, and Ritu Chadha. Limitations of the Lipschitz constant as a defense against adversarial examples. *arXiv:1807.09705 [cs, stat]*, July 2018.

Elena Ikonomovska, João Gama, and Sašo Džeroski. Learning model trees from evolving data streams. *Data Mining and Knowledge Discovery*, 23(1): 128–168, July 2011a.

Elena Ikonomovska, João Gama, Bernard Ženko, and Sašo Džeroski. Speeding Up Hoeffding-Based Regression Trees with Options. In *28th International Conference on Machine Learning*, page 8, Bellevue, WA, USA, 2011b.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

Rong Jin, Shijun Wang, and Zhi-Hua Zhou. Learning a distance metric from multi-instance multi-label data. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference On*, pages 896–902. IEEE, 2009.

Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems 31*, pages 3146–3154, 2017.

Michael J. Kearns and Robert E. Schapire. Efficient distribution-free learning of probabilistic concepts. *Journal of Computer and System Sciences*, 48(3): 464–497, June 1994.

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

Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. Deep Neural Decision Forests. In *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1467–1475, Santiago, Chile, December 2015. IEEE. ISBN 978-1-4673-8391-2.

Alex Krizhevsky and Geoffrey Hinton. *Learning Multiple Layers of Features from Tiny Images*. Master's Thesis, University of Toronto, 2009.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25*, pages 1097–1105, 2012.

Anders Krogh and John A. Hertz. A Simple Weight Decay Can Improve Generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957. Morgan-Kaufmann, 1992.

Ron Larson. *Elementary Linear Algebra*. Nelson Education, 2016.

Quoc V. Le, Jiquan Ngiam, Adam Coates, Abhik Lahiri, Bobby Prochnow, and Andrew Y. Ng. On Optimization Methods for Deep Learning. In *Proceed-*

ings of the 28th International Conference on International Conference on Machine Learning, ICML'11, pages 265–272, USA, 2011. Omnipress. ISBN 978-1-4503-0619-5.

Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 1(4):541–551, December 1989.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, May 2015.

Weiwei Liu and Ivor W Tsang. Large Margin Metric Learning for Multi-Label Prediction. In *AAAI*, pages 2800–2806, 2015.

David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

Chaitanya Manapragada, Geoff Webb, and Mahsa Salehi. Extremely Fast Decision Tree. In *Knowledge Discovery in Databases 2018*, London, United Kingdom, August 2018. ACM.

H Brendan McMahan, Eider Moore, Daniel Ramage, and Seth Hampson. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, page 10, 2017.

Renqiang Min. Deep Supervised t-Distributed Embedding. In *27th International Conference on Machine Learning*, page 8, Haifa, Israel, 2010.

Tom M. Mitchell. The Need for Biases in Learning Generalizations. Technical report, 1980.

Takeru Miyato, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. Spectral Normalization for Generative Adversarial Networks. In *6th International Conference on Learning Representations*, 2018.

S. K. Murthy, S. Kasif, and S. Salzberg. A System for Induction of Oblique Decision Trees. *Journal of Artificial Intelligence Research*, 2:1–32, August 1994.

Claude Nadeau and Yoshua Bengio. Inference for the Generalization Error. In *Advances in Neural Information Processing Systems*, pages 307–313, 2000.

Behnam Neyshabur. *Implicit Regularization in Deep Learning.* PhD thesis, Toyota Technilogical Institute at Chicago, September 2017.

Behnam Neyshabur, Srinadh Bhojanapalli, and Nathan Srebro. A PAC-Bayesian Approach to Spectrally-Normalized Margin Bounds for Neural Networks. In *International Conference on Learning Representations*, February 2018.

Hung Son Nguyen. A Soft Decision Tree. In *Intelligent Information Systems 2002*, Advances in Soft Computing, pages 57–66. Physica-Verlag HD, 2002. ISBN 978-3-7908-1777-5.

Omkar M. Parkhi, Andrea Vedaldi, and Andrew Zisserman. Deep Face Recognition. In *Procedings of the British Machine Vision Conference 2015*, pages 41.1–41.12, Swansea, 2015. British Machine Vision Association. ISBN 978-1-901725-53-7.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. October 2017.

Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. New Options for Hoeffding Trees. In *AI 2007: Advances in Artificial Intelligence*, Lecture

Notes in Computer Science, pages 90–99. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76928-6.

Sanjay Ranka and V Singh. CLOUDS: A decision tree classifier for large datasets. In *Proceedings of the 4th Knowledge Discovery and Data Mining Conference*, volume 2, 1998.

Jesse Read, Bernhard Pfahringer, Geoff Holmes, and Eibe Frank. Classifier chains for multi-label classification. *Machine learning*, 85(3):333–359, 2011.

Jesse Read, Peter Reutemann, Bernhard Pfahringer, and Geoff Holmes. MEKA: A Multi-label/Multi-target Extension to WEKA. *JMLR*, 17(21): 1–5, 2016.

David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533, October 1986.

Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3): 211–252, December 2015.

Tim Salimans and Diederik P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *Advances in Neural Information Processing Systems*, pages 901–909, 2016.

Cullen Schaffer. A Conservation Law for Generalization Performance. In *Machine Learning Proceedings 1994*, pages 259–265, San Francisco (CA), January 1994. Morgan Kaufmann. ISBN 978-1-55860-335-6.

Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

L. N. Smith. Cyclical Learning Rates for Training Neural Networks. In *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 464–472, March 2017. ISBN 978-1-5090-4822-9.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural networks for machine learning*, 4(2):26–31, 2012.

Joel Aaron Tropp. *Topics in Sparse Approximation.* Thesis, University of Texas at Austin, 2004.

Grigorios Tsoumakas, Eleftherios Spyromitros-Xioufis, Jozef Vilcek, and Ioannis Vlahavas. Mulan: A java library for multi-label learning. *JMLR*, 12: 2411–2414, 2011.

Yusuke Tsuzuku, Issei Sato, and Masashi Sugiyama. Lipschitz-Margin Training: Scalable Certification of Perturbation Invariance for Deep Neural Networks. *arXiv:1802.04034 [cs, stat]*, February 2018.

Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. Deep content-based music recommendation. In *Advances in Neural Information Processing Systems 26*, pages 2643–2651. Curran Associates, Inc., 2013.

Twan van Laarhoven. L2 Regularization versus Batch and Weight Normalization. *arXiv:1706.05350 [cs, stat]*, June 2017.

Joaquin Vanschoren, Jan N. van Rijn, Bernd Bischl, and Luis Torgo. OpenML:

Networked Science in Machine Learning. *SIGKDD Explorations*, 15(2):49–60, 2013.

V. N. Vapnik and A. Ya. Chervonenkis. On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities. In *Measures of Complexity: Festschrift for Alexey Chervonenkis*, pages 11–30. Springer International Publishing, Cham, 2015. ISBN 978-3-319-21852-6.

Li Wan, Matthew Zeiler, Sixin Zhang, Yann Le Cun, and Rob Fergus. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pages 1058–1066, 2013.

Kilian Q Weinberger, John Blitzer, and Lawrence K Saul. Distance metric learning for large margin nearest neighbor classification. In *Advances in Neural Information Processing Systems*, pages 1473–1480, 2005.

B. P. Welford. Note on a Method for Calculating Corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962.

Paul Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD Thesis, Harvard University, 1974.

Jörg Wicker, Andrey Tyukin, and Stefan Kramer. A Nonlinear Label Compression and Transformation Method for Multi-label Classification Using Autoencoders. In *Advances in Knowledge Discovery and Data Mining*, pages 328–340. Springer, 2016.

David H. Wolpert. The Lack of A Priori Distinctions Between Learning Algorithms. *Neural Computation*, 8(7):1341–1390, October 1996.

Xi-Zhu Wu and Zhi-Hua Zhou. A Unified View of Multi-Label Performance Measures. In *International Conference on Machine Learning*, page 9, Sydney, Australia, 2017.

Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-MNIST: A Novel Image

Dataset for Benchmarking Machine Learning Algorithms. *arXiv:1708.07747 [cs, stat]*, August 2017.

Yongxin Yang, Irene Garcia Morillo, and Timothy M. Hospedales. Deep Neural Decision Trees. *arXiv:1806.06988 [cs, stat]*, June 2018.

Yuichi Yoshida and Takeru Miyato. Spectral Norm Regularization for Improving the Generalizability of Deep Learning. *arXiv:1705.10941 [cs, stat]*, May 2017.

Sergey Zagoruyko and Nikos Komodakis. Wide Residual Networks. In *Proceedings of the British Machine Vision Conference (BMVC)*, September 2016.

Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530*, 2016.

Min-Ling Zhang and Zhi-Hua Zhou. ML-KNN: A lazy learning approach to multi-label learning. *Pattern recognition*, 40(7):2038–2048, 2007.

Zhi-Hua Zhou and Ji Feng. Deep Forest: Towards An Alternative to Deep Neural Networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 3553–3559, Melbourne, Australia, August 2017. International Joint Conferences on Artificial Intelligence Organization. ISBN 978-0-9992411-0-3.