LEARNING DATA REPRESENTATIONS FOR ROBUST NEIGHBOUR-BASED INFERENCE

by

Tharindu Salinda Bandara Adikari Mudiyanselage

A thesis submitted in conformity with the requirements for the degree of Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto

© Copyright 2018 by Tharindu Salinda Bandara Adikari Mudiyanselage
Abstract

Learning data representations for robust neighbour-based inference

Tharindu Salinda Bandara Adikari Mudiyanselage
Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical & Computer Engineering
University of Toronto
2018

The recently proposed *Boundary Trees* algorithm by Mathy et al. (2015) enables fast neighbour-based classification, regression and retrieval in large datasets. While boundary trees use a Euclidean measure of similarity, the *Differentiable Boundary Tree* (DBT) algorithm by Zoran et al. (2017) was introduced to learn low-dimensional representations of complex input data, on which semantic similarity can be measured, to train boundary trees. The DBT approach contains a few limitations that prevents it from scaling to large datasets. In this thesis, we introduce *Differentiable Boundary Sets*, an algorithm that overcomes the computational issues of the DBT scheme and also improves its classification accuracy and data representability. Our algorithm is efficiently implementable with existing tools and offers a significant reduction in training time. We test and compare the proposed algorithm on the well known MNIST handwritten digits dataset and the newer Fashion-MNIST dataset by Xiao et al. (2017).
Contents

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Contribution ................................................. 2
  1.3 Thesis outline .............................................. 2

2 Background 3
  2.1 Neighbour-based classification ................................ 3
    2.1.1 Nearest neighbour classification ........................ 3
    2.1.2 Boundary trees and forests ............................... 5
    2.1.3 Other related work ...................................... 7
  2.2 Representation learning ...................................... 8
    2.2.1 Multi-layer neural networks ............................ 8
    2.2.2 Siamese networks ....................................... 8
    2.2.3 Triplet networks ....................................... 10
    2.2.4 Differentiable boundary trees ......................... 11
  2.3 Adversarial examples ....................................... 14
    2.3.1 Generating adversarial examples ..................... 14
    2.3.2 Adversarial training .................................. 15

3 Methodology 18
  3.1 Motivation .................................................. 18
  3.2 Boundary sets .............................................. 20
  3.3 Differentiable boundary set algorithm ..................... 20
  3.4 Boundary sets and adversaries ............................ 21

4 Experiments 23
  4.1 Differentiable boundary sets ............................... 23
  4.2 Boundary sets and adversaries ............................ 26

5 Conclusions 28
  5.1 Summary ................................................... 28
  5.2 Future work ................................................. 28

Bibliography 29
Chapter 1

Introduction

In machine learning and statistics, classification refers to the problem of identifying to which of a set of categories a new observation belongs. Neighbour-based classification is arguably one of the oldest classification methods. Neighbour-based classification is based on the simple intuition that ‘things that appear similar are likely similar’. The underlying idea behind neighbour-based classification techniques, often referred to as nearest neighbour (NN) methods, can be traced back to the medieval text *Book of Optics* written in early 11th century by eminent scholar Alhazen (Chen and Shah). Over a time span close to a millennium, NN methods have been well studied and used in a variety of applications. However, due to the high computational complexity and inability to handle complex data, NN methods have fallen out of favour. These methods have perhaps also received somewhat less attention in the past few years due to advances in rival algorithms such as support vector machines and neural networks. Nevertheless, neighbour-based methods are making a comeback as they are now becoming feasible for many applications, due in no small part to fast neighbour search algorithms that scale to high-dimensional massive datasets such as the ones proposed by Mathy et al. (2015), Li and Malik (2016) and Andoni and Razenshteyn (2015).

1.1 Motivation

In certain use-cases, there are non-quantitative factors involved when selecting classifiers. Although neighbour-based methods are not always the best performing in terms of accuracy, they have commendable properties that more accurate state-of-the-art methods do not always posses. For example, neighbour-based methods are nonparametric and let data directly drive predictions without making strong modeling assumptions (Chen and Shah). Despite the high accuracy of the state-of-the-art methods like neural network classifiers, their lack of interpretability has hindered their adoption in critical areas like medicine, the criminal justice system, and financial markets. Interpretability, in the context of machine learning literature, refers to interpreting the training method, the learned model, or the inferred result. In addition to generating predictions, a neighbour-based classifier is able to provide similar instances to a given query point, which can be used to help explain its decision making process (Lipton 2016). This is akin to how humans sometimes justify decisions — by analogy. For example, it can help a medical practitioner to decide which treatment to prescribe to a patient if the practitioner is provided with examples of the raw data of patients that had similar symptoms. This can be a significant advantage
of neighbour-based methods when compared to black-box type classifiers such as neural networks.

1.2 Contribution

The work presented in this thesis contributes to the much larger effort of developing efficient neighbour-based methods. We present our main contribution, namely the Differentiable Boundary Set algorithm. Differentiable boundary sets is an approach to efficiently learn approximate nearest neighbour-based classifiers, inspired by the recently proposed Boundary Tree \cite{Mathy2015BoundaryTree} and Differentiable Boundary Tree \cite{Zoran2017DifferentiableBoundaryTree} algorithms. The boundary tree (BT) algorithm is a nonparametric, computationally fast, and memory-efficient approach to neighbour-based classification, regression and retrieval problems. The BT algorithm is also able to learn in an online fashion where training data is made available sequentially. Like other neighbour-based methods, the boundary tree algorithm requires a metric that measures the semantic similarity of training data.

The differentiable boundary tree (DBT) algorithm generalizes the boundary tree idea by, additionally, learning representations of training data in a supervised fashion, on which semantic similarity can be approximated. The DBT algorithm learns a transformation function in conjunction with a BT, therefore, can be considered ‘semiparametric’. The learned representations are then used to compute pairwise similarity of data, towards building a boundary tree. However, as noted by the original authors, batching is infeasible for the proposed DBT scheme. At a high level, batching or batch-implementation takes multiple iterative operations among data points and turns them into matrix operations which can be efficiently executed on modern hardware such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs) \cite{Jouppi2017In-datacenter}. The limitation with regard to batching prevents the DBT algorithm from exploiting fully the benefits offered by modern machine learning tools, which are heavily reliant on batch-implementation. In this thesis, we present the differentiable boundary sets algorithm which overcomes the computational issues of DBT and also improve accuracy and data representability. Our algorithm can be efficiently implemented with existing tools and offers a significant improvement in convergence time.

In our second main contribution we provide a strong motivation to use the proposed DBS method in the place of vanilla neural network classifiers, in presence of adversarial examples. Adversarial examples are generated by an adversary who perturbs valid inputs from a dataset. Even though the perturbations are unnoticeable or do not alter the semantic meaning of the example, an accurate classifier may assign a wrong label to the perturbed version often with high confidence. Our experiments suggest that the DBS method is more resistant to adversarial attacks if the output dimension of the learned transformation function is large.

1.3 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 discusses background and relevant work. We motivate and present the proposed algorithms in Chapter 3. In Chapter 4 we describe the experimental framework used to evaluate the performance of the proposed algorithms. We also present experimental results and compare with existing approaches in Chapter 4 before concluding in Chapter 5.
Chapter 2

Background

2.1 Neighbour-based classification

In this chapter we provide some background for neighbour-based classification methods. Even though we focus only on classification, the presented algorithms are often generalizable to regression problems as well. Throughout this thesis we use $[K]$ to denote the index set $\{1, \ldots, K\}$ for any positive integer $K$. We assume $\mathcal{D} = \{(x_n, y_n) \mid x_n \in \mathbb{R}^D, y_n \in [C], n \in [N]\}$ is a given dataset where $y_n$ is the label of $n$-th data point $x_n$ and $C$ is the number of classes. The set $\mathcal{X} = \{x_n \mid (x_n, y_n) \in \mathcal{D}\}$ and $\mathcal{Y} = \{y_n \mid (x_n, y_n) \in \mathcal{D}\}$ are respectively defined to be the set of data points and labels in $\mathcal{D}$. We use $\|\cdot\|$ to denote the $L_2$ norm, and big O notation, $O(\cdot)$, to describe the computational complexity of algorithms.

2.1.1 Nearest neighbour classification

1-nearest neighbour

Given the training dataset $\mathcal{D}$ and a query data point $x$, the simplest version of neighbour-based classification, known as 1-nearest neighbour (1-NN), is to search the set $\mathcal{X}$ for the closest point to $x$ and assign the label of the search result as the predicted label for $x$. Concisely, the predicted label for $x$ is $y_{n^*}$ where

$$n^* = \arg \min_{n \in [N]} \|x - x_n\|.$$

Since there are no learnable parameters associated with this approach, this is called a nonparametric method. The naive nearest neighbour algorithm is online, meaning that the output can be updated even when training data is made available incrementally. Being a long-known method, there is extensive work on trying to improve the computational, space and time requirements of 1-NN. One major drawback of the 1-NN method is that it is not scalable to large and high dimensional datasets. This can be observed by analyzing the computational complexity of the above algorithm. Since the distance from $x$ to each $x_n$ has to be calculated at least once, the runtime of the algorithm is $O(ND)$ where $D$ is the dimension of data points. Even though the runtime is linear in $N$, $O(ND)$ still is a heavy cost to pay for a single prediction. To put this computational complexity in perspective, a simple parametric method like logistic regression has a constant runtime for a single prediction.
\textit{k-nearest neighbours}

A generalization of the 1-NN method is \textit{k}-NN where, instead of one, the \textit{k} closest data points of \( \mathbf{x} \) are selected and, from their associated labels, the most common label is taken as the prediction. The \textit{k}-NN algorithm can be employed for regression as well by computing a weighted sum of target values of the nearest neighbour points. To identify the closest \textit{k} data points of \( \mathbf{x} \), the Euclidean distance has to be computed from \( \mathbf{x} \) to each element of \( \mathcal{X} \) and the set of computed distances must be searched for the smallest \textit{k} among them. To understand how the computation complexity scales for this nearest neighbour generalization, we outline a possible implementation of \textit{k}-NN in Algorithm 1. The first loop in Algorithm 1 precomputes the distances the same way as done in 1-NN. The computational complexity of this operation is given by \( \mathcal{O}(ND) \). The second loop sequentially searches for the \textit{k}-nearest neighbours in \( \mathcal{O}(kN) \) time. Combining both steps, the total runtime of the algorithm is \( \mathcal{O}(ND + kN) \). In practice \( D \) is much greater than \( k \) and the complexity of the \textit{k}-NN algorithm thus becomes \( \mathcal{O}(ND) \). As mentioned before, since both \( N \) and \( D \) are large values, this means that nearest neighbour algorithms, in their original form, pay a heavy cost even for a single prediction.

\begin{algorithm}
\caption{\textit{k}-NN algorithm}
\begin{algorithmic}
\State \textbf{input} : \( \mathbf{x} \), \( k \), \( \mathcal{X} \)
\State \textbf{output}: \( \mathcal{A} \)
\ForEach {\( n \in [N] \)}
\State \( d_n \leftarrow \| \mathbf{x} - \mathbf{x}_n \| \); \quad \quad \quad \quad // \text{complexity grows linearly with} \ D
\State \( v_n \leftarrow 0 \);
\EndFor
\State \( \mathcal{A} = \Phi \);
\ForEach {\( i \in [k] \)}
\State \( \mathcal{N}_i \leftarrow \) subset of \([N]\) for which \( v_n = 0 \);
\State \( n^* \leftarrow \) index of smallest \( d_n \) for \( n \in \mathcal{N}_i \); \quad \quad // \text{complexity grows linearly with} \ N
\State \( v_{n^*} \leftarrow 1 \);
\State \( \mathcal{A} \leftarrow \mathcal{A} \cup \{ n^* \} \);
\EndFor
\end{algorithmic}
\end{algorithm}

**Approximate nearest neighbours**

Many methods have been proposed to alleviate the computational challenges of nearest neighbour algorithms. As noted by [Zoran et al. (2017)](https://www.robots.ox.ac.uk/~az/algorithms/), these methods can broadly be categorized into tree-based methods and hashing-based methods. The former leverage tree structures, consisting of nodes and edges, to store data points and search nearest neighbours efficiently. In contrast, hashing-based methods typically compute a low-dimensional hash value for given data points and search for nearest neighbours based on the proximity of hashes. Whether it is tree-based or hashing-based, both types of algorithms are known as ‘approximate methods’, and are only able to compute \textit{approximate} nearest neighbours (ANNs) of a given data point. Whereas the 1-NN and \textit{k}-NN methods return the true neighbours of a given data point, the approximate methods do not always consider the true nearest neighbours but rather data points in the neighbourhood (an approximation) of the true nearest neighbours. This way, by relaxing the constraint of searching for the true nearest-neighbour, the approximate methods now can return a sub-optimal solution while reducing the computation time. Depending on the algorithm
Chapter 2. Background

and properties of a given dataset, there are different theoretical bounds on how well the approximate methods perform ([Beygelzimer et al., 2006] [Gionis et al., 1999]).

In the next section we present the Boundary Trees and Forests algorithm ([Mathy et al., 2015]) that performs approximate nearest neighbour classification. This algorithm makes use of a tree structure to store data points, and performs distance comparisons to traverse down the tree when searching for approximate nearest neighbours. The boundary trees and forests algorithm plays a pivotal role in the work presented in this thesis. We selected boundary trees algorithm to work on as it offers fast search and, as we describe in subsequent sections, it can be easily extended to work with complex datasets such as images.

2.1.2 Boundary trees and forests

The boundary trees and forests proposed by [Mathy et al., 2015] elegantly overcomes the scalability issues of $k$-NN. It can be used for classification, regression and nearest neighbour retrieval problems. In our work, we only focus on the application of this algorithm to classification.

Boundary trees

When presented a large dataset, the boundary tree (BT) algorithm learns a small subset of data points that is able to represent adequately the whole dataset. The proposed scheme is built around a tree structure $T$, consisting of nodes and edges. Each node of $T$ uniquely represents some $(data point, label)$ pair. In order to perform classification, the tree is first trained considering $D$ in a sequential manner. In the beginning $T$ is empty and some random data point/label pair from $D$ is set as the root node of $T$. Since we have no prior knowledge about the order of the elements in $D$, without loss of generality, we assume $(x_1, y_1)$ is set as the root. Given any subsequent training pair $(x_i, y_i)$, $i \in \{2, \ldots, N\}$, $T$ is traversed starting from the root, searching for the node with the closest data point to $x_i$. At each step the data point of the current node and the data points of all its children are considered. If the closest node happens to be a child of the current node, the search is recursively repeated by setting the child to be the current node. Otherwise, the search ends and the current node is taken as the node in the tree that is (approximately) closest to $x_i$. Assuming the tree traversal returns a node pair $(x_j, y_j)$, then the training point $(x_i, y_i)$ is discarded if $y_i = y_j$ as this training point is already sufficiently well represented by $T$. On the other hand, if $y_i \neq y_j$ the training point is added to the tree as a child node of $(x_j, y_j)$.

We illustrate this training procedure in Figure 2.1.

Note that the training of boundary tree can be performed even when the training data is made available sequentially, making it an online learning algorithm. This way, the learned tree structure can be quickly modified to incorporate new data points. By construction, the boundary tree has the property that any two neighbouring nodes belong to different classes. The majority of the nodes in the tree will be near class boundaries, hence the name boundary tree.

After training, $T$ can be used for approximate nearest neighbour classification. In the BT algorithm, if we are given query point $x$, classification is performed by first searching $T$ for the (approximately) closest point to $x$. The same search scheme is used as during training. The data point returned by the search is selected as the approximate nearest neighbour of $x$, and its label is assigned to $x$ as the classification result. One can think of the BT algorithm as a combination of two concepts. The first is the selection of data points near class boundaries. The second is to make use of a tree structure that
Differentiable Boundary Trees

Figure 1. Boundary trees are built in an online fashion. Given a training data point, the tree is traversed in a greedy fashion, searching for the locally closest node. In this example the closest node belongs to a different class than that of the training point. Therefore the training point is added to the tree as a child of the closest node as shown in the rightmost image. The above illustrations are taken from the differentiable boundary trees paper by Zoran et al. (2017).

Figure 2.1: Illustration of the boundary tree training procedure.

In each image, the shaded and unshaded regions represent spaces of two classes. The leftmost image shows a trained boundary tree consisting of 7 data points that are on the correct sides of the boundary that separates the two classes. Given a training data point, marked by blue colour in the middle image, the tree is traversed in a greedy fashion, searching for the locally closest node. In this example the closest node belongs to a different class than that of the training point. Therefore the training point is added to the tree as a child of the closest node as shown in the rightmost image. The above illustrations are taken from the differentiable boundary trees paper by Zoran et al. (2017).

enables efficient search for the approximate nearest neighbour of a given data point. In the BT algorithm described above, there is no limit on the number of children that a parent node can possess. This allows the tree to grow horizontally without restraint. This might lead to undesirable properties of the tree since the number of distance calculations that are needed at a certain node would then be unbounded. In mitigating this problem, the authors of the BT algorithm also propose a mechanism to restrict the number of children at any node. In the modified algorithm, if a certain parent node has reached the maximum number of children allowed, the closest node to a training point is searched by considering only the child nodes, leaving out the parent node. This way the algorithm cannot stop at a node that already has reached the limit on children. In the experiments presented in the original paper by Zoran et al. (2017), the maximum children limit at any node is set to 50. With this slight modification the query and training time of the resulting tree scale as $\log(N)$ and $N \log(N)$ respectively.

Boundary forests

It is important to note that the boundary tree created above is dependent on the order of the training data points. The data points included in the tree at the beginning have a strong say in whether a new training data point must be added to the tree or not. Mathy et al. (2015) propose boundary forests to mitigate this effect. A boundary forest (BF) is an ensemble of boundary trees trained on different permutations of a given dataset. Since a boundary tree heavily depends on the data points used at the beginning of the training, a forest generalizes better by randomizing the order of the training data points. In case of the online setting, when not all the training data is available to generate permutations, the ensemble of boundary trees is created by starting each tree at a different position of the training data sequence. Classification with a boundary forest is performed by combining the predicted label of each tree according to some weighting scheme. In the experiments presented by Zoran et al. (2017), around 50 trees is used to create a forest. In our work we focus on boundary trees, but our results pertain to boundary forests as well.
2.1.3 Other related work

Nearest neighbour methods have been known for a long time and thus many different variants have been studied. The computational complexity of the neighbour search being one of the biggest challenges in this context, many methods to overcome this issue have been proposed. As noted before in Section 2.1.1, these methods can be broadly categorized into tree-based methods and hashing-based methods. Tree-based methods can be further divided into two sub-categories. Algorithms in the first sub-category recursively partition the input space using hyperplanes. Random decision forests (Breiman, 2001) and k-d trees (Friedman et al., 1977) are two popular examples of this class. Algorithms in the second sub-category rely on distance comparisons to traverse down the tree structure and the examples include the boundary trees (Mathy et al., 2015), cover trees (Beygelzimer et al., 2006) and ball trees (Liu et al., 2006). Generally tree-based methods often result in a logarithmic search time, drastically reducing the cost of neighbour search in the massive datasets typical of contemporary applications.

In contrast, hashing-based methods like those presented by Andoni and Razenshteyn (2015) and Salakhutdinov and Hinton (2009) better suit high-dimensional data. This family of methods typically computes a low-dimensional hash value for given data points and search for nearest neighbours based on the proximity of hashes. The hash values are computed using a hashing function that preserves some distance measure between original data points. For example, two points that are close in the original space in the Euclidean sense may get two hash values that are close in terms of the Hamming distance.

Distance metrics for complex data

In the boundary tree algorithm described above, Euclidean distance was assumed. That is, given two data points $x_i$ and $x_j$, the similarity between them equals the distance $d(x_i, x_j) = \|x_i - x_j\|$. The Euclidean norm is a popular choice as a distance measure, mainly due to its appealing mathematical properties. One of the important properties in this context is that the Euclidean norm has a smooth derivative (except at the origin). Another property is that the Euclidean norm can be expressed as an inner product. Specifically, $\|x_i - x_j\|^2 = \langle x_i - x_j, x_i - x_j \rangle$ where the right hand side expresses the inner product of the vector $x_i - x_j$ with itself.

While acceptable in certain datasets, Euclidean distance may not be a good choice for complex data such as images. As is proposed by Mathy et al. (2015), this limitation can be mitigated by first transforming the input data using a known feature-extraction function $f(\cdot)$ and, subsequently, measuring Euclidean distance between extracted features. In this case, the distance between $x_i$ and $x_j$ is $d(x_i, x_j) = \|f(x_i) - f(x_j)\|$. One drawback of this proposal is that $f(\cdot)$ needs to be carefully crafted, crafting that might require expert, domain-specific knowledge. An alternative is to learn some transformation function $f_\theta(\cdot)$, parameterized by $\theta$, such that Euclidean distance is a good distance metric in the transformed domain. When building the boundary tree, the distance between $x_i$ and $x_j$ can then be assumed the Euclidean distance in feature domain, $d(x_i, x_j) = \|f_\theta(x_i) - f_\theta(x_j)\|$. In Section 2.2 we describe approaches that have been proposed for the purpose of learning such a transformation function. Since these functions learn to represent data in a transformed domain, the task is often referred to as ‘representation learning’.
2.2 Representation learning

In this section we present three algorithms that use multi-layer neural networks to implement a transformation function $f_{\theta}(\cdot)$. The neural networks are considered universal function approximators. Theoretically, if provided with enough model complexity, a neural network is able to approximate any continuous function on compact subsets (i.e., the domain is a set of closed and bounded intervals) in finite dimensional space \cite{Hornik1989}. Optimization of $f_{\theta}(\cdot)$, implemented as a neural network, is generally performed with gradient descent algorithm (or a variant) by minimizing an appropriate objective function, known as a ‘loss’ function. The first two classes of neural networks, Siamese networks \cite{Koch2015} and triplet networks \cite{Hoffer2015}, are generic algorithms used for the purpose of learning a similarity measure for data. Functions learned with these networks are often used for image verification and retrieval applications. These networks have proven to be very effective even when very small amount of training data is available, giving rise to the name ‘one-shot learning’. The third algorithm presented is differentiable boundary trees \cite{Zoran2017}. The work presented in this thesis is motivated by the differentiable boundary tree algorithm which we discuss extensively in the next sections.

2.2.1 Multi-layer neural networks

We start-off with a brief introduction to the architecture of the neural networks used in this work. A multi-layer feed-forward neural network comprising of $L$-layers can be concisely described as follows. Given an input vector $x^{(0)} \in \mathbb{R}^{D^{(0)}}$, the $\ell$-th ($1 \leq \ell \leq L$) layer output $x^{(\ell)} \in \mathbb{R}^{D^{(\ell)}}$ is defined as

$$x^{(\ell)} = \Phi^{(\ell)}(W^{(\ell)}x^{(\ell-1)} + b^{(\ell)}). \quad (2.1)$$

Here, $W^{(\ell)}$ is a learnable parameter matrix of dimension $D^{(\ell)} \times D^{(\ell-1)}$ and $b$ is a $D^{(\ell)}$-dimensional learnable parameter vector. In this context, learnable means that exact values of the parameters are determined through some optimization scheme. In general, for the neural networks designed for the classification task, the dimensions of layers are selected such that $D^{(0)} > D^{(1)} > \cdots > D^{(L)}$. The function $\Phi^{(\ell)} : \mathbb{R}^{D^{(\ell)}} \to \mathbb{R}^{D^{(\ell)}}$ is referred to as the activation function of $\ell$-th layer. The most popular choice is the rectifier function defined as $\text{relu}(\cdot) = \max(0, \cdot)$, where the $\max(\cdot, \cdot)$ operator selects the maximum of the two arguments. This function is extended to vector inputs simply by applying $\text{relu}(\cdot)$ element-wise. Unless otherwise specified, in all neural networks considered in the following sections, we assume $\Phi^{(L)}$ is the identity function and $\Phi^{(\ell)}$ for $1 \leq \ell < L$ are some non-linear activation functions. For example, in the case of vanilla neural network (VNet) classifiers, $\Phi^{(L)}$ is considered the softmax activation function, whose $i$-th component of the output vector is given by $\exp(h(i))/\sum_{j \in [D^{(L)}]} \exp(h(j))$, where $h(i)$ is the $i$-th component of some input vector $h \in \mathbb{R}^{D^{(L)}}$. Since we use the identity function as the last layer activation, the domain and range of the multi-layer neural networks are $\mathbb{R}^{D^{(0)}}$ and $\mathbb{R}^{D^{(L)}}$, respectively. In the next sections, we avoid specifying details of the neural networks considered and define them by specifying only the domain and range.

2.2.2 Siamese networks

Siamese networks were proposed by \cite{Koch2015}. They identify if any given pair of data points are coming from the same or from different classes. The objective of the Siamese architecture is not to classify input data, but to differentiate between them. One could also argue that the Siamese networks
are in fact binary classifiers that operate on pairs of inputs. The classifier generates a binary output depending on whether the two inputs share the same class or not. In the simplest variant of the Siamese networks, a function $f_\theta(\cdot)$ is modeled by a multi-layer neural network using the learnable parameter set $\theta$. The function takes in raw data points at its input and outputs a vector of a smaller dimension, that is large enough to encode the high level features of the input. Consistent with the neural networks described in Section 2.2.1, the defined function takes the form $f_\theta : \mathbb{R}^D \rightarrow \mathbb{R}^m$ for some dimension $m(<D)$.

The parameter set $\theta$ is initialized randomly and optimized over the samples in dataset $D$ by minimizing the loss function we formulate next. Given two examples $x_i$ and $x_j$ from $D$, the network is trained to make $\|f_\theta(x_i) - f_\theta(x_j)\|$ small if $y_i = y_j$, and large otherwise. Such a strategy will force $f_\theta(\cdot)$ to learn to place transformations of inputs that share the same class closer to each other, minimizing the distance in the transformed domain. The opposite is true for inputs that are of different classes. The name ‘Siamese’ networks is due to the fact that both $x_i$ and $x_j$ go through the same transformation. How Siamese networks can be used for image classification is illustrated in Figure 2.2.

**Contrastive loss**

Motivated by the described training strategy, one could suggest to optimize the parameter set $\theta$ in Siamese networks by minimizing the loss

$$
\mathcal{L}_{\text{sme}} = \mathbb{1}[y_i = y_j]d(x_i, x_j) - \mathbb{1}[y_i \neq y_j]d(x_i, x_j),
$$

where $\mathbb{1}[\cdot]$ denotes the indicator function. When $y_i = y_j$, the loss function simplifies to $d(x_i, x_j)$ and minimizing $\mathcal{L}_{\text{sme}}$ forces $d(x_i, x_j)$ to be close to zero. Similarly when $y_i \neq y_j$, minimizing $\mathcal{L}_{\text{sme}}$ forces $d(x_i, x_j)$ to be large, which is the desired behaviour. However, in the latter case $d(x_i, x_j)$ is unbounded from above and $\mathcal{L}_{\text{sme}}$ can be arbitrarily made small by making $d(x_i, x_j)$ large enough, i.e., $\mathcal{L}_{\text{sme}}$ will not converge. This means that the data points sharing the same label will be forced into clusters while the distance between such clusters bearing different labels will keep growing. To mitigate this problem the authors of the Siamese network paper suggest to use what is known as the contrastive loss, defined as

$$
\mathcal{L}_{\text{cnt}} = \mathbb{1}[y_i = y_j]d^2(x_i, x_j) + \mathbb{1}[y_i \neq y_j] \max(0, \alpha - d^2(x_i, x_j)),
$$

where $\alpha$ is a strictly positive hyperparameter, i.e., a handpicked model parameter that cannot be learned directly. The best or a good enough hyperparameter is selected by trial and error. The proposed loss...
function encompasses the desired training strategy while imposing a lower bound on the loss when \( y_i \neq y_j \). In this case, since \( L_{\text{cnt}} \) is zero for \( d(x_i, x_j) \geq \alpha \), there is no incentive to increase \( d(x_i, x_j) \) further, and as a result the loss converges. The use of squared Euclidean distance in (2.2) is purely for the numerical stability of the algorithm when implemented with finite precision. Minimizing \( L \) further, and as a result the loss converges. The use of squared Euclidean distance in (2.2) is purely for the desired characteristics of the loss function.

The expression for \( \nabla_{\theta} L \) becomes numerically unstable when \( d(x_i, x_j) \) reaches zero, due to the vanishing terms in both numerator and denominator. While there are numerical methods for mitigating such complications, this can be easily avoided by using squared Euclidean distance without harming the desired characteristics of the loss function.

2.2.3 Triplet networks

Triplet networks proposed by Hoffer and Ailon (2015) are a variant of the Siamese architecture. Triplet networks learn a transformation function \( f_\theta(\cdot) \), but use a slightly different training scheme than that described above. Given a triplet of data points \((x_i, x_j, x_\ell)\) with the property that \( y_i = y_j \neq y_\ell \), a triplet network is trained to learn the function \( f_\theta(\cdot) \) such that the inequality

\[
\beta + d^2(x_i, x_j) < d^2(x_i, x_\ell)
\]  

(2.4)

is satisfied for a positive hyperparameter \( \beta \). The same as in before, \( d(x_i, x_j) \) means the distance between the \( f_\theta(\cdot) \)-transformed versions of \( x_i \) and \( x_j \). The name “triplet network” is attributable to the use of three examples for the training. The inequality above guarantees that in the transformed domain, a data point is closer to another data point of the same class than to one from a different class. The hyperparameter \( \beta \) controls the margin that is allowed between the two types of distances.

**Triplet loss**

Triplet loss has been proposed to optimize the parameter set \( \theta \). Minimizing triplet loss, defined as

\[
L_{\text{trp}} = \max(0, \beta + d^2(x_i, x_j) - d^2(x_i, x_\ell)),
\]

enforces the inequality in (2.4). For the same reason as in Siamese networks, the \( \max(0, \cdot) \) operation is used to guarantee that the \( L_{\text{trp}} \) is bounded from below, while incentivizing the separation of \( d^2(x_i, x_j) \) and \( d^2(x_i, x_\ell) \) by a margin of \( \beta \).

Sampling triplets to train the network can be somewhat complicated. Out of all triplets that are possible in a dataset, only the valid triplets, i.e., three data points belonging to exactly two classes, have to be selected. For most of these valid triplets the inequality in (2.4) is trivially satisfied. This is because given a data point, the number of other data points in the dataset that share its class is many fewer than
the number of points that have a different class. As a result, many possible triplets will yield a zero loss and will not contribute to the training. The convergence rate of the loss can be improved by carefully selecting the triplets as suggested by [Schroff et al., 2015] and [Hermans et al., 2017]. Instead of training over all valid triplets, they suggest only to use what are termed ‘hard triplets’. Those on the triplets most violate the condition in (2.4). Given a training data point $x_i$, the optimal parameter set for the transformation function can be selected by first maximizing $L_{\text{trp}}$ over $j$ and $\ell$, then minimizing over $\theta$. However, selecting ‘too hard’ triplets might make the training unstable and lead to poor generalization. This is because the new approach is susceptible to outliers in training data, since outliers are most likely to dominate the hard triplets. The literature suggests that it is unclear what defines ‘good’ hard triplets, and many variants of the algorithm have been proposed (Hermans et al., 2017). Also, the search of hard triplets is an expensive operation involving many redundant distance computations.

**Boundary trees with Siamese and triplet networks**

The problem of finding good combinations of data points for training is common to the Siamese networks as well. But the effect of careless example selection is less severe when choosing ‘pairs’ for Siamese networks than when choosing triplets. In general, Siamese networks have proven to be stable and consistent in learning data representations. But if provided with the right set of training examples along with carefully tuned hyperparameters, the triplet networks and its variants often render better performance (Hermans et al., 2017). After learning a feature extraction function $f_\theta(\cdot)$ using any of above representation learning methods, building a boundary tree or a forest is trivial. The same approach described in Section 2.1.2 is used with $d(x_i, x_j) = \|f_\theta(x_i) - f_\theta(x_j)\|$ as the distance measure between $x_i$ and $x_j$.

Even though Siamese and triplet networks have proven to be good representation learning algorithms, they are geared toward discriminating data to predict whether the data are coming from the same class or not. The differentiable boundary tree algorithm described next is developed specifically for the use of boundary trees and forests, keeping in mind their applications in classification.

### 2.2.4 Differentiable boundary trees

The differentiable boundary tree (DBT) algorithm proposed by [Zoran et al., 2017] is different from the previously discussed algorithms in few important aspects. The DBT method directly involves a boundary tree in its training process whereas in previous cases learning a transformation function and building the boundary tree were two disjoint operations. Although good data representations imply better classification results, the Siamese and triplet networks were not directly optimized for classification application. The DBT scheme jointly optimizes a transformation function $f_\theta(\cdot)$, parameterized by $\theta$, such that boundary trees built with $f_\theta(\cdot)$ yield good classification results. The function $f_\theta(\cdot)$ in DBT is implemented by a multi-layer neural network and the dimension of the output (the range of $f_\theta(\cdot)$) is kept small relative to that of the input. The parameter set $\theta$ is initialized randomly and is jointly optimized with a boundary tree using gradient descent as per the scheme we describe next.

Given training dataset $\mathcal{D}$, let $\tilde{\mathcal{D}}$ be a subset (a mini-batch) of size $N_b+1 (\ll N)$ and $\mathcal{U} = \{(f_\theta(x_i), y_i) \mid (x_i, y_i) \in \tilde{\mathcal{D})}$. First, we build a boundary tree $\mathcal{T}$, using the first $N_b$ elements of $\mathcal{U}$. We use node-$i$ to indicate the node in $\mathcal{T}$ that consists of the pair $(f_\theta(x_i), y_i)$. Also, let $p(i)$ be the parent node index of node-$i$, and $W(i)$ be the index set of the siblings of node-$i$. In the second step, we use the last
Figure 2.3: Visualizing the probabilistic model asserted for differentiable boundary tree transitions.

Portrayed is a partially built boundary tree, along with its member nodes depicted in blue. Given a training data point, shown in green, the tree is traversed starting from the root node searching for the closest data point in the tree. The traversal path for this example is shown in red colour and final* is the closest example in the tree to the training data point. According to the proposed model, the probability of transition 1 is calculated by considering the root node neighbourhood, consisting of the root node and its three children. Similarly, probabilities of transition 2 and 3 are calculated considering the respective neighbourhoods, marked by circles with dash lines. The first term in (2.6) gives the log joint probability of transitions from 1 to 3, and the second term is obtained by separately considering the final node neighbourhood. The figure is taken from the DBT paper by Zoran et al. 2017.

In our experiments, presented in Chapter 4, we replace $d(\cdot, \cdot)$ in (2.5) by $d(\cdot, \cdot)/\sigma$, allowing the positive scalar value $\sigma$ to control the sensitivity of the distance measure. A soft class prediction on $x_r$ is obtained by separately considering the final transition. By letting node-s be the last node in the tree traversal,
the (unnormalized) log probability of $x_r$ belonging to class $c \in [C]$ is

$$\log \text{Pr}^*(y_r = c|x_r) = \left[ \sum_{i \in V \setminus s} \log \text{Pr}(p(i) \to i|r) \right] + \log \left[ \sum_{i \in V(s) \cup \{s\}} \text{Pr}(p(i) \to i|r) 1[y_i = c] \right],$$

(2.6)

where $1[\cdot]$ denotes the indicator function. The computed class probabilities are normalized to obtain a proper distribution, which produces

$$\text{Pr}(y_r = c|x_r) = \frac{\text{Pr}^*(y_r = c|x_r)}{\sum_{c' \in [C]} \text{Pr}^*(y_r = c'|x_r)}.$$  

(2.7)

The described process of tree traversal is visualized in Figure 2.3. Note that the first term in (2.6) is not dependent on $c$ and it is common for all $\log \text{Pr}^*(y_r = c|x_r)$ where $c \in [C]$. These unnormalized log probabilities are converted to probability values by first exponentiating and then normalizing as per (2.7). The exponentiated first term in (2.6) becomes a common factor in the denominator as well as the numerator in (2.7). Though the authors of the differentiable boundary tree algorithm have not noted the following, we observe that the contribution from the first term in (2.6) completely disappears after normalizing, nullifying the effect of this term on the rest of the computations. The significance of this observation will become clear when we motivate our own algorithmic development in Section 3.1.

After obtaining soft class predictions $\text{Pr}(y_r = c|x_r)$, the cross-entropy loss $L_{\text{dbt}}$ is calculated using the class prediction of $x_r$ and its true label $y_r$ according to the formula

$$L_{\text{dbt}} = -\sum_{c \in [C]} 1[y_r = c] \log \text{Pr}(y_r = c|x_r).$$

(2.8)

Finally, the gradient of loss $\nabla_\theta L_{\text{dbt}}$ is computed and one gradient step is taken towards minimizing $L_{\text{dbt}}$. The process of building $T$ and minimizing $L_{\text{dbt}}$ is repeated until the loss converges, using a new mini-batch $\bar{D}$ each time. As per the described algorithm, computing $L_{\text{dbt}}$ involves a tree traversal, which is not a differentiable operation. This means $\nabla_\theta L_{\text{dbt}}$ cannot be computed in closed-form. Alternatively, the DBT algorithm proposes to alternate between building a tree and computing $\nabla_\theta L_{\text{dbt}}$ on the newly built tree. The proposed scheme effectively computes gradients through a boundary tree, hence the name differentiable boundary trees.

The convergence rate of the loss can be increased by considering a mini-batch of size $N_b + N_t\ (\ll N)$ instead of $N_b + 1$. In each iteration, the first $N_b$ elements are used to build $T$. The rest of the elements are then used to take $N_t$ sequential gradient steps towards minimizing $L_{\text{dbt}}$. Minimizing $L_{\text{dbt}}$ causes the data points sharing a class to form clusters, while separating further such clusters of different classes. At this point it is a fair question to ask if $L_{\text{dbt}}$ suffers from the same issue of $L_{\text{sme}}$ in Section 2.2.2, i.e., whether $L_{\text{dbt}}$ can be minimized without restraint. Such as scenario is not observable with $L_{\text{dbt}}$, since, by definition, cross-entropy is lower bounded at zero. However, depending on the model complexity it can be made arbitrarily close to zero.

At the test time, a final boundary tree is built using all available training data, transformed through learned $f_\theta(\cdot)$. Given a test data point $x$, a classification label is assigned to $x$ by first taking transformation $f_\theta(x)$ and subsequently searching the final boundary tree for the approximate nearest neighbours. In the next sections, we refer this final classifier as the ‘DBT classifier’.
Comparing with other work

In terms of its positioning among the other algorithms, the differentiable boundary tree scheme is most closely related to the tree-based methods. We note that it learns a function that transforms high-dimensional data to a low-dimensional space, and thus make a connection to hashing-based methods as well. What makes the DBT algorithm unique among the existing representation learning methods is that it learns a function that is specifically trained to build boundary trees. In contrast, learning the representations and building the boundary tree are performed separately in Siamese or triplet networks. This means Siamese and triplet networks do not learn to appreciate fully the importance of points near class boundaries, points that are the most important in building boundary trees. We further discuss the significance of this concern in Section 3.1.

2.3 Adversarial examples

With the recent advances in machine learning, neural networks have been employed in many different applications such as image classification and speech recognition. However, recent studies have shown that machine learning models are often vulnerable to adversarial manipulation of their inputs which can easily cause incorrect classification (Szegedy et al., 2013). Generating such manipulations, often referred to as adversarial examples, involves an adversary perturbing a valid input from a dataset. Even though the perturbations maybe unnoticeable to a human or may not alter the semantic meaning of the example, the system can assign a wrong label to the perturbed version, often with high confidence. There are various ways of generating adversarial examples, known as adversarial attacks. Of course, many methods have also been proposed to make classifiers robust and to defend against such attacks. But a defense that is successful against a particular type of adversarial attack is often vulnerable to other types of attacks. The adversarial examples are somewhat connected to the broader field of sensitivity analysis. As the name suggests, sensitivity analysis studies the robustness of the output of a mathematical model with respect to variations of its input. The idea behind adversarial examples is very similar to that of sensitivity analysis. Although these methods have been known for years, they have become more important now than ever before.

The fact that an adversary can be able to change completely a model prediction by performing unnoticeable changes to the input is especially problematic in sensitive areas such as healthcare and security-related applications. To date, there has been no widely accepted solution that sufficiently addresses the issue of adversarial examples. In following subsections we briefly review an adversarial attack and a defense. In Section 4.2, we present experimental results and compare how well the vanilla neural network classifier and nearest neighbour methods perform in presence of adversarial examples. The superior performance of neighbour-based methods is one of the key motivations for our own algorithm development discussed in Section 3.1.

2.3.1 Generating adversarial examples

Given a data point \( z \in \mathcal{X} \) and its associated label \( t \in [C] \), we assume a certain (trained) classification model assigns \( z \) to class \( c \in [C] \) with probability \( \Pr(t = c|z) \). For this example we remind the reader
that the cross entropy loss is defined as
\[ L_{ce}(z, t) = -\sum_{c\in[C]} I[t = c] \log Pr(t = c|z). \]

One intuitive method of generating an adversarial example for \( z \) is to perturb \( z \) in a way that the perturbed version incurs in a large classification loss. Consider the constrained optimization problem
\[
\max_{\tilde{z} \in B_z} L_{ce}(\tilde{z}, t), \tag{2.9}
\]
where \( B_z \subset \mathbb{R}^D \). Note that \( D \) is the dimension of the elements in \( X \). By choosing \( B_z \) to be the set of data points that are closer to \( z \) according to some appropriate measure, we can ensure that the difference between \( z \) and \( \tilde{z} \) is unnoticeable to a human observer. However, \( L_{ce}(\tilde{z}, t) \) is not necessarily convex in its arguments and we cannot use readily available convex optimization methods to solve (2.9).

Goodfellow et al. (2014) propose Fast Gradient Sign (FGS) method, a simple yet efficient method of approximating the solution of (2.9). The FGS method is designed for image datasets where pixel values are vectorized and serve as the classifier input. For a given input image, this method determines a perturbation that maximizes a first-order approximation of the classification loss, over an input set \( B_\epsilon(z) = \{ \tilde{z} | \epsilon \geq |\tilde{z}(i) - z(i)| \forall i \in [D] \} \). In other words the set \( B_\epsilon(z) \) is the \( L_\infty \) norm ball with radius \( \epsilon \), centered at \( z \). By constraining the input set to be \( B_\epsilon(z) \), the FGS method ensures that the deviation of any pixel from its original value is confined in the \([-\epsilon, \epsilon]\) interval. The FGS method considers the first-order Taylor approximation of the loss function at the data point \( z \), which can be expressed as
\[ L_{ce}(\tilde{z}, t) \approx L_{ce}(z, t) + (\tilde{z} - z)^T \nabla_z L_{ce}(z, t) \]
\[ \leq L_{ce}(z, t) + \epsilon \|\nabla_z L_{ce}(z, t)\|_1. \]

The approximation is upper bounded since \( \tilde{z} \in B_\epsilon(z) \), and the equality is achieved by selecting \( \tilde{z} = z_{ig} \) where
\[ z_{ig} = z + \epsilon \text{sgn}(\nabla_z L_{ce}(z, t)). \tag{2.10} \]

Here, \( \text{sgn}(\cdot) \) denotes the sign function. Having a smaller \( \epsilon \) ensures that \( z_{ig} \) is visually similar to \( z \), therefore a human would classify both of them as originating from the same class. In our notation, \( z_{ig} \) stands for ignorant adversary. The reason behind this naming will become clear in next section wherein we discuss adversarial training. In Figure 2.4 we illustrate a few different adversarial images generated with fast gradient sign method. In the next section we describe how to train classifiers to be more robust in presence of adversarial examples.

### 2.3.2 Adversarial training

The classical method of training a classifier involves minimizing the sum of the \( L_{ce}(z, t) \) values of all \( z, t \) pairs in the dataset. To make classifiers robust to adversarial inputs, Szegedy et al. (2013) have proposed to consider adversarial examples as augmented training data and to train on a mixture of clean and adversarial examples. However this method has been proven to be inefficient due to the practical limitations of neural network training tools. An alternative efficient method proposed by Goodfellow et al. (2014) uses the fast gradient sign method as the adversarial example generating mechanism. The
Chapter 2. Background

Figure 2.4: Adversarial examples generated using fast gradient sign method. The figure is taken from the experiment results performed by Kurakin et al. (2016). The clean image in the top-left corner is used to generate rest of the adversarial examples using different $\epsilon$ values. The clean image is a 3-channel RGB image and the pixel values in each channel range from 0 to 255. Note that the clean image and the image for $\epsilon = 4$ images are nearly indistinguishable to the naked eye. The classifier correctly recognizes the clean image as a ‘washer’, while rest of the images are classified into incorrect classes.

The proposed training procedure involves minimizing a new loss function

$$L^*_\text{adv}(z, t) = \lambda L_{ce}(z, t) + (1 - \lambda) L_{ce}(z_{ig}, t),$$

(2.11)

where the hyperparameter $\lambda$ is typically set to 0.5 to give equal importance to both clean and adversarial training data points. In our discussion below, we assume $\lambda = 0.5$ and without loss of generality consider a substitute loss function

$$L_{\text{adv}}(z, t) = 2L^*_\text{adv}(z, t) = L_{ce}(z, t) + L_{ce}(z_{ig}, t)$$

(2.12)

to avoid clutter in the expressions. Adversarial training is performed on the classifier by minimizing $L_{\text{adv}}(z, t)$ over all $z, t$ pairs in the dataset $\mathcal{D}$.

In the testing phase, we assume that there are two types of adversaries that can be used to generate adversarial examples using test examples, namely Ignorant Adversary and Cognizant Adversary. The ignorant adversary ignores the fact that a model has been trained to minimize $L_{\text{adv}}$, i.e., to correctly classify adversarial examples in addition to original training data. Its objective as an adversary would be limited to generating adversarial examples according to the original formula in (2.10). But a classifier that received adversarial training will correctly classify most of $z_{ig}$ examples generated at the test time, producing a lower error rate on the adversarial as well as the clean examples. In contrast, the cognizant adversary is aware that the network is trained to minimize $L_{\text{adv}}$ in (2.12), and is careful to take into account the composite loss function when generating adversarial perturbations. Adversarial examples
generated by the cognizant adversary take the form

\[ z_{\text{co}} = z + \epsilon \sgn(\nabla_z \mathcal{L}_{\text{adv}}(z, t)) \]
\[ = z + \epsilon \sgn(\nabla_z \mathcal{L}_{\text{ce}}(z, t) + \nabla_z \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t)). \]

By using the property that

\[ \nabla_z \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t) = \nabla_{z_{\text{ig}}} \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t) \odot \nabla_z z_{\text{ig}} = \nabla_{z_{\text{ig}}} \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t) \odot 1, \]

where we use \( \odot \) to denote element-wise multiplication. We can simplify \( z_{\text{co}} \) as

\[ z_{\text{co}} = z + \epsilon \sgn \left( \nabla_z \mathcal{L}_{\text{ce}}(z, t) + \nabla_{z_{\text{ig}}} \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t) \right). \quad (2.13) \]

The interpretation of (2.13) is intuitive. In (2.10), the ignorant adversary considers the gradient computed only with respect to original data point \( z \). But in (2.13), the cognizant adversary considers gradient computed with respect to the adversarial example of ignorant adversary, \( z_{\text{ig}} \), as well. The extra term \( \nabla_{z_{\text{ig}}} \mathcal{L}_{\text{ce}}(z_{\text{ig}}, t) \) in (2.13) non-linearly contributes towards generating \( z_{\text{co}} \). In experiments performed in Section 4.2, we consider both these types of adversaries and, not surprisingly, the cognizant adversary proves better in attacking classifiers trained in an adversarial manner.
Chapter 3

Methodology

In this chapter we present our main contribution, Differentiable Boundary Sets. The differentiable boundary sets is an algorithm that overcomes some of the practical issues of differentiable boundary trees discussed in Section 3.1. We empirically validate the claims of the proposed model by performing experiments that we present in Chapter 4. The key contributions provided by the boundary tree (BT) and differentiable boundary tree (DBT) algorithms can be summarized as follows. The BT algorithm offers fast search for approximate nearest neighbours among a large number of high dimensional data points. Applications include classification, regression and data retrieval. The DBT algorithm learns a function $f_\theta(\cdot)$ that transforms high dimensional data to a space where Euclidean distance is a good measure of similarity with respect to labels. In this section we first make a number of observations about BTs and the DBT schemes we presented in Sections 2.1.2 and 2.2.4 which motivate our new algorithm differentiable boundary sets. We then describe the proposed algorithm.

3.1 Motivation

With regard to BT and DBT algorithms we make following observations and deductions that motivate us to alter the DBT scheme.

(i) As mentioned in Section 2.2.4, the contribution from the first term in (2.6) disappears after normalization and the term has no effect on the loss $L_{dbt}$. This suggests that only a small set of siblings of the ANN of a given training data point actually contribute to learning $f_\theta(\cdot)$. Most of the data points in the tree, which otherwise could have helped the learning process, are completely ignored.

(ii) The number of nodes in $\mathcal{T}$ is upper bounded by $N_b$, the number of examples used to train the tree. In the experiments performed by Zoran et al. (2017) on MNIST and CIFAR10 datasets, $N_b$ is set to 1000 and 100 respectively, which are relatively small compared to the dataset sizes.

(iii) The transformation $f_\theta(\cdot)$ throws away a lot of information that is unrelated to labels and simplifies class boundaries. As a direct result, often only a small number of data points need to be included in the tree. This can be observed in the experiments performed by Zoran et al. (2017) where the boundary tree learned for MNIST dataset is represented by only 25 samples and that for the CIFAR10 dataset by only 22 samples.
Figure 3.1: Illustrating the importance of selecting points near class boundaries when learning $f_\theta(\cdot)$.

Each sub-figure portrays the same underlying problem with a boundary between two classes red and green. A few data points, marked by empty circles, lie on the correct sides of the boundary. In this example, we will ignore the tree structure for clarity, but the conclusions pertain to BTs. Given the query data point $\mathbf{x}$, marked by the circled cross, we consider its 4-NNs (with no regard to the boundary) as illustrated in the left-side figure. In this case, the class prediction already has a high confidence that $\mathbf{x}$ belongs to green class since it is closer to and is surrounded by points in said class. Such a prediction offers little incentive to separate further the two classes. The figure in right-side demonstrates the case when the 4-NNs that are also near the boundary are considered. In this case, class prediction has a low confidence that $\mathbf{x}$ belongs to green class, which incentivizes $f_\theta(\cdot)$ to learn to separate the two classes and to pull the green points away from the boundary.

(iv) Observations [iii] and [iii] indicate that, while training the DBT, the size of the tree is within our control, and often falls below 30 even for real-world datasets. Since the true benefit of the tree structure, which is efficient search, emerges only when the stored number of samples is large, there is then little reason to believe that such tree structure has a significant impact on the training time of DBT algorithm.

(v) The original motivation to develop the DBT algorithm is to learn a feature-extraction function $f_\theta(\cdot)$ such that $d(\mathbf{x}_i, \mathbf{x}_j) = ||f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)||$ is a good measure of similarity between the labels of $\mathbf{x}_i$ and $\mathbf{x}_j$. Subsequently, the learned function can be used to transform the given dataset into the new feature space to build a BT or BF for the intended application. To this end, any method that efficiently learns an $f_\theta(\cdot)$ with the desirable properties is applicable. One such desirable property is appreciation of the importance of points near class boundaries. Therefore, we argue that there is no requirement to involve a tree structure in the training phase.

(vi) The observation made in (v) provides an inkling that one may not need to use a tree structure in the DBT algorithm, rather using data points near class boundaries is what has a significant impact on learning $f_\theta(\cdot)$. This is because points near a class boundary are the furthest from the cluster center, and closest to the boundary of another class. Given a random query data point $\mathbf{x}$, its label $y$ can be predicted by either considering the ANNs of $\mathbf{x}$, or considering the ANNs of $\mathbf{x}$ that are also near the class boundary. With regard to the latter, the predicted label is most likely to be weak or erroneous. Consequently, $f_\theta(\cdot)$ is forced to make such weak predictions stronger by learning clusters whose class boundaries are well separated. This can be thought of as learning in an adversarial setting, where the provided neighbour points are carefully selected by an adversary to be the poorest choice among the ANNs. Thus making the learned classifier more robust, as we illustrate in Figure 3.1.

(vii) State-of-the-art neural network learning tools such as GPUs and TPUs [Jouppi et al., 2017] use batch-wise optimization to scale to large datasets. As pointed out by [Zoran et al., 2017], many practical issues arise when a function $f_\theta(\cdot)$ is optimized in conjunction with a tree, mainly because
the depth of tree traversal and the number of elements in the final node neighbourhood (as depicted in Figure 2.3) differ from one training point to another. This causes the batch-implementation of DBT to be inefficient and prevents the algorithm from leveraging the benefits offered by modern tools that work best with batch-implementation.

Based on the above observations, we argue that the difficulties due to the joint optimization of \( f_\theta(\cdot) \) with a tree structure outweigh the benefits. We stress the necessity to revamp the DBT algorithm or to look for an alternative, that not only serves its original purpose but also is efficient and conforms to modern machine learning techniques. To overcome the difficulties inherent to the DBTs, we propose the boundary set and differentiable boundary set algorithms which we present next.

### 3.2 Boundary sets

The boundary set (BS) is, in effect, a modified version of a boundary tree, including modifications geared toward efficiently learning a transformation function \( f_\theta(\cdot) \). The pivotal change in a boundary set is that it accumulates samples in a set \( S \) without building a tree structure. Given the training dataset \( D \), the BS considers data points sequentially, similar to BT. In the beginning \( S \) is empty and \((x_1, y_1)\) is added as its first element. Given any subsequent training pair \((x_i, y_i), i \in \{2, \ldots, N\}\), elements of \( S \) are searched to obtain the pair closest to \( x_i \), according to the distance function \( d(x_i, x_j) \). Assuming the search returns pair \((x_j, y_j)\), the training point \((x_i, y_i)\) is discarded if \( y_i = y_j \), otherwise \((x_i, y_i)\) is added to \( S \). Clearly for the boundary sets to be useful the final set cannot be of too large a cardinality, else we would suffer from the same effects that first motivated the development of boundary trees.

The proposal to accumulate data in a boundary set instead of a boundary tree is based purely on practical considerations. Our goal is to make the training procedure more conforming with modern tools such as GPUs and TPUs. These tools are optimized for operations performed on data structures such as matrices. As a result, operations like nearest neighbour search can be efficiently performed on such data structures. The boundary set, being a collection of vectors, is implementable as a 2-dimensional array and enables batch-wise nearest neighbour search. The boundary tree structure on the other hand is hard to implement to support batch-wise operations. Therefore, the boundary set is more practical to use for training of the transformation function \( f_\theta(\cdot) \).

### 3.3 Differentiable boundary set algorithm

We propose the differentiable boundary set (DBS) algorithm which addresses issues of DBT and improves its accuracy, convergence rate, and data representability. Furthermore, DBS is easily and efficiently implemented using currently available software packages, and scales well to large datasets. Similar to DBT, the end objective of DBS is to learn a function \( f_\theta(\cdot) \). The difference in DBS is that, instead of a boundary tree, \( f_\theta(\cdot) \) is jointly optimized with a boundary set. With the new approach, the label prediction for a given query point \( x \) is computed with respect to all elements in \( S \). In contrast, DBT only considers data points that are in the tree traversal path (as was described in Section 2.2.4). This modification allows efficient implementation of the algorithm and also increases prediction accuracy. Given a training dataset \( D \), a mini-batch size \( N_b \) and a positive scalar value \( \sigma \), Algorithm 2 outlines the
Algorithm 2: DBS training algorithm

\begin{algorithm}
\begin{algorithmic}
  \Require $D, N_b, \sigma$
  \Ensure $\theta$
  \State randomly initialize $\theta$, parameters of $f_\theta(\cdot)$;
  \While{not reached maximum number of epochs}
    \State shuffle elements and partition $D$ to obtain subsets of size $(N_b + 1)$;
    \ForEach{subset $\bar{D}$}
      \State $U \leftarrow \{(f_\theta(\mathbf{x}_n), y_n) \mid (\mathbf{x}_n, y_n) \in \bar{D}\}$;
      \State $U_b \leftarrow$ first $N_b$ elements of $U$;
      \State $S \leftarrow$ boundary set computed using elements of $U_b$;
      \State $(f_\theta(\mathbf{x}_r), y_r) \leftarrow$ last element of $U$;
      \State $d \leftarrow$ row vector consisting Euclidean distances between each data point in $S$ and $\mathbf{x}_r$;
      \State $w \leftarrow$ softmax function applied on $\frac{-d}{\sigma}$, i.e., $w(i) = \frac{\exp(-d(i)/\sigma)}{\sum_{j \in |S|} \exp(-d(j)/\sigma)}$ for $i \in |S|$;
      \State $Y \leftarrow |S| \times C$ matrix where rows are the one-hot label encodings of elements of $S$;
      \State $\hat{y} \leftarrow wY$ where $\hat{y}(c) = \Pr(y_r = c|\mathbf{x}_r, S, \theta)$ for $c \in [C]$;
      \State $L_{\text{dbs}} \leftarrow$ cross-entropy loss calculated with $\hat{y}$ and $y_r$;
      \State Compute $\nabla_{\theta} L_{\text{dbs}}$ and take one step to minimize $L_{\text{dbs}}$;
    \EndFor
  \EndWhile
\end{algorithmic}
\end{algorithm}

In Algorithm 2 the $i$-th component in $w$ is a measure of the \textit{closeness} of $\mathbf{x}_r$ to the $i$-th data point in $S$, where the sensitivity of the measure is controlled by $\sigma$. The vector-matrix product $wY$ compactly computes the label prediction for $\mathbf{x}_r$, by summing the one-hot label encodings of elements in $S$, weighted by their closeness to $\mathbf{x}_r$. The vector $\hat{y}$ is a vector of the probabilities that $\mathbf{x}_r$ belongs to each class. The algorithm can be efficiently implemented with batch support where the $(f_\theta(\mathbf{x}_r), y_r)$ pair is replaced with a size $N_t$ mini-batch. The Euclidean distance calculation step remains the most costly operation of the algorithm. However its computational complexity is now within our control as the size of the set $S$ is upper bounded by $N_b$. After learning $f_\theta(\cdot)$, a final boundary tree or a boundary forest is built with \textit{all} available training data using $d(\mathbf{x}_i, \mathbf{x}_j) = \|f_\theta(\mathbf{x}_i) - f_\theta(\mathbf{x}_j)\|$ as the distance measure.

We emphasize that the boundary set is used only in the training phase, i.e., to efficiently learn a transformation function $f_\theta(\cdot)$. At the test time, we first build a boundary tree using the transformed versions of all available training data. Given a test data point $\mathbf{x}$, a classification label is assigned to $\mathbf{x}$ by first taking transformation $f_\theta(\mathbf{x})$ and subsequently searching the boundary tree for the approximate nearest neighbours. We refer to this final classifier trained using boundary sets as the ‘DBS classifier’, which will be compared to its boundary tree counterpart ‘DBT classifier’ described in Section 2.2.4.

3.4 Boundary sets and adversaries

In this section we describe how the proposed DBS classifier can be tested against adversarial attacks. We also provide a strong motivation to use the proposed method in place of vanilla neural network classifiers. In terms of the adversarial attacks, the same concepts, as were described in Section 2.3 apply. However, we note that the cross entropy loss $L_{\text{ce}}$ defined in Section 2.3.1 is replaced by corresponding
loss term $\mathcal{L}_{\text{dbs}}$ for the differentiable boundary sets. Given a test data point $\mathbf{z} \in \mathcal{X}$ and its associated label $t \in [C]$, the adversarial examples by ignorant and cognizant adversaries are generated according to the formulas (2.10) and (2.13). The adversaries are only allowed to perturb the test input data point $\mathbf{z}$, i.e., the data points that are in the boundary tree remain unperturbed. Adversarial training using the DBS method can be performed in a manner similar to that explained in Section 2.3.2 by minimizing the combined loss function proposed in (2.12).

One of the motivations to use the neighbour-based DBS method over a vanilla neural network classifier is its robustness against adversarial examples. At the heart of the boundary set algorithm is the (approximate) nearest neighbour search that involves distance measurements in the transformed domain. One important detail here is that the dimension of transformed vector is a design parameter and can be set arbitrarily at the beginning of training. As we describe next, this feature proves to be very handy for the DBS classifier if trained in an adversarial manner. The optimization scheme of the boundary set algorithm presented in Algorithm 2 forces the transformed vectors of the data points from the same class to have smaller distances to each other. The opposite is true for data points that bear different class labels. This results in data points from each class forming clusters of their own. We observe the cluster forming behaviour in experimental results presented in Chapter 4.

Given a test data point $\mathbf{z} \in \mathcal{X}$ and its associated label $t \in [C]$, if $\mathbf{z}$ is correctly classified by the DBS classifier, we can assume that $f_{\theta}(\mathbf{z})$ lies closer to the cluster bearing label $t$ than any other cluster. The objective of an adversary is to perturb $\mathbf{z}$ in a way that the class prediction of the perturbed version is incorrect, which involves moving the transformation of the perturbed version towards a cluster of a different class. We argue that if the output dimension of $f_{\theta}(\cdot)$ is high, moving the output vector by perturbing the input of $f_{\theta}(\cdot)$ would be much trickier than doing so for a low dimensional output vector. This property of the DBS method is not observable in the vanilla neural network classifier since its output dimension is set to match the number of classes. If provided with enough dimensionality for the output of $f_{\theta}(\cdot)$, we show that the DBS method is able to outperform vanilla neural network classifier, in terms of the error rate, for adversarial examples. We provide experimental proof of this claim in Section 4.2.
Chapter 4

Experiments

In this chapter we test and compare the performance of the proposed algorithms. We use the well-known MNIST dataset of handwritten digits (‘Digit-MNIST’) and the newer ‘Fashion-MNIST’ dataset (Xiao et al., 2017). The images in the Fashion-MNIST dataset are of fashion material such as cloths, bags, and boots. It is similar to Digit-MNIST in terms of image dimension, the number of classes, and the amount of training and test data. However, classification is more challenging. Each training dataset consists of 60000 grayscale images that are of dimension $28 \times 28$ (784 pixels in total) and belong to one of ten classes. Each datasets have a separate test set of 10000 images.

4.1 Differentiable boundary sets

In this section we present our experiments performed on the DBS and DBT algorithms. As per the DBT training procedure described in Section 2.2.4, given a mini-batch of size $N_b + 1$, we build a boundary tree $\mathcal{T}$ using the first $N_b$ elements, and use the class prediction of the last, i.e., the $N_b + 1$st element, to take a gradient step towards minimizing the cross-entropy loss. To understand better the DBT algorithm, we implement two DBT variants with one significant difference. In the first variant, DBT-v1, the gradient $\nabla_{\theta} L_{\text{dbt}}$ is calculated only considering the last data point in the mini-batch. This means that even if the data points in $\mathcal{T}$ are obtained via transformation through $f_{\theta}(\cdot)$, they are considered constants with respect to $\theta$. This is equivalent to completely freezing the boundary tree along with the data points in nodes, and then optimizing $f_{\theta}(\cdot)$. In contrast, the second variant, DBT-v2, considers points in the boundary tree as functions of $\theta$ when computing the gradient. Put differently, DBT-v2 computes $\nabla_{\theta} L_{\text{dbt}}$ considering the last data point in the mini-batch, as well as the points in $\mathcal{T}$. This variant can be thought of as jointly optimizing $f_{\theta}(\cdot)$ along with points in the boundary tree. To the best of our understanding, DBT-v1 is what is implemented by Zoran et al. (2017).

All algorithms are implemented using TensorFlow (Abadi et al., 2017) and feed forward fully connected neural networks. In the DBS and DBT algorithm variants, a neural network with two hidden layers is used to implement $f_{\theta}(\cdot)$. We use $\text{relu}(\cdot) = \text{max}(0, \cdot)$ as the activation function for all layers except for the last layer. The Identity function is used as the activation of last layer to model $f_{\theta}(\cdot)$ with an unconstrained range. Along with the layer dimensions, the network can be compactly described as $784 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{identity}} 20$. For comparison, we train a vanilla neural network (VNet) classifier that shares the same $f_{\theta}(\cdot)$ architecture. In the VNet classifier, $f_{\theta}(\cdot)$ is followed by an extra 10-dimensional
Chapter 4. Experiments

layer that uses the softmax activation function to produce class association probabilities at the output. This network can be described as \( 784 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{identity}} 20 \xrightarrow{\text{softmax}} 10 \). We note that all neural networks we employ have a similar construction and henceforth we avoid specifying activation functions in the architecture description.

For each algorithm, we select the hyperparameter set that empirically results in the fastest convergence. Both \( N_b \) and \( N_t \) are set to 1000 in the DBT variants and 100 in DBS. We set \( \sigma \) to 1 in DBT-v1 and to 60 in both DBT-v2 and DBS. The Adam optimizing scheme \cite{Kingma2015} is used with an initial learning rate of 0.0001 for the DBT variants and 0.001 for DBS and VNet. We successively decrease the learning rates by a factor of 10 after 400, 1000 and 3000 epochs. Each model is optimized until the test error no longer decreases or a maximum of 5000 epochs is reached. We consider that one epoch is completed when all 60000 training data points have participated in the training process exactly once. To compute the test errors in the DBT and DBS related experiments, we use a single BT built from all 60000 training points. Consistent with the observations made by \cite{Zoran2017}, we find that testing on a single boundary tree (instead of a forest) does not make a noticeable impact on the test error.

The results are reported in Table 4.1. The ‘number of nodes’ columns indicate the sizes of the final boundary trees built using all available training data. Note that the test error for DBS in each dataset is on a par with the VNet counterpart. Also, we observe that in both the DBT and DBS results, the number of nodes in the final BT is smaller for Fashion-MNIST than for Digit-MNIST, even though the test errors are in reverse order. We argue that this is acceptable because a higher test error does not necessarily mean a more complex class boundary. Next, one of the most beneficial aspects of DBS when compared to DBT is its computational efficiency. The authors of DBT note that, due to the discrete nature of the tree traversals for different query data points, a different TensorFlow computation graph must be built in each iteration of DBT. We note though that our implementation of the DBT algorithm mitigates this limitation by exploiting the disappearance of the traversal-dependent first term in (2.6), as described at the end of Section 2.2.4. However, batch-implementation of the DBT algorithm is still impractical, which remains a significant limiting factor. The training times of the two algorithms depend heavily on their implementations and the type of hardware used, but generally is higher for DBT, as DBT is unable to exploit fully the benefits offered by batch-implementation dependent machine learning tools. In our simulations we observe that, to reach a given test error rate, the proposed DBS algorithm dramatically reduces the required training time. This can be observed in Figure 4.1(a). Our simulations are carried out on a GeForce GTX 1060 6GB GPU and we expect the differences in the training time be even more profound if a more powerful GPU is used.

To better illustrate the representations learned by the DBS algorithm, we set the neural network architecture of \( f_\theta(\cdot) \) to \( 784 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{relu}} 400 \xrightarrow{\text{identity}} 2 \), and plot the learned representations on a 2-dimensional map. Notice the change of the last layer dimension in the new network. The activation functions are kept the same as in the previous experiments. As seen in Figure 4.1(b) the transformation function learns compact clusters that are well separated, so that Euclidean distance can be used to search for neighbours that share the class of a given point. The final BT consists of 37 nodes for this version and achieves a test error of 11.5\% (i.e., higher than in Table 4.1 because here we use a 2-dimensional output layer to aid in visualization).
### Table 4.1: Test error comparison of DBT, DBS and VNet.

<table>
<thead>
<tr>
<th>Model</th>
<th>Digit-MNIST</th>
<th>Fashion-MNIST</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Test error %</td>
<td># of nodes</td>
</tr>
<tr>
<td>DBT-v1</td>
<td>2.23</td>
<td>220</td>
</tr>
<tr>
<td>DBT-v2</td>
<td>1.71</td>
<td>46</td>
</tr>
<tr>
<td>DBS</td>
<td>1.52</td>
<td>29</td>
</tr>
<tr>
<td>VNet</td>
<td>1.48</td>
<td>-</td>
</tr>
</tbody>
</table>

![Figure 4.1](image-url)

**Figure 4.1:** Training time for DBT vs. DBS, and 2-d map of learned representations for Fashion-MNIST. Sub-figure (a) compares training time for DBS and DBT on the Fashion-MNIST dataset. Sub-figure (b) is the visualization of 2-dimensional representations learned for a random subset of Fashion-MNIST training data points. Observe that the five clusters in top-right corner are the closest to each other and are of fashion material that consist of similar prototypical examples.
Chapter 4. Experiments

4.2 Boundary sets and adversaries

One of the motivations to use the boundary sets algorithm in place of vanilla neural networks is the robustness of the boundary set classifier in presence of adversarial examples. We validate this claim by evaluating the two classes of models with Digit-MNIST dataset. The first class of models is obtained by performing adversarial training on the differentiable boundary sets (DBS) method, which uses a neural network architecture of $784 \rightarrow 400 \rightarrow 400 \rightarrow m$ to model the transformation function $f_\theta(\cdot)$. The dimension $m$ is varied from 5 to 340 to obtain multiple models for this class. For comparison, a second class consisting of vanilla neural network classifiers (VNet) is also tested. In this case the neural network architecture is $784 \rightarrow 400 \rightarrow 400 \rightarrow m \rightarrow 10$. Fully connected neural networks employing ReLU activation functions were used in all models. The evaluation results are summarized in Figures 4.2 and 4.3(a).

In Figure 4.2, the error rate plot for the examples of ignorant adversary, ‘DBS - Ign. Adv.’, has a downward trend as the output dimension $m$ is increased. We conjecture that this is because when $m$ is large, small perturbations in the input become less effective in moving the output vector away from its original position. In contrast, the neural network counterpart ‘VNet - Ign. Adv.’ has an upward trend as $m$ is increased. At this point we remind the reader that the dimension of the last layer of the vanilla neural network classifier is fixed at 10, the number of classes. Perturbations at the input layer affect the output layer only through the second-to-last layer, of which the dimension is varied. A larger $m$ means that a greater number of component-wise perturbations are possible in this second-to-last layer. These perturbations can add up constructively when computing the last layer values and result in an output vector that has a large deviation from its original position. With a larger $m$, this property makes it increasingly easier for an adversary to change the predicted label. We conjecture that above is the reason for the upward trend in ‘VNet - Ign. Adv.’.

Figure 4.2: Neural network vs. boundary set performance against an ignorant adversary.

The line plots represent test error rates reported for perturbed and unperturbed versions of MNIST test dataset. The abbreviations VNet and DBS in the legend refer to vanilla neural network and differentiable boundary set classifiers respectively. Keyword Clean refer to the clean test data (unperturbed), whereas Ign. Adv. refer to the examples generated by an ignorant adversary. See text for details and the explanation of observations.
Figure 4.3: Comparing performance against a cognizant adversary on Digit-MNIST dataset.

Sub-figure (a) serves as an extension to Figure 4.2. The line plots ‘DBS - Cogn. Adv.’ and ‘VNet - Cogn. Adv.’ correspond to test error rates obtained for adversarial examples generated by the cognizant adversary. Even though the neural network classifier is doing comparatively better in this case, error rates produced by both classifiers remain higher than 20%. Even with the cognizant adversary, we observe that the error plot for boundary set has a downward trend when the output dimension $m$ increases, which is consistent with our reasoning for observations in Figure 4.2. Depicted in sub-figure (b) are the subtle visual differences of examples generated by two adversaries. In each row, columns from left to right include the original image, example from the ignorant adversary, example from the cognizant adversary, and the difference image between two adversarial examples. Even though the two sets of adversarial examples seem visually similar, the test error rate for cognizant adversary is around three times higher than that for the ignorant adversary.

We note that in the VNet classifier related experiments in Figure 4.2, the minimum test error rates for Clean and Ign. Adv. examples are observed at different training epochs. The line plot ‘VNet - Clean’ corresponds to the minimum error rates observed for Clean examples, and the corresponding error rates for Ign. Adv. at the same epoch are plotted in ‘VNet - Ign. Adv.’. For completeness, we also include the plot ‘VNet - Ign. Adv. min’ which corresponds to the minimum error rates observed for Ign. Adv. examples in any epoch. For a particular model, the ability of an adversary to change the predicted label of an example is dependent on how much the adversary is able to move the output vector in the last layer, only by perturbing the values of the input layer. One of the disadvantages of the vanilla neural networks is that their output dimension is constrained to the number of classes in the dataset and cannot be arbitrarily changed. In contrast, the boundary set method works with an arbitrary output dimension since the label is predicted by measuring the distance to training examples. The results presented in Figure 4.2 provides conclusive proof on how this feature of boundary sets method is useful in defending against examples generated by an ignorant adversary.

Figure 4.3(a) includes the error rates obtained against the cognizant adversary. We observe that the error rates for the VNet classifier are lower than those for the DBS classifier. A similar behaviour is observed when the classifiers are trained without adversarial examples, although we have not included results in this document. The error rate reported for adversarial examples on a regularly trained (without adversarial training) VNet is lower than that on a similarity trained DBS classifier. In the regular trained and cognizant adversary cases, adversaries are attempting to maximize the exact loss functions the classifiers are trained with. We conjecture that the VNet classifier does better in such a circumstance since the objectives of the adversary and the training are matched. On the other hand, the DBS method produces a higher error rate for the examples from the cognizant adversary, due to the same reason the regular trained DBS classifier produces a higher error rate for adversarial examples.
Chapter 5

Conclusions

5.1 Summary

In this thesis, we introduced the boundary set (BS) and differentiable boundary set (DBS) algorithms, which build off the recently proposed boundary trees and differentiable boundary tree algorithm. The DBS procedure iteratively trains a neural network and a boundary set to learn simple representations of complex inputs efficiently. Through the proposed algorithm, we address the computational issues of differentiable boundary trees, and also improves their classification accuracy and data representability. We also compare the performance of proposed DBS algorithm with vanilla neural networks in presence of adversarial examples. Our algorithm is efficiently implementable on currently available software packages and is able to fully exploit the benefits offered by modern machine learning tools.

5.2 Future work

One of the original motivations of boundary trees is the development of a learning algorithm that can quickly adapt to new training data. We note that when combined with the transformation function learned by the DBS algorithm, the ability to adapt quickly can prove useful in certain contexts. For example, consider a situation where the transformation has been learned using the DBS scheme, and a BT is trained and deployed. Whenever new training data becomes available, the BT can easily be re-trained on the new data without adjusting the learned transformation function. This may not be possible in, for example, a neural network classifier since the network itself has to be re-trained to adapt to each new data point. We performed few experiments on this use case but did not obtain satisfactory results with the current form of DBS algorithm. However we point out that it would provide a great value addition if an improved version of the DBS algorithm can prove useful in this use case.

In our experiments related to adversarial examples, we considered only the fast gradient sign method for generating such examples. The existence of adversarial examples, being a relatively recently discovered phenomenon, is currently a very active research area. Researchers have proposed many other methods of generating adversarial examples, some of which are very successful in deceiving classifiers but at a considerable computational cost. We have not yet been able to examine the performance of proposed algorithms with other forms of adversarial examples due to the limitation of hardware resources. We leave this as future work.
Bibliography


