Higher Order Neural Networks and Neural Networks for Stream Learning

Yue Dong

Thesis submitted to the Faculty of Graduate and Postdoctoral Studies in partial fulfillment of the requirements for the degree of Master of Science in Mathematics[1]

Department of Mathematics and Statistics
Faculty of Science
University of Ottawa

© Yue Dong, Ottawa, Canada, 2017

[1]The M.Sc. program is a joint program with Carleton University, administered by the Ottawa-Carleton Institute of Mathematics and Statistics

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance a thesis entitled **Higher Order Neural Networks and Neural Networks for Stream Learning** by **Yue Dong** in partial fulfillment of the requirements for the degree of **Master of Science in Mathematics**.

Dated: _

Abstract

The goal of this thesis is to explore some variations of neural networks. The thesis is mainly split into two parts: a variation of the shaping functions in neural networks and a variation of learning rules in neural networks.

In the first part, we mainly investigate polynomial perceptrons - a perceptron with a polynomial shaping function instead of a linear one. We prove the polynomial perceptron convergence theorem and illustrate the notion by showing that a higher order perceptron can learn the XOR function through empirical experiments with implementation.

In the second part, we propose three models (SMLP, SA, SA2) for stream learning and anomaly detection in streams. The main technique allowing these models to perform at a level comparable to the state-of-the-art algorithms in stream learning is the learning rule used. We employ mini-batch gradient descent algorithm and stochastic gradient descent algorithm to speed up the models. In addition, the use of parallel processing with multi-threads makes the proposed methods highly efficient in dealing with streaming data. Our analysis shows that all models have linear runtime and constant memory requirement. We also demonstrate empirically that the proposed methods feature high detection rate, low false alarm rate, and fast response.

The paper on the first two models (SMLP, SA) is published in the 29th Canadian AI Conference and won the best paper award. The invited journal paper on the third model (SA2) for Computational Intelligence is under peer review.

Dedications

Dedicated to my supervisors, my families and friends.

Acknowledgement

I would like to express my sincere gratitude to my supervisors Dr. Nathalie Japkowicz and Dr. Vladimir Pestov for the continuous support throughout my Masters study, for their encouragement, enthusiasm, and immense knowledge. Their guidance helped me in all the time of research and writing of this thesis.

Besides my advisors, I would like to thank the Data Science and Machine Learning Group for their supports and collaborations. I would also like to express my appreciation to my thesis examiners – Maia Fraser, Tanya Schmah, and Yuhong Guo – for their useful comments on the original version of the thesis that have led to a number of improvements. In addition, I would like to thank the NSERC, the FRQNT, and the Department of Mathematics and Statistics for the Graduate Scholarships and financial supports.

I would also like to thank my parents and my family for their wise counsel and their companion. Finally, there are my friends. Thank you for standing on my side and being there for me through the hard times of my life and thank you for the encouragement.

Contents

List of Symbols **viii**

Higher Order Neural Networks . . . 3
Neural Networks for Stream learning . . . 4

1 Preliminaries **6**

1.1 Supervised Learning and Classification Problems . . . 6
1.2 Feedforward Neural Networks . . . 7
1.2.1 Single Layer Neural Networks . . . 8
The Perceptron with Perceptron Learning Procedure . . . 8
Learning the Weights with Delta Rule . . . 12
1.2.2 Multi-layer Neural Networks . . . 16
Backpropagation Algorithm . . . 18
Various Gradient Descent Algorithms . . . 21
1. Batch Gradient Descent . . . 21
2. Stochastic Gradient Descent . . . 22
3. Mini-batch Gradient Descent . . . 23
Multilayer NN as Universal Approximators . . . 23

2 Higher Order Neural Networks **24**

2.0.1 Vapnik-Chervonenkis Dimension . . . 25
2.1 VC Dimension of Perceptrons and Linear Threshold Networks . . . 26
2.1.1 VC Dimension for Perceptrons . . . 26
2.1.2 VC Dimension for Linear Threshold Neural Networks . . . 27
2.2 VC Dimension for Polynomial Perceptrons and Higher Order Neural Networks . . . 28
2.2.1 VC Dimension for Polynomial Perceptrons . . . 29
2.2.2 VC Dimension for Higher Order Neural Networks . . . 30
2.3 Polynomial Perceptron Convergence Theorem . . . 35
2.3.1 Solving XOR with a Polynomial Percetron . . . 38
Second Order Polynomial Percetrons for the XOR Problem . . . 39
Third Order Polynomial Percetrons for the XOR Problem . . . 39

3 Stream Learning and Concept Drift **46**
3.0.1 "Threaded Ensembles of Supervised and Unsupervised Neural Networks for Stream Learning" (Winner of the Best Paper Award at the 29th Canadian Artificial Intelligence Conference) 46
3.0.2 "Threaded Ensembles of Autoencoders for Stream Learning" (Currenly under Peer Review for *Computational Intelligence*) 59

4 Conclusions and Discussions **80**

List of Symbols

(X, Y)	a random variable taking values in $\Omega \times \{0, 1\}$	6
$\ell(\hat{y}, y)$	loss function of predicting y wrongly by $\hat{y}$	7
η	learning rate	9
$\mathbb{E}(f)$	expected risk over $d\mu$	7
$\mathbb{E}^{(k)} = \ell^{(k)}$	the loss for one example $x^{(k)} \in \sigma$	14
$\mathbb{E}_n(f)$	empirical risk over n samples	7
$\mathbf{b}$	biases	7
$\mathbf{w}$	weights	7
$\mathbf{x}^{(i)}$	ith instance of inputs from a training sample	9
$\mathbf{z} = (\mathbf{x}, y)$	one training example with label	7
$\mathbf{z}^{(i)}$	ith data instance in σ	7
μ	the distribution of the random variable (X, Y)	6
Ω	domain or feature space	6
σ	an training sample with labels	6
θ_s	a state: the values of all parameters in a neural netowrk at a time	9
$a : \mathbb{R} \to \{0, 1\}$	activation function	8
$f_\theta(\mathbf{x})$	a classifier with parameters θ	7
$net_j^{(l)}$	the net input into neuron $X_j^{(l)}$	16
$s : \mathbb{R}^n \to \mathbb{R}$	integration (shaping) function	8
X or $\mathbf{x}$	an observation or an input	6
$X_i^{(l)}$	ith neuron in layer l	16
Y	a label	6
$y^{(i)}$	ith instance's label from a training sample	9

Introduction

Nowadays, millions or even billions of new data records are generated per day; for instance, Google handles over 3 billion searches per day [1]. These datasets, known as *Big data*, have large sizes and are complex. Traditional data analysis techniques, which require loading the entire dataset into the main memory, are usually inadequate in dealing with Big data.

Data Science is a field of studying these datasets (regardless of the size of the datasets) with the major focus on extracting valuable information from the datasets [2]. With the boom of Big data, data scientists start to shift their algorithms towards the direction of parallel and cloud computing [3].

One example involving Big data and data science is to detect anomalies from big sensor datasets (possibly data streams) after training a detection model on the *traning datasets*. *Supervised* anomaly detection techniques require a dataset with the corresponding labels ("normal" or "abnormal") and involve training a *classifier*. The trained models are then used to predict the label for any new observation. On the contrary, *unsupervised* anomaly detection techniques detect anomalies in an unlabeled test dataset. It works well in the class imbalance problems where we assume that the majority of examples in the training dataset are from the normal class. Therefore, a classifier built on these instances would represent normal instances. Then how likely a new test instance is generated by the learned model is measured for detecting the anomaly.

The focus of this thesis is on classification problems where the goal is to classify new inputs into one of the possible categories. In binary classification problems, we suppose (X, Y) is a random variable taking values in $\Omega \times \{0, 1\}$, where Ω is called the *domain* or *feature space*. The variable X is called an *observation* and $Y \in \{0, 1\}$ is called a *label*. The underlying distribution of (X, Y) is given by μ , a probability measure on $\Omega \times \{0, 1\}$. A *binary classifier* is a function (mapping) $f : \Omega \to \{0, 1\}$, and $f(X)$ is called the prediction for the point X.

In any learning algorithm, parameters are involved and they need to be learned through the training process. In other words, each learning algorithm $f_\theta(\mathbf{x})$ is associated with a family of functions ($\mathcal{F}$) with parameters θ. The goal of the learning process is to find the best parameter values for a specific training dataset and generalize them to predict the unseen data points. This is usually achieved by finding a

function $f \in \mathcal{F}$ that minimizes the loss $Q(\mathbf{z}, \theta) = \ell(f_\theta(x), y)$ averaged on the training examples.

There are many classification algorithms in existence. We study neural networks, one of the most popular such algorithms, in this thesis. In general, a *neural network model* has a set of adjustable parameters $\mathbf{w}$ called *weights* (including *biases*). *Learning* or *training* a neural network refers to the process of finding the optimal values for these weights based on the training data.

Novel Contributions of the Thesis

There are two main goals of this thesis. The first goal is to formally introduce higher order neural networks for solving the XOR problem. The second goal is to propose streaming learning algorithms based on neural networks for anomaly detection.

Regarding theoretical contributions of this thesis, we provide a complete proof of the perceptron convergence theorem for a polynomial perceptron. Although this result is already known for perceptrons with a linear shaping function, a full and direct proof for polynomial perceptrons has not been found in the current literature as far as we are aware.

In regard to new practical contributions, this thesis demonstrates the ability of a polynomial perceptron learning the XOR problem. The inability of solving the XOR problem by a perceptron with linear shaping functions has been shown in [4]. This result led almost the whole field of neural networks into hibernation in the Seventies and early Eighties. Through the empirical studies, we show that the XOR problem can be solved easily by changing the shaping function in a perceptron. In fact, we will be able to solve the XOR problem efficiently by a polynomial perceptron in only a few iterations.

The second significant practical contribution of this thesis is the introduction of three models for stream learning and anomaly detection: Streaming Multilayer Perceptrons (SMLP), Streaming Autoencoders (SA), and Streaming Autoencoders with an additional concept drift checker (SA2). All these methods work with Big data and streaming data.

The proposed methods distinguish themselves from the existing models by the following means. Firstly, they process data in only one pass without the need to store all the data instances in the memory. Secondly, SA and SA2 are one-class anomaly detectors, which have the ability to learn data streams with a significant amount of normal data. Thirdly, all three models adapt to concept drift as the models are updated incrementally with continuous learning. Lastly, the models are fast because the data are separated into multiple threads and processed in parallel. Compared to existing benchmark methods, our models feature high anomaly detection rate, low false alarm rate, and less runtime [5].

The main contributions of this work are summarized as follows:

- We propose three fast and accurate algorithms based on neural networks for data stream mining and anomaly detection. These models are novel, fast and scalable.
- We employ an additional checking step on the third model (SA2) to reduce the false positive rate in anomaly detection.
- Empirical studies on multiple benchmark datasets demonstrate that the proposed models outperform or perform comparable to the state-of-the-art methods on most of the benchmark datasets in terms of AUC scores and runtime.

Outline

Higher Order Neural Networks

Chapter 1 introduces the notion of learning and the basics of feedforward neural networks. We start by presenting the computing units, neurons, in neural networks. A neuron is indeed a composition of two functions: a shaping function $s(\mathbf{x})$ and an activation function $a(s(\mathbf{x}))$. We then discuss the Rosenblatt perceptron: the neuron with a linear affine function as the shaping function and a Heaviside function as the activation function. A single Rosenblatt perceptron can only be used for binary classification problems with discrete outputs since the activation function is a step function with range $\{0, 1\}$. The Rosenblatt perceptron learns by the perceptron learning rule as discussed in more details at section 1.2.1.

One notion of guaranteeing the performance of perceptrons is the perceptron convergence theorem. It basically tells that the perceptron learning algorithm will converge under certain steps if the training data are linearly separable. However, the perceptron learning algorithm will not stop if the training samples are not linearly separable. This is because the learning process will not terminate unless all vectors are classified correctly. According to [4], "the most famous example of the perceptron's inability to solve problems with linearly non-separable vectors is the Boolean exclusive-or problem".

Chapter 2 is then focused on solving the XOR problem with higher order neural networks. Higher order neural networks are those networks with higher order shaping functions such as polynomials or trigonometric functions. In fact, we could solve the XOR problem with a single polynomial perceptron.

In Chapter 2, we generalize the perceptron learning rule to polynomial perceptrons and prove the perceptron convergence theorem for polynomial perceptrons. As far as we know, it is the first time such proof has been shown in the literature for polynomial perceptrons. With the theoretical guarantee, we proceeded to implement such polynomial perceptrons. The empirical experiments indicate that polynomial perceptrons with second or third order polynomial shaping functions are very efficient in

learning the XOR problem. We also found that increasing the order of polynomials in the shaping function will speed up the convergence, with the price of bigger chance to over-fit.

Neural Networks for Stream learning

In order to study the increasingly complex data, some continuous neurons, other than perceptrons, have been widely used in the literature as discussed in Chapter 1. These neurons usually use differentiable monotone increasing functions as the activation functions. These activation functions are smooth and therefore allow the delta rule and the backpropagation algorithm to be used for updating the weights in neural networks.

With the delta rule or the backpropagation algorithm, we could propagate the error back throughout neural networks. Thus, we know how to find the gradient of the error (loss or the empirical risk) with respect to a particular weight or bias. For neural networks to learn, we also need algorithms for weights update.

There are a few different update methods based on the backpropagation. Batch gradient descent algorithm updates weights after passing all the data in the dataset and minimize the empirical risk E_n. It is the most commonly used updating rule before the boom of Big data. Unfortunately, it is usually not applicable in stream learning where data streams arrive continuously without stopping. In fact, any batch learning algorithm that requires storing the entire stream for analysis is not suitable for stream learning.

The alternative updating methods are stochastic gradient descent and mini-batch gradient descent algorithms. Both methods work with Big data, as well as streaming data. In Chapter 3, we focus on the applications using these two updating methods in neural networks for stream learning. We propose new algorithms based on neural networks for anomaly detection in data streams.

Chapter 3 is mainly composed of two papers. The first paper was published at the 29th Canadian Artificial Intelligence Conference and won the best paper award. In this paper, we proposed two fast stream learning algorithm based on ensembles of neural networks for anomaly detection. The first model, *Streaming Multilayer Perceptrons*, is a supervised online learning algorithm involving an ensemble of threaded multilayer perceptrons (MLP). The other model, *Streaming Autoencoders* (SA), is a one-class learning algorithm with an ensemble of threaded autoencoders. Training the latter model only requires data from the positive class. Both models feature an ensemble of multilayer perceptrons or autoencoders from multi-threads which evolve with data streams. The employment of multi-threads makes the methods highly efficient due to the parallel processing. Our analysis shows that both methods require linear runtime and constant memory storage space. Compared to the state-of-the-art algorithm, Very Fast Decision Trees (VFDT), the two proposed methods performed

favorably in terms of detection accuracy and training time for the datasets under consideration [5].

The second paper is an invited paper submitted to the journal *Computational Intelligence* which expands the first paper. In this paper, we improved the model *Streaming Autoencoders* (SA) by adding a checking step to distinguish anomalies from concept drift. This ensures that point anomalies are detected in real-time with a low false positive rate in detection. The method is highly efficient because it processes data streams in parallel with multi-threads and alternating buffers. Empirical comparisons to the state-of-the-art methods demonstrate that the proposed method has high detection accuracy, low false alarm rate and fast response with low sensitivity to the parameter settings on the benchmark datasets.

Chapter 1

Preliminaries

1.1 Supervised Learning and Classification Problems

This section introduces the notion of supervised learning and classification problems. In classification problems, we are trying to classify new inputs into one of the possible discrete categories. Applications in which the training data comprise pairs of observations and their corresponding target labels are known as supervised learning problems.

In binary classification problems, we suppose (X, Y) is a random variable taking values in $\Omega \times \{0, 1\}$, where Ω is called the *domain* or *feature space*. Feature space usually refers to the n-dimensional space where the variables of inputs, called features, can take values from. Features are numeric representations of the raw data which can be grouped into two kinds: 1) *quantitative features* where the values can change continuously, and one cannot count the number of different values; 2) *categorical features* where the values can only be chosen from discrete categories such as genders, age groups, etc [6].

The variable X is called an *observation* and $Y \in \{0, 1\}$ is a *label*. Note that an observation and a label can also mean $\mathbf{x}, y$ which are instances of X, Y, we will adopt the latter notation in this thesis. The distribution of this pair is given by μ , a probability measure on $\Omega \times \{0, 1\}$. A *binary classifier* is a function (mapping) $f : \Omega \to \{0, 1\}$. Note that a classifier could have arbitrary range such as $\mathbb{R}$. A prediction for the point X is merely $f(X)$, and it does not need to equal Y.

In supervised learning, *training sample* σ, with sample size n, is a collection of independent labeled observations drawn from μ:

$$\sigma = \{\mathbf{z}^{(1)} = (\mathbf{x}^{(1)}, y^{(1)}), \mathbf{z}^{(2)} = (\mathbf{x}^{(2)}, y^{(2)}), \ldots, \mathbf{z}^{(n)} = (\mathbf{x}^{(n)}, y^{(n)})\}. \tag{1.1.1}$$

Of course, a training sample can itself be treated as a single random variable. Each

example $\mathbf{z} = (\mathbf{x}, y)$ is composed of an observation $\mathbf{x}$ and a label y. Given a classifier with a prediction y, we define the *loss function* as $\ell(\hat{y}, y)$. The loss function measures the error when the true label is y and the model predicts $\hat{y}$. In some applications (mainly binary classification problems), loss ℓ only takes value of 0 or 1.

In any type of learning algorithm, parameters are involved and they need to be learned through the training process. In other words, each learning algorithm $f_\theta(\mathbf{x})$ is associated with a family $\mathcal{F}$ of functions with parameters θ. The goal of the learning process is to find the optimal parameter values for a specific training dataset and generalize them for predicting the unseen data points. This is usually achieved by finding a classifier $f \in \mathcal{F}$ such that the empirical loss $Q(\mathbf{z}, \theta) = \ell(f_\theta(x), y)$ is minimized by f.

Ideally, we want to minimize the averaged loss over the unknown distribution $d\mu$, called the *expected risk*, which is defined as $\mathbb{E}(f)$:

$$\mathbb{E}(f) = \int \ell(f(\mathbf{x}), y) d\mu.$$

However, in practice, we can often only settle for *the empirical risk* $\mathbb{E}_n(f)$:

$$\mathbb{E}_n(f) = \frac{1}{n} \sum_{i=1}^{n} \ell(f(\mathbf{x}^{(i)}), y^{(i)}).$$

According to [7], the empirical risk $\mathbb{E}_n(f)$ measures the performance of a classifier $f_\theta(x)$ on a specific training dataset. In practice, what we want is the generalization performance of this classifier. Thus, we would like to know the expected risk $\mathbb{E}(f)$ which measures the expected performance of a classifier on unseen examples. This notion $\mathbb{E}(f)$ is purely theoretical and it only appears in the ideal assumption.

In general, a *neural network model*, of the kind considered in this thesis, can be regarded simply as a particular choice for the set of functions $f = f(\mathbf{z}; \mathbf{w})$, where the parameters $\mathbf{w}$ are called *weights* (including the *biases*). *Learning* or *training* a neural network refers to the process of finding the optimal values for these weights based on the training data.

1.2 Feedforward Neural Networks

Artificial neural networks (ANNs) were first proposed around 1940 with the purpose of simulating our biological brain. The first model, McCulloch-Pitts neuron model, was proposed by McCulloch and Pitts in 1943. In 1958, Perceptrons and Perceptron learning rule were proposed by Rosenblatt. In 1969, Minsky and Papert demonstrated that XOR cannot be learned by a single layer perceptron. This leads to a dramatic slowdown in the whole field of research on neural networks [8].

For several decades, the use of neural networks was very limited due to the lack of computational power and these unsolved theoretical issues. The research in this field progressed very slowly until several research teams discovered the backpropagation learning algorithm for Multilayer Perceptrons more or less simultaneously around 1986. Neural networks regained its popularity until SVMs came about. SVM research replaced neural networks research in the Nineties, although neural networks were still pretty widely applied. More recently, research in neural networks started again thanks to the interest surrounding Deep Learning.

This chapter mainly covers the basics of the feedforward neural networks. We start with the building blocks for constructing neural networks: computing units (neurons). We then introduce different types of neural networks: from the simplest model such as perceptrons to more complicated model such as multilayer perceptrons. We will discuss the learning rules for neural networks with the backpropagation and different gradient descent algorithms.

1.2.1 Single Layer Neural Networks

This section covers the simplest feedforward neural networks, namely the single-layer neural networks. The basic building blocks of neural networks are called *computing units* (*units*) or *neurons*. Suppose $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$. In a binary supervised setting, a computing unit can be viewed as a mapping f from $\mathbb{R}^n$ to $\{0, 1\}$. f can be further split into two functional parts: an integration (shaping) function $s : \mathbb{R}^n \to \mathbb{R}$ and the activation function $a : \mathbb{R} \to \{0, 1\}$ as figure 1.1 shows.

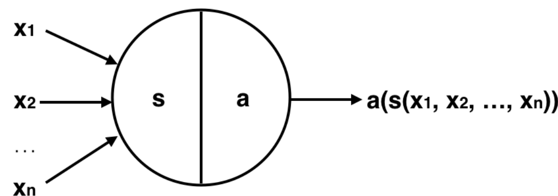

Figure 1.1: A generic computing unit

The Perceptron with Perceptron Learning Procedure

The above computing unit is called a *Rosenblatt perceptron* [9] if $f = a(s(\mathbf{x}))$ such that $s(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$ and $a(x)$ is the Heaviside function define as:

$$a(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0. \end{cases}$$

In summary, the perceptron is a binary classifier:

$$f(\mathbf{x}) = \begin{cases} 1 & \text{if} \quad \mathbf{w} \cdot \mathbf{x} + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

where n is the number of input neurons in a perceptron which is decided by the dimension of training data and $\mathbf{w} \in \mathbb{R}^n$ is the weights. b is called the *bias* in a neural network which shifts the decision boundary in the training sample space. In a perceptron, we could view the bias as a weight from a constant input with value 1 to the neuron.

Given a training set $\sigma = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, \mathbf{x}^{(n)}, y^{(n)})\}$, $\mathbf{x}^{(i)} \in \mathbb{R}^m$. Fix the learning rate $\eta \in (0, 1)$. Suppose the current parameters have the values $\theta_s = (\mathbf{w}, b) \in \mathbb{R}^{m+1}$. The perceptron learns by adjusting weights and bias to reduce the empirical risk $\mathbb{E}_n$ over the training set. Rosenblatts initial perceptron rule can then be summarized by the following steps:

- Initialize the weights and bias to 0:

$$\theta_{s1} = 0 \in \mathbb{R}^{m+1}.$$

- *Update the state sequentially as the algorithm cycle repeatedly through the training samples until stop.*
  In iteration $t = 1, 2, \ldots$, for each training sample $(\mathbf{x}^{(i)}, y^{(i)}) \in \sigma$:
  - Calculate the output value $f^{(i)}(\mathbf{x}) = a(\langle \mathbf{w}^{(i)}, \mathbf{x}^{(i)} \rangle + b^{(i)})$.
  - Update the weights

$$\mathbf{w} = \mathbf{w} + \eta(y^{(i)} - f^{(i)}(\mathbf{x}))\mathbf{x}, \tag{1.2.1}$$

$$b = b + \eta(y^{(i)} - f^{(i)}(\mathbf{x})). \tag{1.2.2}$$

In fact, no update is made if a point is classified correctly. If the pair $(\mathbf{x}, y)$ is misclassified, the value of the weight $\mathbf{w}$ is being moved a bit either towards the data point $\mathbf{x}$ or away from $\mathbf{x}$ depending on the true label of $\mathbf{x}$ as figure 1.2,1.3 show. The algorithm stops when we go through the whole cycle without making any further adjustments, thus, if f^{θ_s} classifiers all the points correctly. Then the current state θ_s is the output [8].

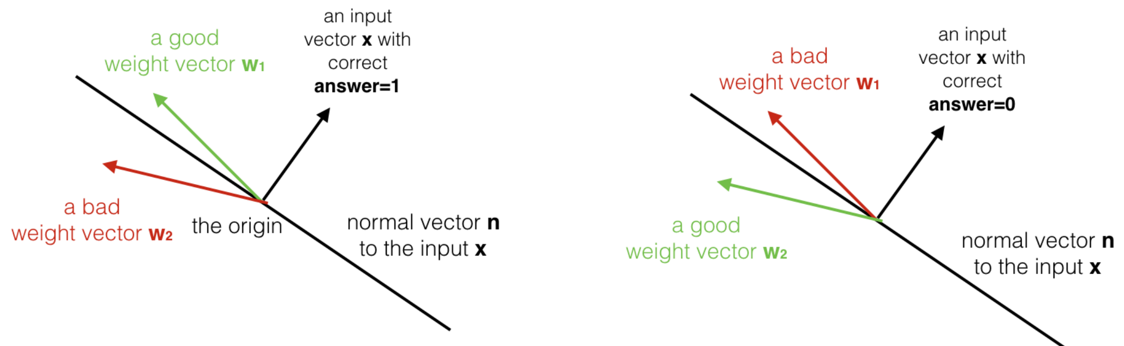

Figure 1.2: Each training point $\mathbf{x}$ defines a plane which is perpendicular to $\mathbf{x}$ [8]. If $y(\mathbf{x}) = 1$, we want a weight vector like $\mathbf{w_1}$ so that $\mathbf{w_1} \cdot \mathbf{x} \geq 0$. Therefore, we move the weights towards $\mathbf{x}$. On the other side, if $y(\mathbf{x}) = 0$, we want a weight vector like $\mathbf{w_2}$ so that $\mathbf{w_2} \cdot \mathbf{x} < 0$. Therefore, we move the weights away from $\mathbf{x}$.

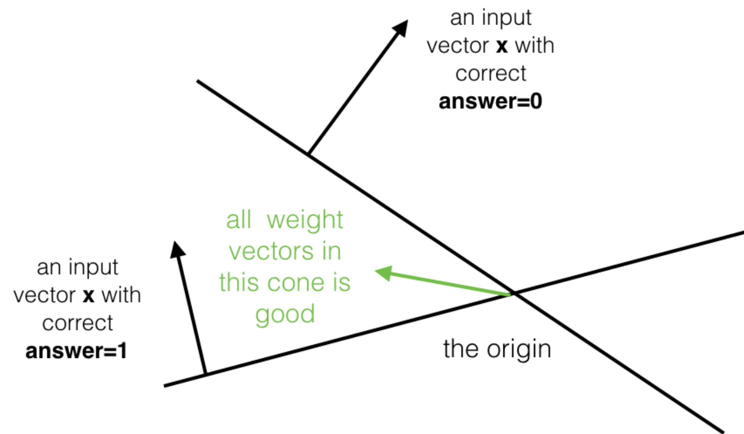

Figure 1.3: To classify all training points right, a weight vector has to be on the different side of all inputs with $y = 1$ and on the same side of all inputs with $y = 0$. If the two classes are linearly separable, there exist some weight vectors that classify all points correctly. Accoding to [8], these weights vectors lie in a hyper-cone with its apex at the origin. Note that the weights lie in a concave region and the average of two good weight vectors is a also good weight vector [8].

Definition 1.1. (Perceptron Convergence Theorem [10])
Let $\sigma = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$ be a training sample. Suppose there exists a state $\theta_{s^*} = (\mathbf{w}^*, b^*) \in \mathbb{R}^{m+1}$ with

$$||\theta_{s^*}|| = 1 \tag{1.2.3}$$

and $f^{\theta_s^*}(\mathbf{x}) = y$ for all $(\mathbf{x}, y) \in \sigma$.

Denote the margin (minimal distance between a data point and the separation hyperplane) as

$$\gamma = min_{(\mathbf{x},y)\in\sigma}|\langle \mathbf{w}^*, \mathbf{x}\rangle + b|.$$

Suppose $\gamma > 0$ and let R be the radius of a ball in $\mathbb{R}^m$ centered at zero which contains the datasets σ:

$$R = \max_{(\mathbf{x},y)\in\sigma}||\mathbf{x}||.$$

Then for every learning constant $\eta > 0$, the perceptron learning algorithm converges in no more than

$$\frac{R^2 + 1}{\gamma^2}$$

steps to a state θ_s which classifies all the training data right:

$$\forall(\mathbf{x}, y) \in \sigma, f^{\theta_s}(\mathbf{x}) = y.$$

Proof. See book [10] for the full proof, and a similar proof can be found in the proof of polynomial perceptron convergence theorem in section 2.3.

Remark. The result is independent of learning rate because the terms η cancel out during establishing upper and lower bounds for θ_s.

Remark. Note 1.2.3 is possible because every state θ_s is an element of the Euclidean space $\mathbb{R}^{m+1}$ which has a norm denoted as

$$||\theta_s||^2 = \sum_{i=1}^{m} w_i^2 + b^2.$$

The Perceptron Convergence Theorem basically tells that the perceptron learning algorithm will converge under certain number of steps if the training dataset is linearly separable. However, the perceptron learning algorithm will not stop if the training dataset is non-linearly separable. This is because the learning process will not terminate unless all vectors are classified correctly. According to [4], “the most famous example of the perceptron’s inability to solve problems with linearly non-separable vectors is the Boolean exclusive-or problem”.

In order for neural networks to achieve better approximations, one common practice is to add layers of hidden neurons (neurons which neither represent inputs nor

outputs) as discussed in the section 1.2.2 of multilayer perceptrons. Another way to boost the capability of a single layer neural network is to change the shaping function. As demonstrated in the chapter of higher order neural networks (2), XOR problem can be solved by a perceptron if we change the shaping function from linear to polynomial.

Learning the Weights with Delta Rule

Note that the perceptron learning rule guarantees its performance by adjusting the weights to be closer to a feasible set of "good" weights. The perceptron learning rule only works for classification problems due to the range of the Heaviside function (which is $\{0, 1\}$). The most intuitive loss, 0-1 loss, is used in such applications for binary classifications.

In other applications, such as regression with neural networks, we need to define a loss function smoothly depending on the weights. Even in classification problems, sometimes we want the outputs of the neural networks to be continuous, so we could define our own thresholds for classes. In these applications, perceptron learning rule is not applicable anymore.

A more widely used learning rule for single layer neural networks is the *delta rule*. The delta rule are used in single-layer neural networks for updating the weights of neurons with gradient descent algorithms. Contrary to adjusting the weights vectors to be closer to a feasible set of "good" weights, delta rule ensures that the empirical risk is minimized by moving the output values $\hat{\mathbf{y}}$ closer the true values $\mathbf{y}$ ($\mathbf{y}, \hat{\mathbf{y}} \in \mathbb{R}^n$ where n is the number of samples in σ).

To apply delta rule for learning, we need a continuous loss function $\ell(y, \hat{y})$ rather than the $0-1$ discrete loss. The empirical risk is then defined as the average loss over the training data. In order for the delta rule to work, we also need to use differentiable monotone increasing activation functions in neurons. Similarly, the shaping functions have to be continuous and differentiable as well. The delta rule is a simplified version of the backpropagation algorithm [11].

Given a labeled data sample $\mathbf{z} = (\mathbf{x}, y)$, where $\mathbf{x} \in \mathbb{R}^m$. Consider a neuron N with a differentiable shaping function $s : \mathbb{R}^m \to \mathbb{R}$ parameterized by the weights $\mathbf{w} \in \mathbb{R}^q$ and a differentiable monotone increasing activation function $a : \mathbb{R} \to \mathbb{R}$. The output predicted by the neuron is then $\hat{y} = a(s(\mathbf{x}))$. The delta rule minimizes the loss by adjusting weights w_i for all $i = 1, 2, \ldots, q$. Thus, we need to calculate the gradient of loss ℓ with respect to the the weights $\mathbf{w}$. The following definition 1.2 is the delta rule for a single layer neural network with k neurons. We assume that each neuron $N_j, j = 1, 2, \ldots, k$ uses the loss function $\ell(y_j, \hat{y_j}) = \frac{1}{2}(y_j - \hat{y_j})^2$ and the shaping functions$s_j = \sum x_i w_{ji}$.

Definition 1.2 (Delta Rule [12])**.** Suppose for a neuron N_j, we define the loss function as $\ell(y_j, \hat{y_j}) = \frac{1}{2}(y_j - \hat{y_j})^2$ and use the linear shaping function $s_j = \sum x_i w_{ji}$. Assume

the activation function $a(x)$ is differentiable monotone increasing. Then the *delta rule* for N_j's ith weight w_{ji} is defined as

$$\Delta w_{ji} = -\eta \frac{\partial \ell}{\partial w_{ji}} = \eta(y_j - \hat{y_j})a'(s_j)x_i \quad w_{ji} = w_{ji} + \Delta w_{ji}. \tag{1.2.4}$$

Where $\eta \in (0,1)$ is the learning rate, x_i is the ith input to N_j, y_j is the true value of the jth neuron N_j, and $\hat{y}_j = a(s_j)$ is the predicted value by the neuron N_j.

Remark. We usually use the squared error for the loss and the empirical risk. Moreover, we add the coefficient 1/2 for easy differentiation: $\ell(y_j, \hat{y_j}) = \frac{1}{2}(y_j - \hat{y_j})^2$ and $\ell' = (y_j - \hat{y_j})$.

Note we also substitute the bias of the jth neuron (b_j) by $x_0 w_{j0}$ where $x_0 = 1$.

The above definition only learns through incremental (sequential) learning. Thus, all weights are updated every time a training data point is passed for training. In other words, we update the weights every time after seeing an example. To minimize the empirical risk $\mathbb{E}_n$ over a training sample with n data points $\sigma = \{(\mathbf{x}^{(1)}, y^{(1)}), (\mathbf{x}^{(2)}, y^{(2)}), \ldots, (\mathbf{x}^{(n)}, y^{(n)})\}$, we need to modify the above delta rule for batch learning [12]:

$$\Delta w_{ji} = -\eta \frac{\partial \mathbb{E}_n}{\partial w_{ji}} = \eta \sum_{k=1}^{n} (y_j^{(k)} - \hat{y_j}^{(k)})a'(s_j^{(k)})x_i^{(k)} \quad w_{ji} = w_{ji} + \Delta w_{ji}. \tag{1.2.5}$$

Where $\eta \in (0,1)$ is the learning rate, $y_j^{(k)}$ is the true label of the jth neuron for the kth training data point, $\hat{y}_j^{(k)} = a(s_j^{(k)})$ is the output of jth neuron for the kth data point in the training dataset and $x_i^{(k)}$ is the ith input of the kth data point and $s_j^{(k)} = \sum x_i^{(k)} w_{ji}$.

Contrary to incremental or online learning, in *batch learning*, the weights are only updated after all training samples in the dataset are passed through the learning algorithm. We call the process of passing all data points in the training dataset to a learning algorithm *one epoch* (or *iteration*). The following are a few popular neurons using delta rule for batch learning.

Example 1.3 (Linear neurons [10])**.** *A linear neuron is a composition of a linear shaping function, which is the weighted sum of its inputs, and an identify activation function:*

$$\hat{y} = \sum_{i=1}^{n} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}.$$

As above function shows, a linear neuron has real value outputs. The linear

neuron is usually updated by the delta rule with the empirical risk defined as:

$$\mathbb{E}_n = \frac{1}{2} \sum_{k \in training} (y^{(k)} - \hat{y}^{(k)})^2$$

where $\mathbb{E}^{(k)} = \ell^{(k)} = \frac{1}{2}(y^{(k)} - \hat{y}^{(k)})^2$ is the loss for one example $(\mathbf{x}^{(k)}, y^{(k)}) \in \sigma$.

We differentiate $\mathbb{E}_n$ in order to propagate the error back for weights update:

$$\begin{aligned} \frac{\partial \mathbb{E}_n}{\partial w_i} &= \sum_k \frac{\partial \mathbb{E}^{(k)}}{\partial w_i} \\ &= \sum_k \frac{\partial \mathbb{E}^{(k)}}{\partial \hat{y}^{(k)}} \frac{\partial \hat{y}^{(k)}}{\partial w_i} \\ &= \sum_n -(y^{(k)} - \hat{y}^{(k)}) x_i^{(k)}. \end{aligned} \tag{1.2.6}$$

The batch delta rule then adjusts each weight w_i in proportion to the error derivative $\frac{\partial \mathbb{E}_n}{\partial w_i}$:

$$\Delta w_i = -\eta \frac{\partial \mathbb{E}_n}{\partial w_i} = \eta \sum_k (y^{(k)} - \hat{y}^{(k)}) x_i^{(k)}.$$

Example 1.4 (Logistic neurons [13])**.** *In a logistic neuron, the shaping function is* $s = \sum_i x_i w_i$ *and the activation function is a sigmoid function defined as* $a(s) = \frac{1}{1+e^{-s}}$.

This derivative of this function can be easily calculated in terms of the function itself by the Chain Rule:

$$a'(s) = a(s)(1 - a(s)).$$

To minimize $\mathbb{E}_n$ with respect to the weights, we compute

$$\begin{aligned} \frac{\partial \mathbb{E}_n}{\partial w_i} &= \sum_k \frac{\partial \mathbb{E}^{(k)}}{\partial w_i} \\ &= \sum_k \frac{\partial \mathbb{E}^{(k)}}{\partial \hat{y}^{(k)}} \frac{\partial \hat{y}^{(k)}}{\partial h^{(k)}} \frac{\partial h^{(k)}}{\partial w_i} \end{aligned} \tag{1.2.7}$$

where $\mathbb{E}^{(k)} = \ell^{(k)} = \frac{1}{2}(y^{(k)} - \hat{y}^{(k)})^2$ is the loss for one example $(\mathbf{x}^{(k)}, y^{(k)}) \in \sigma$.

If we use squared sum error for the loss and the empirical risk, the equation 1.2.7 becomes:

$$\frac{\partial \mathbb{E}_n}{\partial w_i} = \sum_k -(y^{(k)} - \hat{y}^{(k)}) \cdot \hat{y}^{(k)}(1 - \hat{y}^{(k)}) \cdot x_i^{(k)}.$$

Therefore,

$$\Delta w_i = -\eta \frac{\partial \mathbb{E}_n}{\partial w_i} = \eta \sum_k (y^{(k)} - \hat{y}^{(k)}) \cdot \hat{y}^{(k)}(1 - \hat{y}^{(k)}) \cdot x_i^{(k)}$$

and the weight is updated by $w_i = w_i + \Delta w_i$.

Remark. If a is the logistic sigmoid function in equation 1.2.4, a' will be close to zero for large inputs and small inputs as figure 1.4 shows. This creates a problem in learning deep neural networks called *vanishing gradient problem* – "the gradients of the network's output with respect to the weights in the early layers become extremely small" [14].

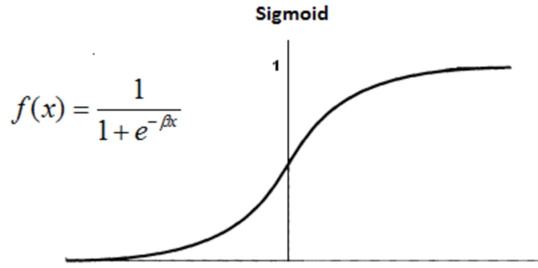

Figure 1.4: A sigmoid activation function. The derivative of this function appoaches to zero when x is large ($x > N$) or x is small ($x < -N$), $N \geq 5$.

Rectified Linear Unit (ReLU) is one of the models introduced to avoid this problem. Instead of using a "squashing" activation function, which maps the input space into a small region, ReLU's activation function has the range from 0 to ∞.

Example 1.5 (Rectified linear unit -ReLU [15])**.** *In a rectified linear unit, the shaping function is* $s = \sum_i x_i w_i$ *and the activation function is* $a(s) = max(0, s)$*. The derivative of* $a(s)$ *is defined by:*

$$a'(s) = \begin{cases} 0 & if\ s \leq 0 \\ 1 & if\ s > 0 \end{cases}$$

Therefore,

$$\Delta w_i = -\eta \frac{\partial \mathbb{E}_n}{\partial w_i} = \begin{cases} \eta \sum_k (y^{(k)} - \hat{y}^{(k)}) \cdot x_i^{(k)} & if\ s > 0 \\ 0 & if\ s \leq 0. \end{cases}$$

Note that although $a(0)$ is not differentiable, we define the derivative $a'(0)$ to be zero in order to apply the delta rule.

The proposition of using ReLU as a non-linear activation function [16] has been shown effective and efficient in training deep supervised neural networks. Moreover,

such models don't require unsupervised pre-training by autoencoders [16]. This allows faster training of deep neural networks in applications such as computer vision and natural language processing.

However, in practice, ReLU suffers from the "Knockout Problem": with large learning rates, "a large gradient can kill a ReLU (dying ReLU) such that it never gets activated again" [17]. "Leaky" ReLU was proposed with non-zero values for negative inputs ($y = 0.01x$ if $x < 0$) to improve the knockout problem.

Remark. Note that all the above neurons using linear shaping functions. Therefore, they don't have the ability to learn the XOR problem. There are two general approaches could be used to solve the XOR problem. One is to use non-linear shaping functions, for example, higher order neural networks as discussed in chapter 2. The other approach is to increase the number of layers (add hidden layers) in the neural networks, such as multilayer perceptrons discussed in the next section 1.2.2.

1.2.2 Multi-layer Neural Networks

Without hidden units (neurons which neither represent inputs nor outputs), first order neural networks with linear shaping functions are very limited for learning. In fact, only linear separable data can be effectively learned. However, if we use multiple layers of hidden units, more complex datasets can be learned easily. Such networks with multiple layers of hidden units are called *multi-layer neural networks*.

Consider a general neural network consisting of L layers. Note that the input layer is not counted in these L layers because conventionally, the input layer is counted as layer 0. Therefore, an N layers network represents a neural network with one input layer and N layers of non-input units.

Pick an arbitrary layer l with N_l neurons, denoted as $X_1^{(l)}, X_2^{(l)}, \ldots, X_{N_l}^{(l)}$. Then each neuron has an activation function $a^{(l)}$ and the activation functions from different layers of the network may not be the same. The activation functions in layer l are calculated based on the signal received from the previous layer, $l-1$. The connection between layer $l-1$ to layer l can therefore be represented in an $N_{l-1} \times N_l$ weight matrix $\mathbf{W}^{(l)}$. Each element $W_{ij}^{(l)}$ represents the weight from $X_i^{(l-1)}$ to $X_j^{(l)}$. Each neuron $X_j^{(l)}$ is also associated with a bias vector $b_j^{(l)}$. We call the output of the neuron $X_j^{(l)}$ as its activation a_j^l.

We denote $net_j^{(l)}$ as net inputs into neuron $X_j^{(l)}$. It can be calculated as

$$net_j^{(l)} = \sum_{i=1}^{N_{t-1}} a_i^{(l-1)} w_{ij}^{(l)} + b_j^{(l)}, j = 1, 2, \ldots, N_l. \tag{1.2.8}$$

The activation (output of a neuron after applying the activation function) of

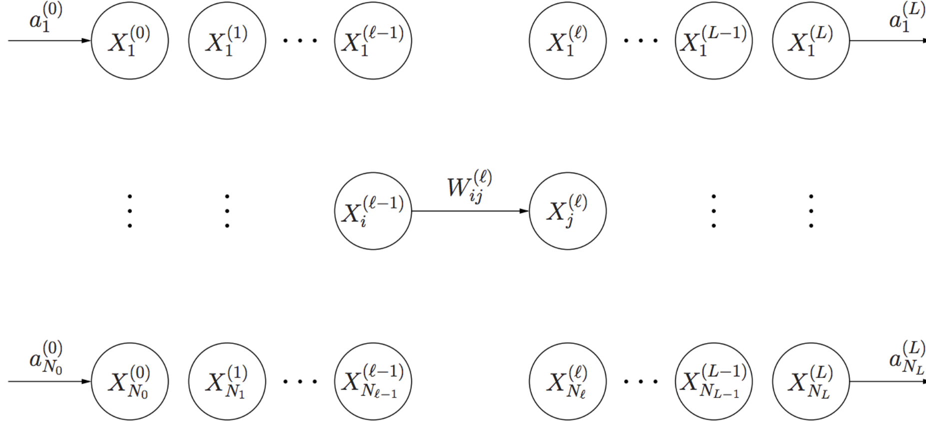

Figure 1.5: This graph demonstrades a general feedforward multilayer neural network [18]

neuron $X_j^{(l)}$ can be computed based on the net inputs:

$$a_j^{(l)} = a^{(l)}(net_j^l) = a^{(l)}\left(\sum_{i=1}^{N_{l-1}} w_{ij}^{(l)} + b_j^{(l)}\right). \tag{1.2.9}$$

As denoted, we count the input layer as layer 0. Suppose a data instance $\mathbf{x}$, also called an input vector, has N components. Then the input layer of the network has the same number of inputs neurons $N_0 = N$. Assume the activations of layer 0 are $a_i^{(0)} = x_i, i = 1, 2, \ldots, N_0$.

In a neural network, the last layer L is the output layer with class labels. The size of the output layer is decided by how many discrete classes the data points in a particular dataset can be classified in. We assuming that there are M such categories, then $N_L = M$. Each element in the output vector $\mathbf{y}$ is given by $y_j = a_j^{(L)}, j = 1, 2, \ldots, M$.

For any neuron in an arbitrary layer, we could calculate the activations forwardly by using the above equations with weights and biases. The activations $\mathbf{y}$ of the output layer is the network output vector which makes the prediction. In order to find optimal weights and biases, we need a mechanism to train the network for performing certain tasks. One popular such mechanism is the backpropagation algorithm.

Backpropagation Algorithm

Consider a general network as described in the previous section, we demonstrate how the backpropagation algorithm (BP) algorithm works in this section. We assume the labels of the training data are given and therefore the models are trained under supervised setting. Suppose that a labeled data sample $\sigma = \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), \ldots, (\mathbf{x}^{(n)}, \mathbf{y}^{(n)})\}$ are given. Assume that all training vectors $\mathbf{x}^{(i)}(i = 1, \ldots, n)$ have N components,

$$\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)}, \ldots, x_N^{(i)})$$

and their corresponding targets have M components,

$$\mathbf{y}^{(i)} = (y_1^{(i)}, y_2^{(i)}, \ldots, y_M^{(i)}).$$

Note that the targets can be arbitrary vectors $\mathbf{y} \in \mathbb{R}^M$ with M components. In multi-classes classification problems with M classes, the target vector $\mathbf{y} \in [0,1]^M$ has the constraint that the components of $\mathbf{y}$ sum up to 1. Compared to binary classifications where we only need one output neuron which takes value $y \in \{0, 1\}$, we need M neurons in the output layer for multi-classes classification problems with M classes. Each neuron gives an output (can be a probability) indicating if the data instance belongs to this class or not.

We first consider the case of incremental learning. Thus all weights and biases are updated after one data sample is presented to the network during training. The activations and the outputs $\hat{\mathbf{y}}$ are obtained by propagating the inputs forwardly using equations 1.2.8, 1.2.9. Backpropagation is then used to propagate the error (usually defined as 1.2.10) back to each weight or bias based on this training data sample.

$$\mathbb{E}_1 = \frac{1}{2}||\mathbf{y} - \hat{\mathbf{y}}||_2^2 = \frac{1}{2}\sum_{i=1}^{M}(y_i - \hat{y}_i)^2 \tag{1.2.10}$$

Since $\mathbf{y}$ is obtained by propagate inputs through all layers, the squared error $\mathbb{E}_1$ is then a function of all weights and biases of the network. To update the model, we therefore need to find the derivatives for all weights and biases $\nabla_{\mathbf{w}}\mathbb{E}_1(\mathbf{z}, \mathbf{w_t})$. In online learning, we update the weights and biases after seeing one example:

$$w_{ij}^{(l)}(t+1) = w_{ij}^{(l)}(t) - \eta\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(l)}(t)}$$

$$b_j^{(l)}(t+1) = b_j^{(l)}(t) - \eta\frac{\partial \mathbb{E}_1}{\partial b_j^{(l)}(t)}.$$

Where η is the learning rate and $\mathbb{E}_1 = \ell(\hat{y}, y)$ is the error or loss of one example.

We can rewrite $\mathbb{E}_1 = \ell(\hat{y}, y)$ in terms of the activations and net inputs in order to investigate the dependency between the error and all the weights and biases.

$$\begin{aligned} \mathbb{E}_1 &= \frac{1}{2}||\hat{\mathbf{y}} - \mathbf{y}||^2 = \frac{1}{2}||\mathbf{a}^{(l)} - \mathbf{y}||^2 \\ &= \frac{1}{2}||a^{(l)}(\mathbf{net}^{(l)}) - \mathbf{y}||^2 \end{aligned} \tag{1.2.11}$$

Based on the above equation, we could use the chain rule to compute the partial derivatives of $\mathbb{E}_1$. To simplify the computation, we define the sensitivity vector based on [18]:

$$sen_k^{(l)} = \frac{\partial \mathbb{E}_1}{\partial net_k^{(l)}} \quad k = 1, 2, \ldots, N_l.$$

Then,

$$\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(L)}} = \sum_{k=1}^{N_L} \frac{\partial \mathbb{E}_1}{\partial net_k^{(L)}} \frac{\partial net_k^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial \mathbb{E}_1}{\partial net_j^{(L)}} \frac{\partial net_j^{(L)}}{\partial w_{ij}^{(L)}} = sen_j^{(L)} \cdot \frac{\partial net_j^{(L)}}{\partial w_{ij}^{(L)}}. \tag{1.2.12}$$

Similarly,

$$\frac{\partial \mathbb{E}_1}{\partial b_j^{(L)}} = \frac{\partial \mathbb{E}_1}{\partial net_j^{(L)}} \frac{\partial net_j^{(L)}}{\partial b_j^{(L)}} = sen_j^{(l)} \cdot \frac{\partial net_j^{(L)}}{\partial b_j^{(L)}}. \tag{1.2.13}$$

For a general layer l other than the last layer L, we can write

$$\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(l)}} = \frac{\partial \mathbb{E}_1}{\partial net_j^{(l)}} \frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}} = sen_i^{(l)} \cdot \frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}}. \tag{1.2.14}$$

The sensitivity vectors $sen^{(l)}$ for $l = 1, 2, \ldots, L-1$ can then be computed as:

$$\begin{aligned} sen_j^{(l)} = \frac{\partial \mathbb{E}_1}{\partial net_j^{(l)}} &= \sum_{k=1}^{N_{l+1}} \frac{\partial \mathbb{E}_1}{\partial net_k^{(l+1)}} \frac{\partial net_k^{(l+1)}}{\partial net_j^{(l)}} \\ &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)} \cdot \frac{\partial net_k^{(l+1)}}{\partial net_j^{(l)}}. \end{aligned} \tag{1.2.15}$$

As equation 1.2.15 shows, the sensitivity of a neuron in layer $l-1$ depends backwardly on all sensitivities in layer l. Thus, the sensitivities can be calculated recursively from layer L to layer 0. Thus, we compute the sensitivities backwardly from the last layer to any given layer, as the name backpropagation indicates [18].

On the other hand, before we could calculate all sensitivities, we forward the input values from layer 0 to an arbitrary layer to compute the activations and the net

inputs with equations 1.2.8 and 1.2.9.

Similarly to the delta rule, we need a differentiable monotone non-decreasing function as the activation function in multilayer neural networks for training with the backpropagation algorithm. One of the most popular multilayer neural networks trained by backpropagation is the multilayer perceptrons.

Example 1.6 (Multilayer Perceptrons)**.** *In multilayer perceptrons, we use linear shaping functions $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b$ and sigmoid activation functions in every layers. If we use the squared error $\mathbb{E}_1 = \frac{1}{2}||\hat{\mathbf{y}} - \mathbf{y}||_2^2 = \frac{1}{2}\sum_{i=1}^{M}(\hat{y}_i - y_i)^2$ as the error (loss) function, the equation 1.2.17 can be simplified as:*

$$\begin{aligned}\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(L)}} &= \frac{\partial \mathbb{E}_1}{\partial net_j^{(L)}}\frac{\partial net_j^{(L)}}{\partial w_{ij}^{(L)}} = \frac{\partial \mathbb{E}_1}{\partial y_j}\frac{\partial y_j}{\partial net_j^{(L)}}\frac{\partial net_j^{(L)}}{\partial w_{ij}^{(L)}} \\ &= -(\hat{y}_j - y_j)\cdot y_j(1-y_j)\cdot a_i^{L-1}.\end{aligned} \tag{1.2.16}$$

Similarly, the equation 1.2.13 can be simplified as:

$$\begin{aligned}\frac{\partial \mathbb{E}_1}{\partial b_j^{(L)}} &= \frac{\partial \mathbb{E}_1}{\partial net_j^{(L)}}\frac{\partial net_j^{(L)}}{\partial b_j^{(L)}} = \frac{\partial \mathbb{E}_1}{\partial y_j}\frac{\partial y_j}{\partial net_j^{(L)}}\frac{\partial net_j^{(L)}}{\partial b_j^{(L)}} \\ &= -(\hat{y}_j - y_j)\cdot y_j(1-y_j).\end{aligned} \tag{1.2.17}$$

For layers other than layer L, the sensitivity vector 1.2.15 can be computed as

$$\begin{aligned}sen_j^{(l)} &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot\frac{\partial net_k^{(l+1)}}{\partial net_j^{(l)}} \\ &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot\frac{\partial\Big(\sum_{m=1}^{N_l} a_{log}(net_m^{(l)})\cdot w_{mk}^{(l)} + b_k^{(l)}\Big)}{\partial net_j^{(l)}} \\ &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot\frac{\partial\Big(\sum_{m=1}^{N_l} a_{log}(net_m^{(l)})\cdot w_{mk}^{(l)} + b_k^{(l)}\Big)}{\partial net_j^{(l)}} \\ &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot\frac{\partial\Big(a_{log}(net_j^{(l)})\cdot w_{jk}^{(l)} + b_k^{(l)}\Big)}{\partial net_j^{(l)}} \\ &= \sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot a'_{log}(net_j^{(l)})\cdot w_{jk}^{(l)} \\ &= a_{log}(net_j^{(l)})\cdot(1-a_{log}(net_j^{(l)}))\sum_{k=1}^{N_{l+1}} sen_k^{(l+1)}\cdot w_{jk}^{(l)}.\end{aligned} \tag{1.2.18}$$

Therefore, based on 1.2.19 we could calculate $\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(l)}}$ *for every layer recursively:*

$$\begin{aligned}\frac{\partial \mathbb{E}_1}{\partial w_{ij}^{(l)}} &= sen_j^{(l)} \cdot \frac{\partial net_j^{(l)}}{\partial w_{ij}^{(l)}} \\ &= sen_j^{(l)} \cdot \frac{\partial \Big(\sum_{m=1}^{N_{l-1}} w_{mj}^{l-1} \cdot a_m^{l-1} + b_j^{l-1} \Big)}{\partial w_{ij}^{(l)}} \\ &= sen_j^{(l)} \cdot a_i^{l-1}.\end{aligned} \tag{1.2.19}$$

Similarly,

$$\frac{\partial \mathbb{E}_1}{\partial b_j^{(l)}} = sen_j^{(l)} \cdot \frac{\partial net_j^{(l)}}{\partial b_j^{(l)}} = sen_j^{(l)}. \tag{1.2.20}$$

Equation 1.2.19 and 1.2.20 demonstrate how to calculate the gradient for the error (loss) with respect to a particular weight or bias in multilayer perceptrons.

For more on this topic, see [18], where the above comes from.

Various Gradient Descent Algorithms

In general, backpropagation refers to the method of propagate the error back. Thus, how to find the gradient of the error (loss or the empirical risk) with respect to a particular weight or bias. For neural networks to learn, we also need weights updating methods. We discuss a few different backpropagation-based methods on updating the weights in this section.

1. Batch Gradient Descent

Batch gradient descent algorithm (GD) has often been used in static datasets because it minimizes the empirical risk $\mathbb{E}_n(f_{\mathbf{w}})$ directly over n data samples in the training dataset [7]. The weights $\mathbf{w}$ are updated in iterations based on the the gradient $\mathbb{E}_n(f_{\mathbf{w}})$ averaged over all data samples :

$$\nabla \mathbb{E}_n(f_{\mathbf{w}}) = \sum_{i=1}^{n} \nabla_{\mathbf{w}} \mathbb{E}_1(\mathbf{z}^{(i)}, \mathbf{w}_t)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \frac{1}{n} \sum_{i=1}^{n} \nabla_{\mathbf{w}} \mathbb{E}_1(\mathbf{z}^{(i)}, \mathbf{w}_t) \tag{1.2.21}$$

where η is the learning rate. In batch learning, the learning rate has to be chosen carefully. A small learning rate will lead to slow parameters updating and a large

learning rate might make the weights oscillate around the global or local minimum [7].

2. Stochastic Gradient Descent

Compared to GD, stochastic gradient descent (SGD) is faster in updating since it only requires the gradient of one data sample in every update. However, SGD usually only converges to the area of global or local minimum of the error [7].

In SGD, the gradient is computed on the basis of a randomly picked single example $\mathbf{z}_t$:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta_t \nabla_{\mathbf{w}} \mathbb{E}_1(\mathbf{z}_t, \mathbf{w}_t). \tag{1.2.22}$$

Each stochastic gradient $\nabla_{\mathbf{w}} \mathbb{E}_1(\mathbf{z}_t, \mathbf{w}_t)$ is an approximation of the gradient averaged over all training data samples 1.2.21. However, the stochastic process often introduces noises for the updating as figure 1.6 shows.

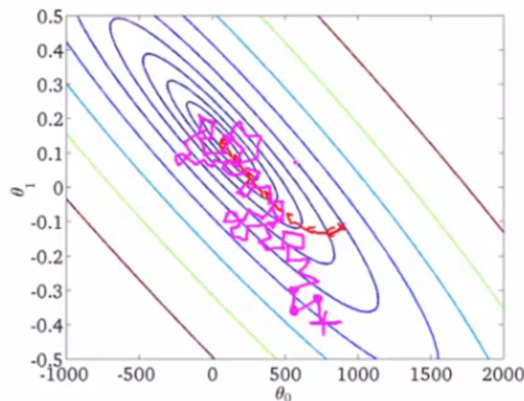

Figure 1.6: The figure is taken from [19]. Here we consider a neural network with two weights θ_0 and θ_1. By altering the values of these two parameters, we could reach the local or global minimum of the error function. The circle lines are the contour lines of the error. The red line is the trace of batch gradient descent algorithm and the purple line is the the trace of stochastic gradient descent algorithm.

According to [7], "stochastic gradient descent directly optimizes the expected risk". However, as we can see from figure 1.6, SGD suffers from the noises introduced by individual examples. The purple circles in the figure 1.6 indicate that stochastic gradient descent will keep updating without stopping. In practice, this problem is less severe with the following techniques:

- shuffle the training set before each epoch;

- use an adaptive learning rate instead of a constant one to reduce the range of the approximation for the global or local minimum.

3. Mini-batch Gradient Descent

Mini-batch gradient descent is an another weights updating approach which can work faster than GD and more reliable than SGD [20]. Mini-batch gradient descent works by updating weights on the basis of the gradient accumulated over a fixed size b ($b << n$) of training data:

$$\mathbf{w}(k) = \mathbf{w}(k-1) - \eta \frac{\partial \mathbb{E}_b}{\partial \mathbf{w}}(\mathbf{w}(k-1))$$

$$= \mathbf{w}(k-1) - \eta \frac{1}{b} \sum_{i=1}^{b} \frac{\partial \mathbb{E}_1}{\partial \mathbf{w}}(\mathbf{z_i}, \mathbf{w}(k-1)).$$

Where $\mathbf{w}$ is the weights vector (including the bias), η is the learning rate, $\mathbb{E}_1$ is the loss function of a particular data point $\mathbf{z}$ and $\mathbb{E}_b$ is the empirical risk of b data examples.

In summary, the multilayer neural networks used in Chapter 3 of this thesis pass input values to calculate net inputs (net_j) and the output a_j of each neuron. Then we propagate the error back on each weight or bias with the backpropagation algorithm. The models are then updated by various gradient descent algorithms.

Multilayer NN as Universal Approximators

One application of multi-layer neural networks is to approximate any continuous mapping g. In such applications, the outputs of neural networks take values from the range of g.

The Kolmogorov mapping neural networks existence theorem (1957) [21] has shown that "a feedforward neural networks with an input layer, a hidden layer, and an output layer can represent any continuous function".

In 1989, George Cybenko proved the universal approximation theorem for sigmoid activation functions [22]. This theorem shows that "a feed-forward network with a single hidden layer containing a finite number of sigmoid neurons, can approximate continuous functions on compact subsets of $\mathbb{R}^n$".

Kurt Hornik showed in 1991 [23] that the universal approximation theorem works for arbitrary bounded and non-constant activation functions. The author proposed that "the multilayer feedforward architecture itself gives neural networks the potential of being universal approximators instead of the choices of activation functions".

Chapter 2

Higher Order Neural Networks

In the literature, neural networks with linear shaping functions are the most common models in the field of pattern recognition. The popularity of linear shaping functions are due to their easy differentiation and fast matrix multiplication [24]. However, as explained previously, the perceptron has limitations of solving non-linearly separable problems. On the other hand, some researcher proposed models based on higher order neural networks for various applications.

Giles and Maxwell [25] were the first to publish a paper on Higher Order Neural Networks (HONNs) in 1987. The first book on HONN was written by Bengtsson [26]. As stated in [25], higher Order Neural Networks contain computing units that are capable of incorporating functions such as polynomial or trigonometric functions.

According to Zurada [27], "a Higher Order Neural Network (HONN) is one in which the neurons can combine their inputs nonlinearly. While HONNs may be more expensive computationally, they have certain advantages over linear networks". For example, higher-order neurons are able to capture higher-order correlations by nonlinear shaping functions. To achieve a similar performance, a first order neural network will need a large number of hidden neurons. In addition, Zurada points out that HONNs may generalize better than the first order neural networks [27].

HONNs are used in applications such as pattern recognition, nonlinear simulation, and classification [24]. It has been demonstrated that HONNs are faster, more accurate, and easier to explain than neural networks with linear shaping functions [26]. Compared to other types of more complex NNs, HONNs converge to the global minimum of the error functions faster, have smaller network sizes and have more accurate curve fittings [24]. The exclusion of hidden layers allows for simpler training methods to be used. For example, the polynomial perceptron learning rule which is demonstrated in this Chapter can be used to update polynomial perceptrons.

In this Chapter, we first introduce the concept of Vapnik-Chervonenkis dimension (VC-dimension). Then we prove the perceptron convergence theorem for perceptrons with polynomial shaping functions via the VC-dimension. We then implement such

polynomial perceptrons with the extended perceptron learning rule. The empirical experiments demonstrate that a single polynomial perceptron learns the XOR problem very efficiently.

2.0.1 Vapnik-Chervonenkis Dimension

Definition 2.1 (Shattering)**.** Let ξ be a family of subsets of a set Ω (a concept class). A finite set $A \subset \Omega$ is *shattered* by ξ if for every subset $B \subset A$, $\exists C \in \xi$ such that

$$B = A \cap C.$$

Definition 2.2 (VC-dimension)**.** The *Vapnik-Chervonenkis dimension* (VC dimension) of a concept class ξ is define as the supremum of cardinalities of finite subset $A \subset \Omega$ shattered by ξ.

Theorem 2.3 (Steele(1975),Dudley(1978))**.** *Let X be a set, and let $\mathcal{F}$ be a vector space consisting of functions on X. i.e. $\mathcal{F} \subset \mathbb{R}^X$. Consider the concept class:*

$$\mathfrak{C} = \{\{x \in X : f(x) \geq 0\} : f \in \mathcal{F}\}$$

suppose $dim\mathcal{F} = d < \infty$, then $VC(\mathfrak{C}) \leq d$.

Proof: Full proof can be found in theorem 13.9 of [28].

∎

Corollary. *Let X be a set, and let $\mathcal{F}$ be a vector space of functions (functions contain no constant terms) on X. i.e. $\mathcal{F} \subset \mathbb{R}^X$. Consider the concept class:*

$$\mathfrak{C} = \{\{x \in X : f(x) \geq c\} : f \in \mathcal{F}, c \in \mathbb{R}\}.$$

Suppose $dim\mathcal{F} = d < \infty$, then $VC(\mathfrak{C}) \leq d + 1$.

Proof: The result can be easily deduced from theorem 2.3 by adding constant terms in the functions of $\mathcal{F}$, then apply theorem 2.3. Suppose $\mathcal{F}_0$ is the vector space of functions without constant terms and $\mathcal{F}_1$ is the vector space of functions after adding constant terms, then $dim\mathcal{F}_0 = dim\mathcal{F}_1 - 1$. Applying theorem 2.3 directly will give us the result.

∎

2.1 VC Dimension of Perceptrons and Linear Threshold Networks

This section first demonstrates the perceptron learning rule for perceptrons with linear shaping functions. When we combine multiple such perceptrons, a linear threshold network is formed. We then review the perceptron convergence theorem for such perceptrons and linear threshold networks.

2.1.1 VC Dimension for Perceptrons

The concept class generated by a perceptron, for all possible values of parameters, consists of all half spaces in the space of the corresponding dimension [29]. Thus, we first start with a simple example of VC-dimension for half plane in $\mathbb{R}^2$.

Example 2.4. *Let $\mathfrak{C}$ consist of all closed half planes in $\mathbb{R}^2$:*

$$\mathfrak{C} = \{H_{\mathbf{v},b} : \mathbf{v} \in \mathbb{R}^2, b \in \mathbb{R}\}$$

where

$$H = H_{\mathbf{v},b} = \{\mathbf{x} \in \mathbb{R}^2 : \langle \mathbf{x}, \mathbf{v} \rangle \geq b\}.$$

Then $VC(\mathfrak{C}) = 3$.

Remark. The above example can be shown easily. Consider three non-collinear points, we can always find a half plan to separate them according to their corresponding labels. Therefore, they can be shattered by $\mathfrak{C}$. We then show that we can't find a set of four points that can be shattered by $\mathfrak{C}$. Suppose $\mathcal{A} = \{a, b, c, d\}$, then either one of the points falls into the convex hull of the other three (case 1) or the four points form a quadrilateral (case 2). In case 1, suppose a is contained in the convex hull formed by $\{b, c, d\}$, then $\{b, c, d\} \subset \mathcal{A}$ can not be carved out by $\mathfrak{C}$. In case 2, any two points on opposite vertices cannot be the intersection of a half plane with $\mathcal{A}$. One example of case 2 is the XOR problem.

Theorem 2.5 ([30])**.** *Let $\mathfrak{C}$ consist of all closed half spaces in the vector space $\mathbb{R}^n$:*

$$\mathfrak{C} = \{H_{\mathbf{v},b} : \mathbf{v} \in \mathbb{R}^n, b \in \mathbb{R}\}$$

where

$$H = H_{\mathbf{v},b} = \{\mathbf{x} \in \mathbb{R}^n : \langle \mathbf{x}, \mathbf{v} \rangle \geq b\}.$$

Then $VC(\mathfrak{C}) = n + 1$.

Proof: We first show $VC(\mathfrak{C}) \geq n + 1$. Thus, we show there exists a set $\mathcal{A}$ with $|\mathcal{A}| = n + 1$, which can be shattered by $\mathfrak{C}$.

Let's pick the $n+1$ points as $\mathcal{A} = \{0, e_1, e_2, ..., e_n\}$ where e_i is the i-th standard basic coordinate vector in $\mathbb{R}^n$. For any subset $\mathcal{B} \subset \mathcal{A}$, we can define such a vector $\mathbf{v} \in \mathbb{R}^n$ as:

$$\mathbf{v}^{(i)} = \begin{cases} 1, & \text{if} \quad e_i \in \mathcal{B} \\ -1, & \text{if} \quad e_i \notin \mathcal{B}. \end{cases} \tag{2.1.1}$$

Now, $\langle \mathbf{e_i}, \mathbf{v} \rangle = 1$ if $\mathbf{e_i} \in \mathcal{B}$, $\langle \mathbf{0}, \mathbf{v} \rangle = 0$ and $\langle \mathbf{e_j}, \mathbf{v} \rangle = -1$ if $\mathbf{e_j} \in \mathcal{A} \backslash \mathcal{B}$. We can set $b = -1/2$ if $0 \in \mathcal{B}$ and $b = 1/2$ if $0 \notin \mathcal{B}$. In both cases, we can see that $\mathcal{B}$ is separated by the half plane defined as above.

We then need to show $VC(\mathfrak{C}) \leq n+1$. This is an immediate result of Theorem 2.3 if we take $\mathcal{F}$ to be the vector space spanned by the following functions

$$g_1(x) = x^{(1)}, \quad g_2(x) = x^{(2)}, \quad ..., \quad g_n(x) = x^{(n)}, \quad \text{and} \quad g_{n+1}(x) = 1$$

where $x^{(i)}$ is the i-th components of the vector $\mathbf{x}$.

More detailed proof can be found in Example 6.10 of [29]. ∎

Corollary. *The VC dimension of the concepts generated by a perceptron is d+1 where d is the number of inputs.*

Proof: The concept class associated to a perceptron consists of all combinations of affine functions with the Heaviside function, which is basically all indicator functions of half spaces. As we have shown, the VC dimension of half spaces of $\mathbb{R}^d$ is $d+1$. ∎

2.1.2 VC Dimension for Linear Threshold Neural Networks

Now, consider a more general neural network with multiple perceptrons. In a multi-layer network, the neurons are grouped by layers l, labeled with integers:

$$l(i) = 0, 1, \ldots, L.$$

Note that every multi-layer network has the property that no units in the same layer are connected to each other. The output units are units in the top layer L.

If all the shaping functions in a multilayer network are affine functions and all activation functions are equal to the Heaviside function, then the network $\mathcal{N}$ is called a *linear threshold network*. It can be shown that the VC-dimension of a linear threshold network has an upper bound.

Theorem 2.6. *Let $\mathcal{N}$ be a feed-forward linear threshold network having W adjustable parameters (including weights and biases) and k computational units. Then*

$$VC(\mathcal{N}) \leq 2W \log_2 \left(\frac{2k}{\log 2} \right).$$

Proof. Full proof can be found in theorem 6.1 of [31] using Sauer-Shelah Lemma (theorem 2.7). A similar proof can also be found in the case of polynomial neural network in theorem 2.10.

Theorem 2.7 (Sauer-Shelah Lemma)**.** *Let $\mathscr{C}$ be a family of subsets of some set with a finite VC dimension, i.e. $VC(\mathscr{C}) \leq d$. Then whenever $d \leq n$,*

$$s(\mathscr{C}, n) \leq \left(\frac{en}{d} \right)^d.$$

Remark. For a natural number n and a family of subsets $\mathscr{C}$, we denote the n-th shattering coefficient as

$$s(\mathscr{C}, n) = max\{\#(\mathscr{C}|_{\sigma_u}) : \sigma_u \subset \Omega, |\sigma_u| \leq n\}.$$

Here

$$\mathscr{C}|_{\sigma_u} = \{C \cap \sigma_u : C \in \mathscr{C}\}$$

is the set of all possible labellings induced on an n-sample σ_u by the concepts from $\mathscr{C}$. For example, if $VC(\mathscr{C}) = d$, then one has the d-th shattering coefficient

$$s(\mathscr{C}, d) = 2^d.$$

Moreover, the VC dimension can be expressed via the shattering coefficients as follows:

$$VC(\mathscr{C}) = sup\{n \in \mathbb{N} : s(\mathscr{C}, n) = 2^n\}.$$

2.2 VC Dimension for Polynomial Perceptrons and Higher Order Neural Networks

So far, we demonstrated that perceptrons and linear threshold networks with linear shaping functions have nice properties: finite VC dimension and converge if the feature space is linearly separable. These nice properties can also be generalized to neural networks with polynomial shaping functions as shown in the next sections.

2.2.1 VC Dimension for Polynomial Perceptrons

Definition 2.8. Let R be any commutative ring, and let $A = R[x_1, \ldots, x_n]$ be a polynomials ring over A in n variables. A monomial in A is defined as $x_1^{\alpha_1} \cdots x_n^{\alpha_n}$ where $\alpha_i \in \mathbb{N}$. The degree of a monomial is defined as the sum of all exponents: $\deg(x_1^{\alpha_1} \cdots x_n^{\alpha_n}) = \alpha_1 + \cdots + \alpha_n$. For instance, a polynomials ring $\mathbb{Z}[x, y]$ has four "degree three" monomials: x^3, x^2y, xy^2, and y^3.

Proposition 2.2.1 ([32]). *The number of monomials with n variables in $A = R[x_1, \ldots, x_n]$ of degree d is $\binom{n+d-1}{d}$.*

Proof: This is equivalent as counting how many ways to put d balls into n boxes. The trick here is to consider each variable as a box, then the degree of each variable can be represented as the same number of balls in the box. For example, consider two variables $n = 2$ with degree three $d = 3$, we can arrange the balls in the box like $\bullet \mid \bullet\ \bullet$ and this is equivalent to $x_1^1 x_2^2$ in terms of monomials.

Then, we count how many ways to arrange the position of $n - 1$ bars and d balls. This is equivalent as choosing d of $n - 1 + d$ symbols to be dots. There are $\binom{n+d-1}{d}$ ways to do this task and therefore the number of monomials in A of degree d is $\binom{n+d-1}{d}$, which is the same as putting d balls into n boxes. ∎

Proposition 2.2.2 ([32]). *The number of monomials with n variables in $A = R[x_1, \ldots, x_n]$ of degree less than or equal to d is $\binom{n+d}{d}$.*

Proof: This result could be easily proved based on the previous proposition by adding a dummy variable 1 in the monomials. Consider a monomial of degree $k \leq d$, we define the degree of 1 as $d-k$. The the total degree of this newly created monomial is d again. For example, a degree three monomial $x_1^2 x_2$ can be expanded to a degree 5 monomial $x_1^2 x_2 x_3^2$. Therefore, we are counting how many ways we could put d balls into $n + 1$ variables, which is $\binom{n+d}{d}$. Thus, there are $\binom{n+d}{d}$ monomials of degree less than or equal to d in n variables. ∎

Theorem 2.9. *Let $\mathbb{R}[x_1, x_2, \ldots, x_n]_{\leq d}$ denote the set of all real polynomials in n variables of degree at most d. Define*

$$\mathcal{P}_{n,d} = \{\{x : p(x) \geq 0\} : p \in \mathbb{R}[x_1, x_2, \ldots, x_n]_{\leq d}\}.$$

Then $VC(\mathcal{P}_{n,d}) \leq \binom{n+d}{d}$.

Proof: **Method 1:** It is obvious that the set of all real polynomials in n variables of degree at most d form a vector space with dimension $\binom{n+d}{d}$. By Theorem 2.3,

$VC(\mathcal{P}_{n,d}) \leq dim(\mathbb{R}[x_1, x_2, \ldots, x_n]_{\leq d}) = \binom{n+d}{d}$.

Method 2: The idea here is to construct a lifting map to a space of higher dimension in which each monomial corresponds to one dimension. Now the polynomial can be expressed as a linear combination of the expanded dimensions and the usual result for half spaces in the resulting space can be used to decide the VC dimension. From this method, we can see that the upper bound $\binom{n+d}{d}$ of the VC dimension is exact.

Let M to be the set of all non-constant monomials of degree at most d with n variables. For instance, with $n = d = 2$, we have $M = \{x_1, x_2, x_1x_2, x_1^2, x_2^2\}$. Let $|M| = m$ be the number of such monomials of degree at most d with n variables. Then we use m as the index for the monomials in co-domain. For example, in the above example, there are five monomials in total and we index them as $y_1 = x_1, y_2 = x_2$, $y_3 = x_1x_2$, $y_4 = x_1^2$, $y_5 = x_2^2$.

We can then define a map $\varphi : \mathbb{R}^n \to \mathbb{R}^m$ by $\varphi(x)_\mu = \mu(x)$ where μ indicates which monomial is used for the corresponding coordinate. For example, with $n = d = 2$, the map is

$$\varphi(x_1, x_2) \in \mathbb{R}^2 \mapsto (x_1, x_2, x_1x_2, x_1^2, x_2^2) \in \mathbb{R}^5.$$

Obviously, φ is injective. we show that if $A \subset \mathbb{R}^n$ is shattered by $\mathcal{P}_{n,d}$, then $\varphi(A)$ is shattered by half-planes in $\mathbb{R}^m$.

Suppose $B \subset A$ and $p = a_0 + \sum_1^m a_i\mu_i \in \mathcal{P}_{n,d}$, p map points in B to be nonnegative and points in $A \backslash B$ to be negative. Define the half plane

$$H_{\mathbf{v},b} = \{\mathbf{y} \in \mathbb{R}^m : \langle \mathbf{y}, (a_1, \ldots, a_m)\rangle + a_0 \geq 0\}.$$

Then $\varphi(A) \cap H_{\mathbf{v},b} = \varphi(B)$. Since φ is injective, we have a set of size $|A|$ in $\mathbb{R}^m$ that can be shattered by half planes. Therefore, $\text{VC}(\mathcal{P}_{n,d}) = VC(H_{\mathbf{v},b} \in \mathbb{R}^m) \leq m+1 = \binom{n+d}{d}$.
∎

Remark. Theorem 2.9 shows the upper bound on the VC dimension of a polynomial perceptron exist.

2.2.2 VC Dimension for Higher Order Neural Networks

Similarly to linear threshold networks, we could stack multiple polynomial perceptrons into a multilayer neural network. Such multilayer neural networks are called *higher order neural networks with polynomial shaping functions*. We show here that higher order neural networks with polynomial shaping functions have finite VC dimensions. If we fix the structures of neural networks, the VC dimension of a HONN

with polynomial shaping functions is larger than that of a linear threshold network. This indicates a larger capacity of higher order neural networks.

Theorem 2.10. *Let $\mathcal{N}$ be a feed-forward network with polynomial shaping functions of degree d and W adjustable parameters (coefficients for polynomials) and k computation units. Then for every $n \geq N$*

$$VC(\mathcal{N}) \leq 2W \log_2 \left(\frac{2k}{\log 2} \right).$$

In order to prove theorem 2.10, we first need to introduce a few notions, such as 2.11 and 2.12.

Definition 2.11. Let X be a random variable taking finitely many different values $x_1, x_2, ..., x_n$ with probabilities $p_1, ..., p_n$, respectively. The entropy of X is defined as

$$H(X) = \sum_{i=1}^{n} -p_i \log p_i.$$

Theorem 2.12 ([30])**.** *The maximum value of the entropy of a random variable X defined as the above equation is achieved when all p_i are equal. In other words,*

$$P(X = x_i) = \frac{1}{n}, i = 1, 2, 3, \ldots$$

and

$$\max H(X) = \log(n).$$

Proof: It is well known that the logarithm function is concave in its domain. This means that $\log(tx + (1-t)y) \geq t\log(x) + (1-t)\log(y)$ for all $x, y > 0$ and $t \in [0, 1]$. Since $\sum_{i=1}^{n} p_i = 1$, by finite induction, for all $x_i > 0$, we have

$$\log \left(\sum_{i=1}^{n} p_i x_i \right) \geq \sum_{i=1}^{n} p_i \log(x_i).$$

Substitute $x_i = \frac{1}{p_i}$, we get

$$\log(n) \geq \sum_{i=1}^{n} p_i \log(p_i)^{-1} = H(X).$$

Moreover, when $p_i = \frac{1}{n}$, the equality holds as the following:

$$H(X) = \sum_{i=1}^{n} p_i \log(p_i)^{-1} = \sum_{i=1}^{n} \frac{1}{n} \log(n) = \log(n).$$

Proof (Theorem 2.10)**.** Define a partial order on all neurons by $1, 2, ..., i$ such that $i < j$ if i is connected to j. Let f_i be a binary function of neuron i with outputs in $\{0, 1\}$. We associate the function f_i^w with a particular state $w \in \mathbb{R}^W$. Let $d_i = fan - in(i)$ and therefore $\sum_1^k d_i = W$. Moreover, we can see that $d_i = \binom{n_i+d}{d}$ for a polynomial function with n_i inputs of degree at most d.

Fix a data sample $\sigma \subset \Omega^n \times \{0, 1\}^n$, where each data point $\mathbf{x} \in \mathbb{R}^m$. Thus, we have a fixed n data points and each point has m variables (inputs).

Pick arbitrary $i \in \{1, 2, ..., k\}$. Two states w and w' are indistinguishable up to i if $f_j^w(x) = f_j^{w'}(x)$ for all $x \in \sigma$ and for all $j \leq i$. We denote this relation by $w \sim_i w'$. In fact, $\sim_i$ is an equivalence relation on the state space $\mathbb{R}^W$ for every $i = 1, 2, ..., k$. This equivalent relation induces a series of partitions γ_i on $\mathbb{R}^m$ based on the partial order i. One can easily check that if $i < j$, then γ_j refines γ_i.

Since there are k units in the network and γ_j refines γ_i if $i < j$, then the number of pairwise distinct functions based on the states f_k^w $(w \in \mathbb{R}^m)$ is bounded by the size of the finest partition γ_k. What we need to compute is the bound of γ_k. We could obtain this bound by recursively computing γ_i starting from γ_1.

Two states w and w' are distinguishable at $i = 1$ iff for some $x \in \sigma$, $f_1^w \neq f_1^{w'}$. Thus, we have

$$|\gamma_1| = |\{f_{1|\sigma}^w : w \in \mathbb{R}^w\}|.$$

We proved earlier in 2.9 that the VC dimension for a polynomial perceptron (polynomial shaping function + Heaviside) with n_i variables (inputs) of degree at most d is $\binom{n_i+d}{d} = d_i$, then using the Sauer-Shelah lemma, we have

$$|\gamma_1| \leq \left(\frac{en}{d_i}\right)^{d_i}.$$

In the inductive step, suppose we have estimated the upper bound for γ_{i-1}. Consider one equivalence class C in the partition γ_{i-1}, we need to find the upper bound of how many different equivalence sets $\sim_i$ can divide in. We know that if $w, w' \in C$, then f_{i-1}^w equals $f_{i-1}^{w'}$ for all $x \in \sigma$. So only the d_i parameters added by neuron i will determine f_i^w and $f_i^{w'}$. By Sauer-Shelah lemma, C can be divide in at most

$$\left(\frac{en}{d_i}\right)^{d_i}$$

new pieces. Each of the pieces corresponds to a set of functions $f^w_{i|\sigma}$. Thus,

$$|\gamma_i| \leq |\gamma_{i-1}| \cdot \left(\frac{en}{d_1}\right)^{d_1}$$

and therefore

$$|\mathscr{C}_{\mathscr{N}}|_\sigma| \leq |\gamma_k| \leq \prod_{i=1}^{k} \left(\frac{en}{d_i}\right)^{d_i} = \left(\frac{en}{d_1}\right)^{d_1 \cdot k} \tag{2.2.1}$$

since all of the computation units take the same number of parameters (all units take the same number of input variables n_i).

Take the logarithm of 2.2.1 on both side of the above equation. Since $\sum_{i=1}^k d_i = W$ and all d_i are equal, we obtain $d_1 \cdot k = W$. Thus, $d_1 = W/k$. We can then write the upper bound for the n-th shattering coefficient as:

$$\log\ s(\mathscr{N}, n) \leq d_1 \cdot k\ \log \frac{enk}{W} = W\ \log \frac{enk}{W} \tag{2.2.2}$$

If not all d_i are equal, we can still take the logarithm of 2.2.1, then we get

$$\log\ s(\mathscr{N}, n) \leq \sum_{i=1}^{k} d_i \log\ \left(\frac{en}{d_i}\right).$$

We know $\sum_{i=1}^n d_i = W$, so we can treat d_i/W as probabilities. The maximum of the entropy is achieved when d_i/W are equal for all $i = 1, ..., k$ (proof in 2.12). Thus we get the same conclusion as equation 2.2.2.

If now

$$n > W \log_2 \left(\frac{enk}{W}\right) \geq W\ \log \frac{enk}{W}, \tag{2.2.3}$$

then

$$2^n > \left(\frac{enk}{W}\right)^W \geq s(\mathscr{N}, n).$$

Since $s(\mathscr{N}, d) = 2^d$ for all $d \leq VC(\mathscr{N})$, we have

$$VC(\mathscr{N}) < n$$

.

Next, we need to check when does 2.2.3 hold. More specifically, we want to find the lower bound for n such that

$$VC(\mathscr{N}) < n \text{ and } VC(\mathscr{N}) \leq n - 1. \tag{2.2.4}$$

We can use the lemma that for all $\alpha, x > 0$,

$$\log x \leq \alpha x - \log \alpha - 1. \tag{2.2.5}$$

Let

$$x = \frac{enk}{W}, \alpha = \frac{\log 2}{2ek}$$

based on 2.2.5 we get

$$\log \frac{enk}{W} \leq \frac{\log 2}{2ek}\frac{enk}{W} - \log(\frac{\log 2}{2ek}) - 1$$

which is equivalent to,

$$\log \frac{enk}{W} \leq \frac{\log 2n}{2W} - \log(\frac{\log 2}{2ek}) - 1$$

replace 1 by log e and rearrange the terms, we get

$$\log \frac{enk}{W} \leq \frac{\log 2n}{2W} + \log(\frac{2ek}{\log 2}) - \log\ e = \frac{\log 2n}{2W} + \log(\frac{2k}{\log 2}).$$

We then multiply W on both side of the above equation and change of base of the logarithm to 2:

$$W \log_2 \frac{enk}{W} \leq \frac{n}{2} + W \log_2(\frac{2k}{\log 2}).$$

Then 2.2.3 follows whenever

$$W \log_2(\frac{2k}{\log 2}) < \frac{n}{2},$$

thus,

$$n > 2W \log_2(\frac{2k}{\log 2}).$$

We can find a such n by setting $n = floor(2W \log_2(\frac{2k}{\log 2})) + 1$. Thus,

$$n > 2W \log_2(\frac{2k}{\log 2}) \geq n - 1.$$

Combine this with 2.2.5, we get,

$$VC(\mathcal{N}) \leq n - 1 \leq 2W \log_2(\frac{2k}{\log 2}).$$

Remark. Note that theorem 2.6 and theorem 2.10 has the same upper bound of VC

dimension as $2W\log_2(\frac{2k}{\log 2})$. However, the bound is very different as W is not the same in the two cases. In linear networks (theorem 2.6), if we have a 3 layers fully connected feed-forward linear network with k computational nodes (all in hidden layer) and d inputs, $W = (d+1)\cdot k$. In polynomial networks (theorem 2.10), if we have a 3 layers fully connected feed-forward polynomial network with k computational nodes (all in hidden layer) and d inputs of degree at most d_i, $W = \binom{d+d_i}{d_i}\cdot k$.

2.3 Polynomial Perceptron Convergence Theorem

In this section, we want to generalize the perceptron convergence theorem to polynomial perceptrons. We need the assumption that the training dataset is separable by a polynomial perceptron. Therefore, there exists a state that we can classify all points correctly. As demonstrated in the later section, the XOR dataset is separable by a polynomial perceptron under finite steps of learning.

First, we describe the perceptron learning rule for polynomial perceptrons.

Algorithm description: Fix a positive $\eta > 0$ (a learning constant). Let $\sigma \subset \Omega^n \times \{0,1\}^n$ ($\Omega = \mathbb{R}^m$) be a training sample.

Suppose the polynomial perceptron is the composition of a polynomial shaping function with m variables of d degree and a Heaviside function. Then the polynomial perceptron has $\binom{m+d}{d}$ parameters.

Consider the current state $\theta_s \in \mathbb{R}^{\binom{m+d}{d}}$. We still initialize everything to be zero at the beginning:

$$\theta_{s_1} = 0 \in \mathbb{R}^{\binom{m+d}{d}}.$$

Then we update the state sequentially as the algorithm cycle through the training dataset.

The algorithm is updated as the following: consider an element $(\mathbf{x}, y) \in \sigma$, first $\mathbf{x}$ is mapped to $\mathbf{x}' \in \mathbb{R}^{\binom{m+d}{d}}$ by a lifting map as defined in the method 2 in the proof of Theorem 2.9.

Then the algorithm computes the output values at $\mathbf{x}'$,

$$f^{\theta_s}(\mathbf{x}') = a(\langle \theta_s, \mathbf{x}' \rangle) \tag{2.3.1}$$

and update the state, $\theta_{s_i} \mapsto \theta_{s_{i+1}}$, as follows:

$$\theta s_{i+1} = \theta_{s_i} + \eta(y - f^{\theta_{s_i}}(\mathbf{x}'))\mathbf{x}'. \tag{2.3.2}$$

If a point is classified correctly, no update is made. If the pair $(\mathbf{x}, y)$ is misclassified, θ_s is being moved a bit either towards $\mathbf{x}'$ or away from $\mathbf{x}'$ depends on the true label of $\mathbf{x}$.

The algorithm stops when we cycle through the whole dataset without making any mistakes. Thus, if $g(\mathbf{x}) = f^{\theta_s}(\varphi(\mathbf{x}))$ classifies all $\mathbf{x}$ from the sample dataset correctly. Then the network keeps the values of the current state θ_s as the final trained parameter values.

Definition 2.13 (Margin)**.** The minimal distance from the separation hyperplane (induced by a classifier) to an arbitrary data point is called the *margin*, which is defined as:

$$\gamma = min_{(\mathbf{x},y)\in\sigma}|\langle\theta_s^*, \mathbf{x}'\rangle|. \tag{2.3.3}$$

Theorem 2.14. *(Polynomial Perceptron Convergence Theorem) Let $\sigma \subset \Omega^n \times \{0,1\}^n$ ($\Omega = \mathbb{R}^m$) be a training sample.*

Suppose the polynomial perceptron consists of a polynomial shaping function with m variables of d degree and a Heaviside activation function. Then the polynomial perceptron has $\binom{m+d}{d}$ parameters and consider all the parameter as a state vector $\theta_s \in \mathbb{R}^{\binom{m+d}{d}}$.

Let M be the set of all monomials of degree at most d with m variables. Set $m' = |M| = \binom{m+d}{d}$ and let the coordinates in $\mathbb{R}^{m'}$ to be indexed by the monomials in M. Define a map $\varphi : \mathbb{R}^m \to \mathbb{R}^{m'}$ ($\mathbf{x} \mapsto \mathbf{x}'$) by $\varphi(\mathbf{x})_\mu = \mu(\mathbf{x})$.

Suppose there exists a state θ_s^* with

$$||\theta_s^*|| = 1 \tag{2.3.4}$$

and $g(\mathbf{x}) = f^{\theta_s^*}(\varphi(\mathbf{x})) = f^{\theta_s^*}(\mathbf{x}') = y$ for all $(\mathbf{x}, y) \in \sigma$.

Suppose $\gamma > 0$ and let R be the radius of a ball in $\mathbb{R}^{\binom{m+d}{d}}$ centered at zero which contains the unlabeled datasets $\sigma'_u = \{\mathbf{x}'|(\mathbf{x}, y) \in \sigma\}$:

$$R = \max_{\mathbf{x}'\in\sigma'_u}||\mathbf{x}'||. \tag{2.3.5}$$

Then for every learning constant $\eta > 0$ the perceptron learning algorithm converges in no more than

$$\frac{R^2}{\gamma^2}$$

steps to a state θ_s which is consistent with the training data:

$$\forall(\mathbf{x}', y) \in \sigma', f^{\theta_s}(\mathbf{x}') = y.$$

Proof. Suppose the network is already updated i times. After passing a data point $(\mathbf{x}, y) \in \sigma$ ($(\mathbf{x}', y) \in \sigma'$ where $\mathbf{x}' = \varphi(\mathbf{x})$), we need to make the $i + 1$th update as $\theta s_i \mapsto \theta_{s_{i+1}}$. Thus, point $\mathbf{x}$ is misclassified,

$$g(\mathbf{x}) = f^{\theta s_i}(\mathbf{x}') \neq y$$

or

$$f^{\theta_{s_i}}(\mathbf{x}') = 1 - y.$$

Then we update the states by

$$\begin{aligned} \theta_{s_{i+1}} &= \theta_{s_i} + \eta(y - f^{\theta_{s_i}}(\mathbf{x}'))\mathbf{x}' \\ &= \theta_{s_i} + \eta(2y-1)\mathbf{x}'. \end{aligned} \tag{2.3.6}$$

Now, we estimate the lower bound of $||\theta s_{i+1}||$:

$$\begin{aligned} \langle \theta_s^*, \theta_{s_{i+1}} \rangle &= \langle \theta_s *, \theta_{s_i} + \eta(2y-1)\mathbf{x}' \rangle \\ &= \langle \theta_s^*, \theta_{s_i} \rangle + \eta(2y-1)\langle \theta_s^*, \mathbf{x}' \rangle \\ &= \langle \theta_s^*, \theta_{s_i} \rangle + \eta|\langle \theta_s^*, \mathbf{x}' \rangle| \\ &\geq \langle \theta_s^*, \theta_{s_i} \rangle + \eta\gamma. \end{aligned} \tag{2.3.7}$$

Note that the third line in the above equation is true because $f^{\theta_s^*}$ classifies all $\mathbf{x}' \in \sigma_u'$ correctly. If $y = 1$, then $a(\langle \theta_s^*, \mathbf{x}' \rangle) = 1$ implies $\langle \theta_s^*, \mathbf{x}' \rangle \geq 0$. Therefore, $(2y - 1)\langle \theta_s^*, \mathbf{x}' \rangle = 1 \cdot \langle \theta_s^*, \mathbf{x}' \rangle \geq 0$. If $y = 0$, then $a(\langle \theta_s^*, \mathbf{x}' \rangle) = 0$ implies $\langle \theta_s^*, \mathbf{x}' \rangle < 0$. Therefore, $(2y-1)\langle \theta_s^*, \mathbf{x}' \rangle = -1 \cdot \langle \theta_s^*, \mathbf{x}' \rangle \geq 0$. In both cases, $(2y-1)\langle \theta_s^*, \mathbf{x}' \rangle = |\langle \theta_s^*, \mathbf{x}' \rangle|$. The forth line of equation 2.3.7 is coming from the definition of the classification margin as equation 2.3.3.

By induction on i, we conclude that

$$\langle \theta_s^*, \theta_{s_{i+1}} \rangle \geq i\eta\gamma. \tag{2.3.8}$$

Then using Cauchy-Schwarz inequality, we get

$$||\theta_s^*|| \cdot ||\theta_{s_{i+1}}|| \geq |\langle \theta_s^*, \theta_{s_{i+1}} \rangle|.$$

Since $||\theta_s^*|| = 1$, one get

$$||\theta_{s_{i+1}}|| \geq i\eta\gamma,$$

and therefore,

$$||\theta_{s_{i+1}}^2|| \geq i^2\eta^2\gamma^2. \tag{2.3.9}$$

Then we estimate the upper bound for $||\theta_s^*||$:

$$\begin{aligned}||\theta_{s_{i+1}}||^2 &= \langle\theta_{s_{i+1}}, \theta_{s_{i+1}}\rangle \\ &= \langle\theta_{s_i} + \eta(2y-1)\mathbf{x}', \theta_{s_i} + \eta(2y-1)\mathbf{x}'\rangle \\ &= \langle\theta_{s_i}, \theta_{s_i}\rangle + 2\eta(2y-1)\langle\mathbf{x}', \theta_{s_i}\rangle + \eta^2(2y-1)^2\langle\mathbf{x}', \mathbf{x}'\rangle \\ &= ||\theta_{s_i}||^2 + 2\eta(2y-1)\langle\mathbf{x}', \theta_{s_i}\rangle + \eta^2(2y-1)^2||\mathbf{x}'||^2 \\ &\leq ||\theta_{s_i}||^2 + \eta^2||\mathbf{x}'||^2 \\ &\leq ||\theta_{s_i}||^2 + \eta^2 R^2.\end{aligned} \tag{2.3.10}$$

In line 5 of the above equation, we first used the observation that $(2y-1)^2 = 1$, so $\eta^2(2y-1)^2||\mathbf{x}'||^2 = \eta^2||\mathbf{x}'||^2$. Then since $\mathbf{x}$ is misclassified with θ_{s_i}, we could conclude $2\eta(2y-1)\langle\mathbf{x}', \theta_{s_i}\rangle \leq 0$. This can be shown by cases. If $y = 0$, then $f^{\theta_{s_i}}(\mathbf{x}') = 1$. Therefore, $(2y-1) = -1$ and $\langle\mathbf{x}', \theta_{s_i}\rangle \geq 0$. Hence, $2\eta(2y-1)\langle\mathbf{x}', \theta_{s_i}\rangle \leq 0$. If $y = 1$, then $f^{\theta_{s_i}}(\mathbf{x}') = 0$. Therefore, $(2y-1) = 1$ and $\langle\mathbf{x}', \theta_{s_i}\rangle < 0$.

By using induction on i, one can conclude

$$||\theta_{s_{i+1}}||^2 \leq i\eta^2 R^2. \tag{2.3.11}$$

Combine equation 2.3.8 and equation 2.3.11, we get

$$i^2\eta^2\gamma^2 \leq ||\theta_{s_{i+1}}||^2 \leq i\eta^2 R^2 \tag{2.3.12}$$

and therefore, $i\gamma^2 \leq R^2$. Since $\gamma^2 > 0$, we get

$$i \leq \frac{R^2}{\gamma^2}.$$

Remark. *Note that equation 2.3.12 indicates why the result is independent of learning rate η.*

2.3.1 Solving XOR with a Polynomial Percetron

In the previous section, we proved theoretically that a polynomial perceptron will converge if the dataset is separable by such perceptrons. In this section, we demonstrate that the XOR problem can be solved by a single polynomial perceptron through empirical studies. Note that it has been shown theoretically that the XOR problem can be solved by a degree two polynomial perceptron [33]. In this section, we verify this result through empirical experiments with polynomial perceptrons and polynomial perceptron learning rules (as described in the previous section). Our experiments show that a single polynomial perceptron with degree greater than one has the ability to learn non-linear separable classification problems.

Second Order Polynomial Percetrons for the XOR Problem

In these series of experiments for learning the XOR problem, we use a single neuron with the second order polynomial shaping function and the Heaviside activation function. Thus, given a data point with two inputs $\mathbf{x} = (x_1, x_2)$, the shaping function is $s(\mathbf{x}) = w_1 \times x_1^2 + w_2 \times x_2^2 + w_3 \times x_1 x_2 + w_4 \times x_1 + w_5 \times x_2 + w_6 \times 1$ and the activation function is

$$a(s) = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s \geq 0. \end{cases}$$

We use different learning rates for the polynomial perceptron. Throughout our experiments, we found that the polynomial perceptron is not very sensitive to the learning rate in solving the XOR problem. This is very likely due to the simple structure of the XOR dataset itself. We observed that the second order polynomial perceptron converges in averagely 11 to 12 epochs (iterations). The decision boundary of the second order polynomial perceptron with learning rate 0.1 in each epoch are shown in figures 2.1,2.2,2.3.

Third Order Polynomial Percetrons for the XOR Problem

We also conducted the experiments with third order polynomial perceptrons for learning the XOR problem. We use a single neuron with the third order polynomial shaping function and the Heaviside activation function. Thus, given a data point with two inputs $\mathbf{x} = (x_1, x_2)$, the shaping function is $s(\mathbf{x}) = w_1 \times x_1^3 + w_2 \times x_2^4 + w_3 \times x_1^2 x_2 + w_4 \times x_1 x_2^2 + w_5 \times x_1^2 + w_6 \times x_2^2 + w_7 \times x_1 x_2 + w_8 \times x_1 + w_9 \times x_2 + w_{10} \times 1$ and the activation function is

$$a(s) = \begin{cases} 0 & \text{if } s < 0 \\ 1 & \text{if } s \geq 0. \end{cases}$$

We also experiment with different learning rates for third order polynomial perceptrons. Similarly to second order polynomial perceptrons, we found that third order polynomial perceptrons are not very sensitive to the learning rates in solving the XOR problem. We observed that third order polynomial perceptrons converge in averagely 8 epochs (iterations) which are faster than the convergence of second order perceptrons. The decision boundary of the third order polynomial perceptron with learning rate 0.1 in each epoch are shown in figures 2.4,2.5,2.6.

As we can see from figures 2.1 - 2.6, second order polynomial perceptrons act very similar to third order polynomial perceptrons. In our experiment, we don't have testing data to tackle the problem of over-fitting. However, compare the figure 2.3 with the figure 2.6, we could conclude that the decision boundary from third order polynomial perceptrons are more curvy, which indicates a tendency in over-fitting.

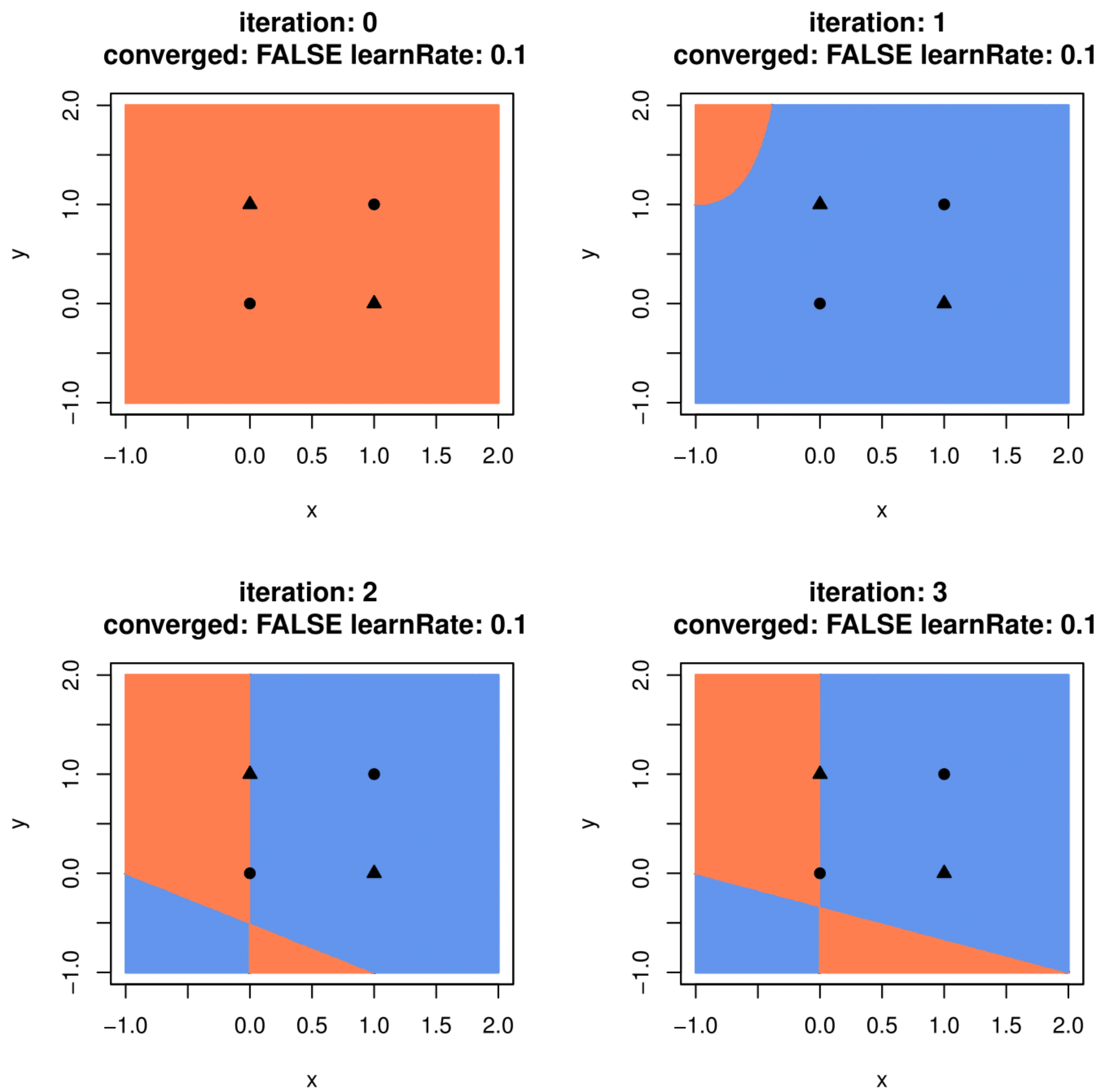

Figure 2.1: The decision boundary of the second order polynomial perceptron in iteration zero to three.

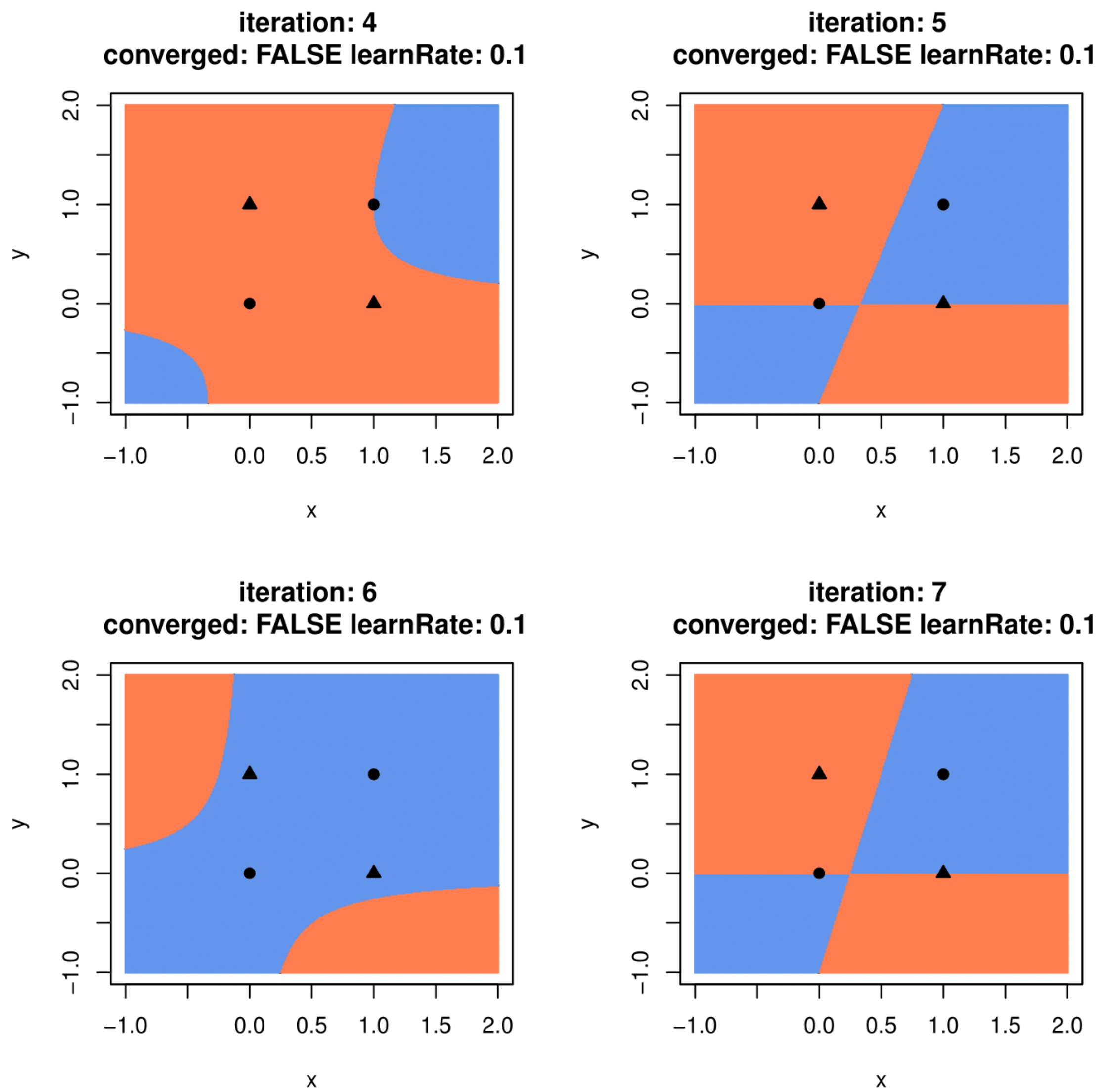

Figure 2.2: The decision boundary of the second order polynomial perceptron in iteration four to seven.

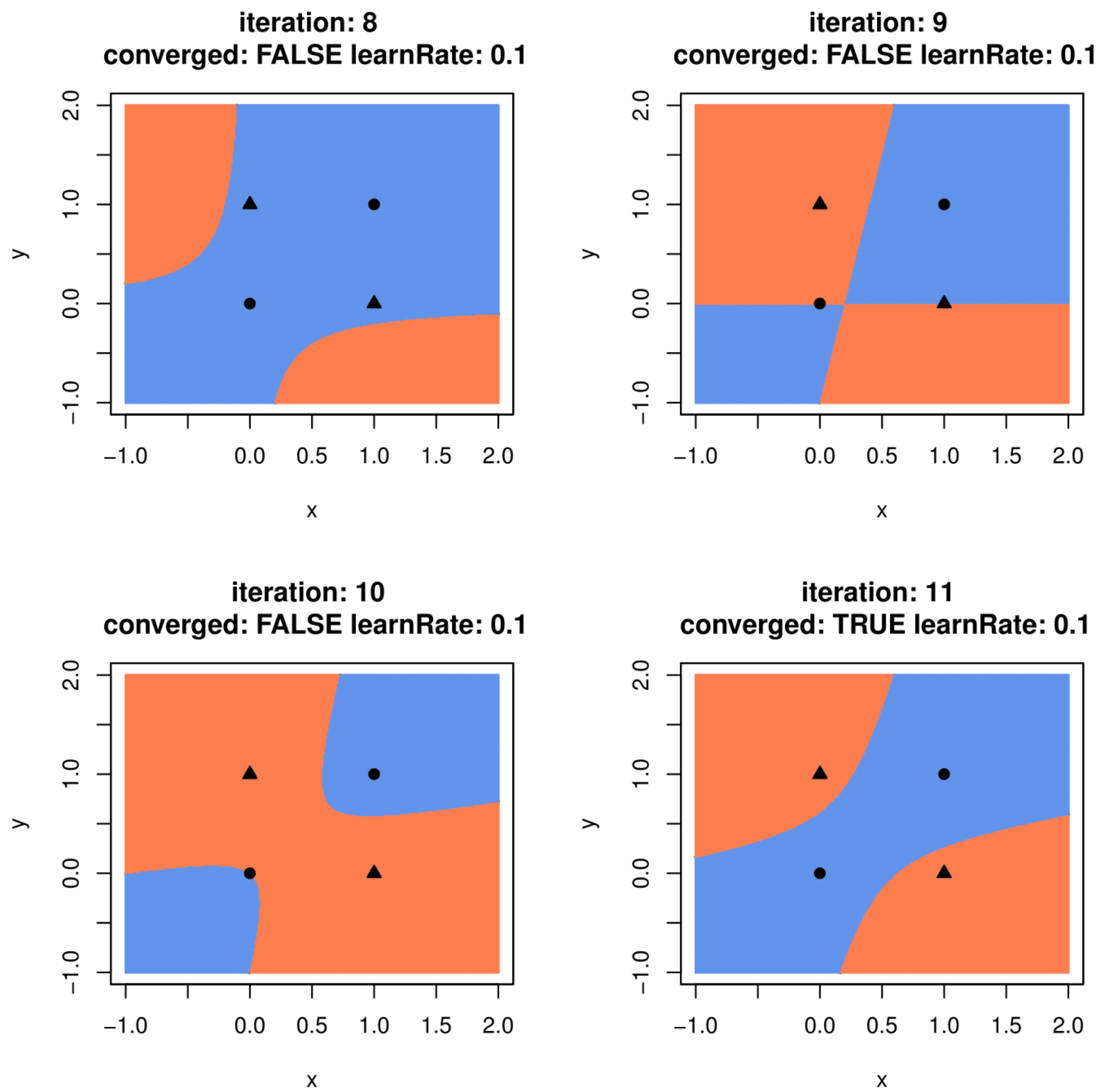

Figure 2.3: The decision boundary of the second order polynomial perceptron in iteration eight to eleven.

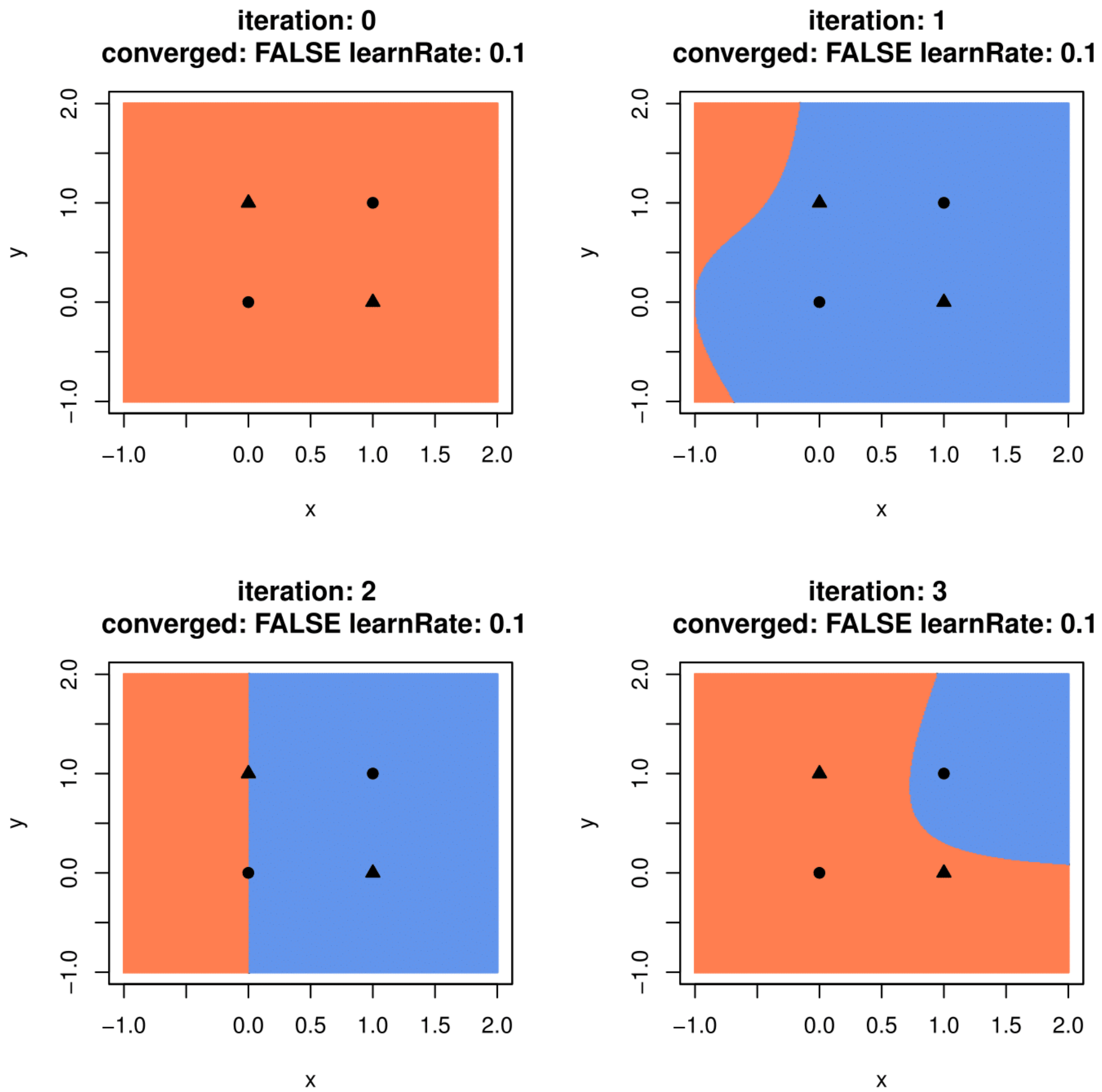

Figure 2.4: The decision boundary of the third order polynomial perceptron in iteration zero to three.

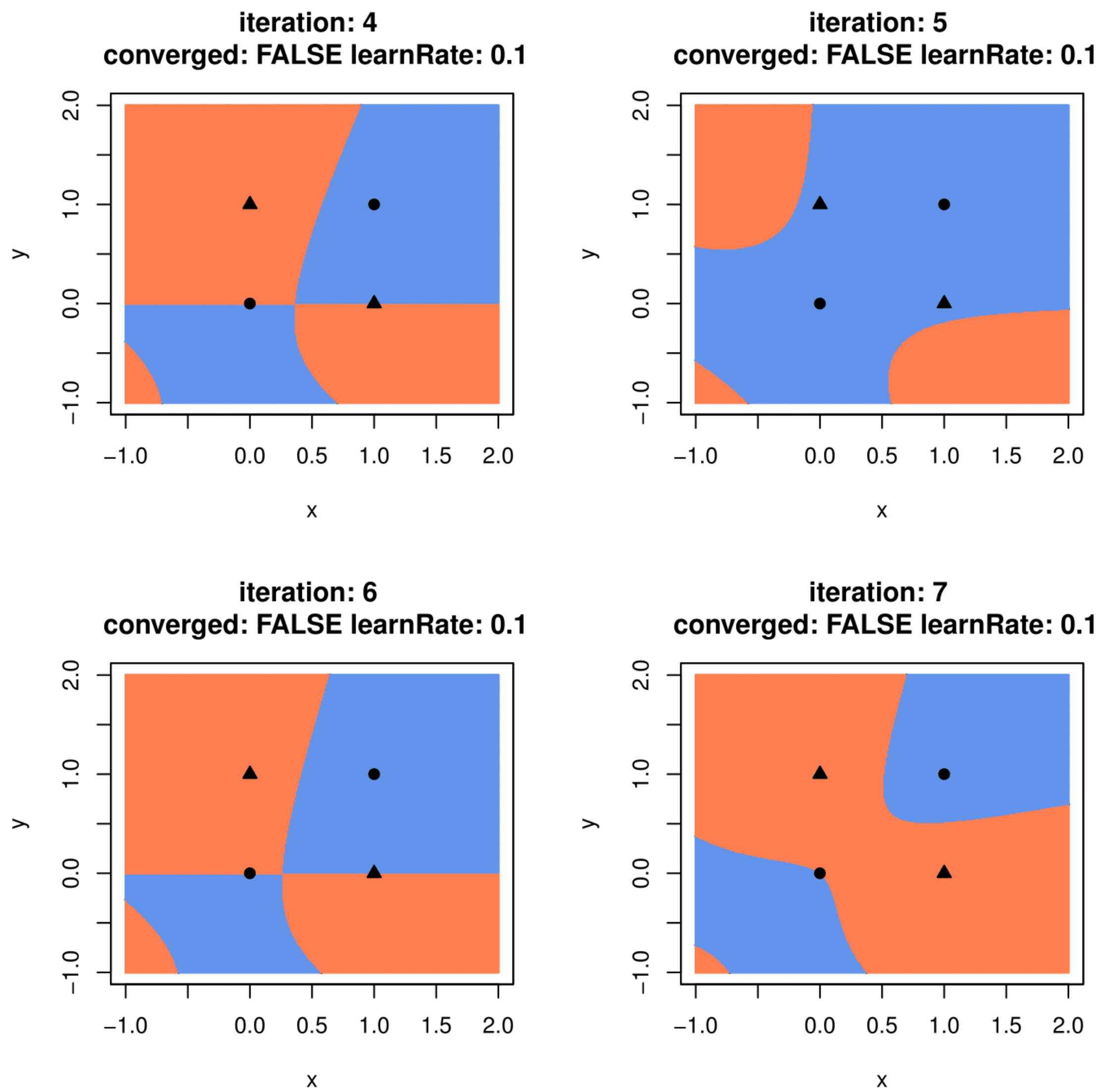

Figure 2.5: The decision boundary of the third order polynomial perceptron in iteration four to seven.

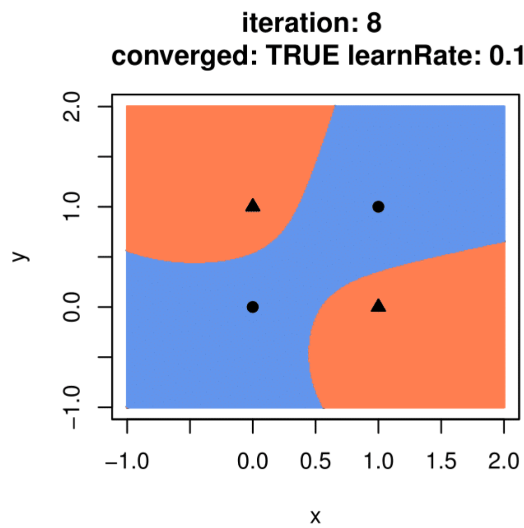

Figure 2.6: The decision boundary of the third order polynomial perceptron in iteration eight.

Chapter 3

Stream Learning and Concept Drift

In this chapter, we present the applications of neural networks on stream learning and anomaly detection. Our proposed models are based on different gradient descent algorithms because it has been shown that online learning is faster than batch learning and is suitable for large scale dataset mining [34]. We mainly use mini-batch gradient descent and stochastic gradient descent for fast weights updating.

3.0.1 "Threaded Ensembles of Supervised and Unsupervised Neural Networks for Stream Learning" (Winner of the Best Paper Award at the 29th Canadian Artificial Intelligence Conference)

The first paper was published at the 29th Canadian Artificial Intelligence Conference and won the best paper award. In this paper, we proposed two fast anomaly detectors based on ensembles of neural networks for evolving data streams.

Threaded Ensembles of Supervised and Unsupervised Neural Networks for Stream Learning

Yue Dong[1] and Nathalie Japkowicz[2]

[1] Department of Mathematics and Statistics, University of Ottawa, Ottawa Ontario, Canada,
[2] School of Electrical Engineering and Computer Science, University of Ottawa, Ottawa Ontario, Canada,

Abstract. Most existing model-based approaches to anomaly detection in streaming data are based on decision trees due to their fast construction speed [1]. This paper proposes two fast anomaly detectors based on ensembles of neural networks for evolving data streams. One model is a supervised online learning algorithm involving an ensemble of threaded multilayer perceptrons (MLP). The other model is a one-class learning algorithm with an ensemble of threaded autoencoders. The latter model only requires data from the positive class for training and is accurate even when anomalous training data are rare. The models feature an ensemble of multilayer perceptrons or autoencoders from multi-threads which evolve with data streams. Using multi-threads makes the methods highly efficient because both methods process data streams in parallel. Empirical studies show that both methods have linear runtime and require only constant memory space. When compared with Very Fast Decision Trees (VFDT), a state-of-the-art algorithm, our methods performed favorably with high detection accuracy and low training time for the datasets under consideration.

Keywords: stream mining, anomaly detection, neural networks, multilayer perceptrons, autoencoder, ensemble learning, supervised learning, one-class learning, ensemble

1 Introduction

Anomalies or outliers are rare events that are statistically significantly different from data which display normal behaviors. Anomaly detection is the problem of identifying outliers from normal data. Without being correctly detected, abnormal events could cause problems in many practical domains, such as safety surveillance, network security, and fraud detection [2]. Moreover, with the advances in technology, many datasets are no longer static and are continuously updated, sometimes with millions or even billions of new records per day. In most cases, this data arrives in streams and if it is not processed immediately or stored, it will be lost forever. In addition, the data arrives so rapidly that it

is usually not feasible to store them all. To extract useful knowledge from this high-speed real-time data, data stream mining became popular.

There are many special characteristics in detecting anomalies from data streams according to [2]. Firstly, data streams arrive continuously without stopping, so any off-line batch learning algorithm that requires storing the entire stream for analysis are not suitable for stream learning. Secondly, data streams for anomaly detection contain mostly normal data instances (positive instances) as the size of the anomaly class might be small compared to the normal class. In this case, one-class classifier is needed when the anomalies are not available for training. Thirdly, streaming data usually involve with concept drift where the distribution of data changes. Therefore, the model must learn adaptively and continuously.

In many streaming data applications, analysis can only be performed in one pass over the data due to the sheer size of the input. Because of this, traditional batch learning algorithms are often unsuitable for data stream mining. We thus need algorithms that can summarize information about prior data instances without keeping all of them in memory.

Several attempts have been made to modify the existing batch learners for data stream mining. According to Gama et al. [3], most of these attempts use decision trees "that continuously evolve over time, taking into account that the environment is non-stationary and computational resources are limited." Examples of these algorithms are Very Fast Decision Trees (VFDT) [4] and Concept-adapting Very Fast Decision Trees (CVFDT) [5].

However, it is not natural to use decision trees for data stream mining because they are very inefficient at dealing with concept drift — an important issue in data streams. Decision trees work top-down in a way involving choosing the best attributes to split data at each level. To grow one level, decision tree algorithms need to scan the whole dataset once. Then, once the attributes are chosen, it is costly to change the shape of the tree or to change the nodes which have already been chosen. In the worst case scenario, if the new training data point requires the root attribute to be changed, then all the nodes have to be re-chosen again. Since the characteristics of streaming data require the algorithm to constantly adjust to the new data, decision trees are not the best for data stream mining.

Unlike algorithms that alter decision trees' structure dynamically as streaming data arrives such as Hoeffding Trees [4], it is easy to see that neural networks (NN) are suitable for dealing with concept drift. In fact, they naturally handle well the task of incremental learning. A typical feed-forward neural network has one input layer, one output layer and several hidden layers. It learns by adjusting the weights between neurons in iterations. When continuous, high-volume, open-ended data streams come, it is necessary to pass the data in smaller groups (mini-batch learning) or one at a time (online learning) before updating the weights.

This paper proposes two anomaly detection algorithms based on ensembles of neural networks with multi-threads and mini-batches. The proposed methods have several advantages over other popular models. Firstly, only one pass of

the data is required for the algorithms to learn. Secondly, Both methods use fixed amounts of memory no matter how large are the datasets. Thirdly, SA only requires data from the normal class, which could work when a data stream contains very few, if any, anomalies. Fourthly, both models have the ability to deal with concept drift as they learn the data continuously with incremental updates. Lastly, the models separate streaming data into multiple threads to speed up the training process, then form an ensemble of multilayer perceptrons (MLP) or autoencoders from those threads to improve the detection accuracy.

Our experimental study shows that ensembles of streaming multi-threaded MLP or autoencoders lead to robust and accurate anomaly detectors. In order to have a good initial weight, we collect the first n samples as a window in each thread and run multiple epochs on it. Then it learns incrementally from the rest of the streaming data. This provides an anytime learner. In particular, it can still perform well early on, when there are not much data yet.

The rest of this paper is organized as follows: in Section 2, we discuss the related works. In Section 3, we propose our methods for mining streaming data. In Section 4, we discuss the datasets and the experimental design. In Section 5, we present the results and compare them with the state-of-the-art stream mining algorithm — VFDT. In Section 6, we conclude with remarks and discussions.

2 Related Works

There have been many works proposed in the past for anomaly detection, but most of them don't work on streaming data. The most popular batch anomaly detectors include the algorithms based on classification [6], clustering [8], and distance [9] as the traditional machine learning algorithms. A few one-class learners were also proposed to solve class imbalance problems in anomaly detection. Examples are One-Class Support Vector Machine (SVM) [10] and Isolation Forest [11]. These batch learning methods require loading the entire dataset for training [2]. Therefore they are not suitable for processing streaming data.

Data stream mining has become prevalent in the last two decades. Based on [12], several classification methods suitable for data streams have been developed. Examples are Very Fast Decision Trees (VFDT) [4] based on Hoeffding trees, Online Information Network (OLIN) [13], On-demand Classification [14], and the clustering algorithm CluStream [15]. There are many other popular algorithms, such as Ultra Fast Forest Trees (UFFT) [1] and Concept Based Decision Tree (CBDT) [16].

As we can see, most of these stream mining algorithms are based on decision trees. However, it is not natural to use decision trees when learning has to be continuous, which it frequently does due to concept drift. When concept drift is present, the decision tree structure is unstable in the setting of stream mining and it is costly to adjust the trees' shape with new data.

We study VFDT in this paper as an illustration for the disadvantages of dynamic tree based models because it is representative for dynamic tree based models. The model is a supervised learning model which requires the labels from

both normal and abnormal classes. This is sometimes infeasible when abnormal instances are rare. However, the biggest problem for VFDT is its slow training time and inability to adjust to concept drift.

VFDT builds each node of the decision tree based on a small amount of data instances from a data stream. According to Domingos [4], VFDT works as follows: "given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively."

To decide how many data instances are necessary for splitting a node, the Hoeffding bound is used. It works as follows: consider r as a real valued random variable, n as the number of independent observations of r, and R as the range of r, then the mean of r is at least $r_{avg} - \epsilon$, with probability $1 - \delta$:

$$P(\bar{r} \geq r_{avg} - \epsilon) = 1 - \delta$$

where

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

With the Hoeffding bound, VFDT can guarantee that the performance of Hoeffding trees converges to that of a batch learning decision tree [4]. However, this data structure is not "stable" because this convergence only works when there is no concept drift. Subtle changes of the data distribution can cause Hoeffding trees to fail as demonstrated in the HTTP+SMTP dataset from our experiment.

To mitigate the issue of concept drift, the same authors introduced Concept-adapting Very Fast Decision Trees (CVFDT). CVFDT adapts better to concept drift by monitoring changes of information gain for attributes and generating alternate subtrees when needed. However, it still can't solve the issue completely because the algorithm only monitors the leaves. Due to this disadvantage of decision tree based models, other researchers tried to stay away from having to regrow decision trees at all.

Many researchers have proposed algorithms based on incremental neural networks, such as the papers [17–20]. These algorithms pass the data points one by one and update the weights every time after scanning a data point. The shortcoming of these algorithms is that the results are highly dependent on the random initial weights. In batch learning, we pass the data in many epochs to avoid the effect of random initial weights. This is impossible in incremental neural networks since the data only gets passed once and the previous data is discarded. Moreover, updating the weights with backpropagation can be slow. If the data arrives too quickly, the data might be lost before it can be processed.

The models proposed in this paper overcome the disadvantages of incremental neural networks by the following three means: (i) using weights obtained from a mini-batch neural network to minimize the effect of random initial weights; (ii) dividing the data into multiple threads and processing them in parallel to speed up the training; (iii) forming ensembles of neural networks (MLP or autoencoders) to improve the detection accuracy.

3 The Proposed Methods

The proposed methods consist of ensembles of online multi-threaded MLP or autoencoders with mini-batch windows. Each MLP or autoencoder is built from one thread of the data streams. To facilitate a better set of initial weights, we collect n data points in a mini-batch window at the beginning of each thread. An MLP or autoencoder is then trained on the corresponding data with multiple epochs. After that, the final weights are passed as the initial weights for training a subsequent neural network on each evolving data thread. The system then operates with multiple threads of incremental MLP or autoencoders for anomaly detection.

3.1 Streaming Neural Networks

The first model we are introducing for data stream mining is called Streaming Multilayer Perceptrons (SMLP). Essentially, this algorithm involves an ensemble of multilayer perceptrons from multiple threads. Each multilayer perceptron uses the initial weights passed from a neural network trained on the first n data points in the corresponding data thread.

The model is trained with the following steps:

1. Initialize m threads for parallel processing of data streams.
2. The data point x_i goes to the thread $i \ mod \ m$.
3. For each thread, collect n data samples at the beginning of the stream. Then train an MLP with the n data points in the window for up to t epochs. Obtain final weights to pass as the initial weights for the subsequent incremental multilayer perceptron.
4. Train an incremental multilayer perceptron with the initial weights obtained from step 3 for each thread. Update the weights every time after one data instance passes. Thus, we train an MLP by feeding the data one by one in order with only one epoch.

After passing all the training data, SMLP is ready for classification. When a testing instance arrives, the m neural networks trained from m threads will vote for the output. If the labels of testing data arrive later, we can use them to update the model.

Time and Space Complexities: The key operation in SMLP is updating the weights each time a training data point passes. Training a neural network with backpropagation can be slow. However, with multi-threaded processing and online learning where we train neural networks with only one epoch in parallel, the speed of processing data improves dramatically. With this fast data processing, we can safely discard the data points which have been processed, so only the weights of the m neural networks are being stored in memory.

In terms of accuracy, SMLP optimizes incremental neural networks since the initial weights are adjusted by training from the first n data instances. This minimizes the problem brought by the random initial weights. Therefore, the

performance of SMLP is less random than that of incremental neural networks. Moreover, having ensembles of MLP from different segments of the data stream improves the performance of SMLP. As shown in the paper [23], ensembles of classifiers usually have better performances than a single classifier as long as individual classifiers are diverse enough. In SMLP, each MLP in the ensemble is trained on a unique subset of the data stream. This guarantees individual MLP in the ensemble are diverse enough to give a better performance than a single classifier.

3.2 Streaming Autoencoders

The second model we are introducing is called Streaming Autoencoders (SA), which is based on ensembles of autoencoders with similar settings as the first model. Since autoencoders are only trained on normal instances in the dataset, this is an unsupervised learning model. This model has the advantage of dealing with the class imbalance problem, namely when the abnormal instances are rare or not available for training [21].

An autoencoder is a feedforward neural network with an input layer, an output layer and one or more hidden layers. What special about autoencoders is that the input layer and the output layer have the same size. In other words, there are same number of nodes in the input layer and the output layer of an autoencoder. Thus, autoencoders have the goal of reconstructing their own inputs X.

We usually call the connections between the input layer to hidden layers of an autoencoder as "encoders". Similarly, we call the connections between hidden layers to the output layer as "decoders". Then an autoencoder can be defined as a composition of two maps ϕ and ψ, such that the goal is to minimize the reconstruction errors of inputs X:

$$\phi : \mathcal{X} \to \mathcal{F}$$

$$\psi : \mathcal{F} \to \mathcal{X}$$

$$\arg\min_{\phi,\psi} \|X - (\psi \circ \phi)X\|^2.$$

In our study, we consider the case where only one hidden layer is present. An autoencoder with one hidden layer takes the input $\mathbf{x} \in \mathbb{R}^d$ and maps it to $\mathbf{h} \in \mathbb{R}^p$ (p is the number of hidden units in the hidden layer) with a non-linearity activation function ψ_1:

$$\mathbf{h} = \psi_1(\mathbf{w}\mathbf{x} + \mathbf{b}).$$

Then $\mathbf{h}$ is mapped back to $\mathbf{x}'$ through the decoding process, where $\mathbf{x}'$ is called a reconstruction of $\mathbf{x}$:

$$\mathbf{x}' = \psi_2(\mathbf{w}'\mathbf{h} + \mathbf{b}')$$

The reconstruction error can be defined in terms of many ways, the most common ones are the squared error and the cross entropy. Autoencoders are trained to minimize the reconstruction error which is usually defined as the follows:

$$L(\mathbf{x}, \mathbf{w}) = \mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma_2(\mathbf{w}'(\sigma_1(\mathbf{w}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2.$$

The empirical risk for n samples is the loss $L(\mathbf{x}, \mathbf{w})$ averaged over n samples:

$$E_n(\mathbf{w}) = \frac{1}{n} \sum_{i-1}^{n} L(\mathbf{x_i}, \mathbf{w}).$$

To minimize the empirical risk, autoencoders need to be trained through weights adjusting. There are different error propagation mechanisms and gradient descent optimizers for this task. We use mini-batch learning combine with stochastic gradient descent in our algorithms.

While constructing an autoencoder, we usually want to compress the data in order to find a good representation. This is because the autoencoder can become useless by learning an identity function. This only happens when the hidden layers are larger than the input layer. In this case, we usually use denoising autoencoders or autoencoders with dropout to avoid learning the identity function [22]. In our model, we use autoencoders with dropout since it gives the best results throughout our experiments.

In order to be used for anomaly detection, autoencoders are trained on normal (positive) instances. Once trained, they have the ability to recognize abnormal instances in new data (those instances with a large reconstruction error). Therefore, the reconstruction error is used as the measurement to profile the degree of anomaly for autoencoders.

To decide the threshold that discriminates normal and abnormal instances, we need to pass the training data from both classes. The threshold is then set empirically as the reconstruction error which gives the best separation of the two classes as discussed in paper [21].

3.3 Dealing with Concept Drift

When dealing with data streams, the underlying data distribution can change over time. This phenomenon is called concept drift. Concept drift can cause predictions to become less accurate over time [24]. Our study is restricted to one type of concept drift where the conditional distribution of the output changes, but the distribution of the input may or may not change.

When the conditional distribution changes in the output, there are two solutions to prevent deterioration of prediction accuracy [25]. Active solutions rely on "triggering mechanisms which explicitly detect concept drift in the data generating process" [25]. On the contrary, passive solutions build models which are continuously updated. This can be achieved by "retraining the model on the most recently observed samples" [24] or "enforcing an ensemble of classifiers" [26].

The two models proposed in this paper are passive predictive models with adaptive learning. They use blind adaptation strategies [26] without any explicit detection of concept drift. As [26] explained, "the blind approaches forget old concepts at a constant rate whether changes are happening or not. As time goes on, the newly arrived data tends to erase away the prior patterns".

In neural networks, learning is always associated with forgetting. The newly arrived data updates the weights of neural networks, which makes the effects of old data in the stream becomes less and less important. Since streaming data arrives at a high speed, the weights in the models get updated rapidly and the model adjusts to the concept drift quickly.

Moreover, the use of threaded ensemble learning makes our models less sensitive to noises and therefore more stable in adapting to concept drift. The ability to continuously learn from data streams while keeping previously learned knowledge is known as the stability–plasticity dilemma [27]. This dilemma happens because there is a trade-off between distinguishing noises from new patterns. Since we split the data into multiple threads, the chance of noises affecting all threads is small compared to a single neural network. If a concept drift is truly happening, it will show up in multiple threads and be detected and adapted by the majority votes of the neural networks in the ensemble.

4 Experimental Setup

4.1 Dataset

The datasets used in our experiments are summarized in Columns 2 to 4 of Table 1. We used typical benchmark datasets for comparisons. SMTP, HTTP and SMTP+HTTP are streaming anomaly data published from the KDD Cup. Note that SMTPHTTP is constructed by using data from SMTP followed by data from HTTP. This dataset involves concept drift as more details can be found in [2].

The fourth dataset COVERTYPE is from the UCI repository. It has some distribution changes since there are multiple different cover-types in the normal class [2]. We used the smallest class Cottonwood/Willow with 2747 instances as the anomaly class. The last dataset we used is the SHUTTLE dataset from UCI repository which has no distribution change [2].

In VFDT, all datasets are processed as streams with a chunk size of 1000. In SMLP and SA, the window size for the mini-batch learning is 100 and the model is updated incrementally (one by one). The order of the data instances is preserved in each data stream. We split the data stream into 80% for training and 20% for testing.

We are aware that our datasets are small for today's standards of data streams which arrive very rapidly in a large volume. However, SMLP and SA are fully distributed streaming algorithms that have the potential to process big data streams. In fact, our models are implemented by the h2o deep learning package, which processes data point-by-point in each core processor.

4.2 Experimental Settings:

The parameter settings for our two models are 8 threads with window sizes of 100 for initial weights. For streaming autoencoders, we use input dropout of 0.1 and tanh functions with 50% dropout as the activation functions.

Our methods are implemented in R with h2o deep learning package. The total memory allocated for the H2O cluster is 1.78 GB. We use VFDT from RMOA package for the comparison. All experiments were conducted on a 2.7 GHz Intel Core i5 CPU with 8GB RAM.

The area under the curve (AUC) based on anomaly scores is used as the evaluation metric. In SMLP and VFDT, the anomaly scores are calculated as the probability that a point is predicted as normal. Thus, a true anomaly instance generally has a low anomaly score, while a normal point has a high probability and therefore has high anomaly scores. For SA, we use the inverse of the reconstruction error as the anomaly score, because a test data point with a smaller reconstruction error is more likely to be a normal instance.

Based on the anomaly scores and the true labels, we then computed the AUC scores for all anomaly detectors. AUC is the main indicator reported in this paper as in Table 1, which ranges from 0 (the worse) to 1 (the best).

In all experiments, we conducted 10 independent runs of each algorithm on each dataset and then computed the average results. A t-test at 5% level of significance was used to compare performance levels of the algorithms. Significant results of our models (results which are significantly better than that of VFDT) are highlighted in bold in Table 1.

5 Experimental Results

We report the experiments' results in this section. First, we compute the AUC of our models and compare them with the AUC of VFDT. VFDT and SMLP are both multi-class classifiers which have the access to data instances from both classes. In contrast, Streaming Autoencoders use only "normal" data for training, by which we mean data from the positive class. We expect VFDT and Streaming Multilayer Perceptrons (SMLP) to produce better results than streaming autoencoders. Since we use VFDT as the baseline for comparisons, our streaming data anomaly detectors will be deemed competitive if their performance can be shown to be comparable with the performance of VFDT.

Interestingly, Table 1 shows that SA actually gives higher AUC scores than VFDT and SMLP on three (i.e., SMTP, SMTPHTTP, and SHUTTLE) out of five datasets tested. Moreover, when there is a strong concept shift as in SMTP+HTTP, SA performs significantly better than the other two approaches.

Compare the last three columns of Table 1, we observe that VFDT's runtime to be four to six times slower than SMLP and eight to ten times slower than SA. This is because that VFDT requires modifications of the tree structures based on new coming data. More interestingly, we found that streaming autoencoders are always much faster than SMLP. This is because it is faster to update the autoencoders when the normal data instances are very similar to each other.

Dataset	Data Size	Dimensi-onality	Anomaly	AUC			Training time (seconds)		
				SMLP	SA	VFDT	SMLP	SA	VFDT
HTTP	623091	41	6.49%	0.988	0.993	0.996	70.03	17.88	348.75
SMTP	96554	41	12.25%	**0.996**	**0.997**	0.993	24.17	6.64	52.25
SMTP+HTTP	719645	41	7.26%	0.630	**0.997**	0.857	31.11	29.47	394.24
COVERTYPE	581012	54	0.47%	0.994	0.977	0.999	110.26	10.38	353.26
SHUTTLE	14500	10	5.97%	0.984	**0.994**	0.990	4.89	2.75	7.02

Table 1. Average AUC scores for Streaming Multilayer Perceptrons (SMLP), Streaming Autoencoders(SA), and Very Fast Decision Trees (VFDT). Using VFDT as a reference, scores significantly higher than VFDT are printed in boldface.

Unlike VFDT, SMLP and SA process data streams based on incremental updates of the weights. The backpropagation in multiple threads is very efficient because it requires neither evaluations for dimensions nor selections for splitting points as in VFDT [2].

In our experiments, we found that the performance of our models are not too sensitive to the number of hidden units we choose in the hidden layers. Suppose the dimensionality (the number of attributes) of the datasets is n. Varying the number of hidden units from $2/3 * n$ to $2 * n$ gives similar results. The effect of randomly picking parameter values reduces as the ensemble size goes up. We can conclude that the insensitivities of our models to the parameters are due to ensemble learning. The random parameter settings in each learner might result in a weak learner, but ensemble several weak learner can lead to a strong learning algorithm.

6 Conclusion and Further Work

In this paper, we propose two neural networks based anomaly detection algorithms, Streaming Multilayer Perceptrons and Streaming Autoencoders. They both process data streams efficiently with the following key features: (i) they process data in only one pass without the need to store the entire dataset into the memories; (ii) they have the ability to deal with concept drift by learning continuously and adaptively. Moreover, as a one class learner, SA could learn different types of anomalies even they are not available in the training data.

Our empirical studies show that both models feature high detection accuracy and low runtime in mining evolving data streams. Both algorithms' AUC scores are comparable to VFDT — a state-of-the-art algorithm. In terms of training time, both models outperform VFDT. Moreover, the streaming autoencoder is very fast compared to SMLP or VFDT. It can be trained using only data instances from the positive class, which is an advantage when the abnormal instances are rare or not even available for training.

References

1. Gama, Joao, Pedro Medas, and Pedro Rodrigues. "Learning decision trees from dynamic data streams." Proceedings of the 2005 ACM symposium on Applied computing. ACM, 2005.
2. Tan, Swee Chuan, Kai Ming Ting, and Tony Fei Liu. "Fast anomaly detection for streaming data." IJCAI Proceedings-International Joint Conference on Artificial Intelligence. Vol. 22. No. 1. 2011.
3. Gama, Joao, Pedro Pereira Rodrigues, and Raquel Sebastio. "Evaluating algorithms that learn from data streams." Proceedings of the 2009 ACM symposium on Applied Computing. ACM, 2009.
4. Domingos, Pedro, and Geoff Hulten. "Mining high-speed data streams." Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2000.
5. Hulten, Geoff, Laurie Spencer, and Pedro Domingos. "Mining time-changing data streams." Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2001.
6. Barnett, Vic, and T. Lewis. "Outliers in statistical data." 3 edition, Wiley. 1994.
7. Abe, Naoki, Bianca Zadrozny, and John Langford. "Outlier detection by active learning." Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2006.
8. He, Zengyou, Xiaofei Xu, and Shengchun Deng. "Discovering cluster-based local outliers." Pattern Recognition Letters 24.9 (2003): 1641-1650.
9. Bay, Stephen D., and Mark Schwabacher. "Mining distance-based outliers in near linear time with randomization and a simple pruning rule." Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003.
10. Heller, Katherine, et al. "One class support vector machines for detecting anomalous windows registry accesses." Workshop on Data Mining for Computer Security (DMSEC), Melbourne, FL, November 19, 2003. 2003.
11. Liu, Fei Tony, Kai Ming Ting, and Zhi-Hua Zhou. "Isolation forest." Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on. IEEE, 2008.
12. Hahsler, Michael, Matthew Bolanos, and John Forrest. "Introduction to stream: An Extensible Framework for Data Stream Clustering Research with R."
13. M., Last. "Online Classification of Nonstationary Data Streams. Intelligent Data Analysis, 6, 129147. ISSN 1088-467X. 2002.
14. CC., Aggarwal, Han J, Wang J, and Yu PS (2004). "On Demand Classification of Data Streams. Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, KDD 04, pp. 503508. ACM, New York, NY, USA.
15. CC., Aggarwal, Han J, Wang J, and Yu PS (2003). "A Framework for Clustering Evolving Data Streams. Proceedings of the International Conference on Very Large Data Bases (VLDB 03), pp. 8192.
16. Hoeglinger, Stefan, Russel Pears, and Yun Sing Koh. "Cbdt: A concept based approach to data stream mining." Advances in Knowledge Discovery and Data Mining. Springer Berlin Heidelberg, 2009. 1006-1012.
17. Polikar, Robi, et al. "Learn++: An incremental learning algorithm for supervised neural networks." Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 31.4 (2001): 497-508.

18. Carpenter, Gail A., et al. "Fuzzy ARTMAP: A neural network architecture for incremental supervised learning of analog multidimensional maps." Neural Networks, IEEE Transactions on 3.5 (1992): 698-713.
19. Furao, Shen, Tomotaka Ogura, and Osamu Hasegawa. "An enhanced self-organizing incremental neural network for online unsupervised learning." Neural Networks 20.8 (2007): 893-903.
20. Shen, Furao, and Osamu Hasegawa. "Self-organizing incremental neural network and its application." Artificial Neural NetworksICANN 2010. Springer Berlin Heidelberg, 2010. 535-540.
21. Japkowicz, Nathalie, Catherine Myers, and Mark Gluck. "A novelty detection approach to classification." IJCAI. 1995.
22. Bengio, Yoshua. "Learning deep architectures for AI." Foundations and trends in Machine Learning 2.1 (2009): 1-127.
23. Wang, Haixun, et al. "Mining concept-drifting data streams using ensemble classifiers." Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining. ACM, 2003.
24. Widmer, Gerhard, and Miroslav Kubat. "Learning in the presence of concept drift and hidden contexts." Machine learning 23.1 (1996): 69-101.
25. Gama, Joao, et al. "A survey on concept drift adaptation." ACM Computing Surveys (CSUR) 46.4 (2014): 44.
26. Elwell, Ryan, and Robi Polikar. "Incremental learning of concept drift in nonstationary environments." Neural Networks, IEEE Transactions on 22.10 (2011): 1517-1531.
27. Carpenter, Gail A., Stephen Grossberg, and John H. Reynolds. "ARTMAP: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network." Neural networks 4.5 (1991): 565-588.

3.0.2 "Threaded Ensembles of Autoencoders for Stream Learning" (Currenly under Peer Review for *Computational Intelligence*)

The second paper is an invited paper submitted to the journal *computational intelligence* which expands the first paper. In this paper, we improved the model *Streaming Autoencoders* (SA) by adding an additional checking scheme to distinguish anomalies from concept drift. This ensures that point anomalies are detected in real-time with a low false positive rate in detection. The method is highly efficient because it processes data streams in parallel with multi-threads and alternating buffers. Empirical comparisons to the state-of-the-art methods demonstrate that the proposed method features high detection rate and low false alarm rate.

Compare to the previous experiments in the first paper, we are able to improve the third model by the following means:

- We were able to experiment on datasets of twice bigger size.
- We installed two buffers in each thread to improve the speed of our new algorithm to 20-40% faster than the previous models.
- We used a master thread to distribute data for faster data processing.
- We employed a two step verification scheme for Streaming Autoencoders to reduce the false positive rate in anomaly detection.
- We compared our model with another state-of-the-art algorithm, Half Space Trees, which is a one-class streaming anomaly detector.

Threaded Ensembles of Autoencoders for Stream Learning

YUE DONG
Department of Mathematics and Statistics, University of Ottawa, Ottawa Ontario, Canada

NATHALIE JAPKOWICZ
Department of Computer Science American University Washington, D.C., USA

Anomaly detection in streaming data is an important problem in numerous application domains. Most existing model-based approaches to stream learning are based on decision trees due to their fast construction speed. This paper introduces *Streaming Autoencoders* (SA), a fast and novel anomaly detection algorithm based on ensembles of neural networks for evolving data streams. It is a one-class learner which only requires data from the positive class for training and is accurate even when anomalous training data are rare. It features an ensemble of threaded autoencoders with continuous learning capacity. Furthermore, SA uses a two-step detection mechanism to ensure that real anomalies are detected with a low false positive rate in detection. The method is highly efficient because it processes data streams in parallel with multi-threads and alternating buffers. Our analysis shows that SA has linear runtime and requires constant memory space. Empirical comparisons to the state-of-the-art methods on multiple benchmark datasets demonstrate that the proposed method detects anomalies efficiently with low false alarm rate.

Key words: stream mining, anomaly detection, neural networks, multilayer perceptrons, autoencoders, ensemble learning, supervised learning, one-class learning

1. INTRODUCTION

Anomaly detection on streaming data is a research area of increasing importance (Tan et al., 2011). Anomalies or outliers are rare events that are statistically different from normal instances. Anomaly detection on streaming data is the problem of identifying outliers from normal data in non-static data streams.

There are many special characteristics in detecting anomalies from data streams according to (Tan et al., 2011). Firstly, data streams arrive continuously without stopping, so any off-line batch learning algorithm that requires storing the entire stream for analysis is not suitable for stream learning. Secondly, data streams for anomaly detection contain mostly normal data instances (positive instances) as the size of the anomaly class might be small compared to the normal class. In this case, one-class classifiers are needed when the anomalies are not available for training. Thirdly, streaming data usually involve with concept drift where the distribution of data changes. Therefore, the model must learn adaptively and continuously.

When anomaly instances exist in the training data, supervised learning algorithms such as VFDT and its variations (Domingos and Hulten, 2000; Hulten et al., 2001; Rutkowski et al., 2013, 2014) can be used for streaming anomaly detection. These algorithms are based on the dynamic decision trees which are very inefficient and slow at dealing with concept drift. Some other authors (Tan et al., 2011; Wu et al., 2014) proposed fast one-class learners which randomly built decision trees in advance without data. These algorithms' performance highly depend on the random trees in the ensemble and the last subsample in data streams. Moreover, these algorithms have high false positive rates in anomaly detection because they make judgments based on a small subsample from the last window in the data stream. There have been other streaming anomaly detectors based on incremental neural networks. These

© The Authors. Journal Compilation © Wiley Periodicals, Inc.

algorithms have the convergence problems and don't predict well before substantial amounts of data are passed.

In response to the above-mentioned problems, this paper proposes a novel, fast and scalable one-class algorithm, Streaming autoencoders (SA), for detecting anomalies in data streams. The proposed method deals with concept drift through adaptive learning, has low false positive rate with a two step checking scheme, and performs well early on with little data required.

SA is based on ensembles of autoencoders with multi-threads and mini-batches. SA learns incrementally with mini-batches from the streaming data. Afterwards, the trained model is used to classify outliers with continuously learning capacity, thus the positive (normal) testing data predicted by the model can be used to update the model. This provides an anytime learner with the ability to adapt to the evolving data. In particular, it can still perform well early on when there is not much data. It is also important to note that when large volumes of data arrives, the technique will leverage a parallel computing model with multi-threads to distribute the load. By doing so, we can have a scalable model for detecting anomalies in real-time.

In addition, SA uses a two-step detection mechanism to check that points detected in the first phase of the system are real anomalies. This is achieved by an additional step using adjacent windows comparisons to eliminate the possibility of concept drift. When an anomaly is detected, the system automatically compares the data points from two different time windows before and after the anomaly occurs. As will be shown later, using the profiles between two windows to distinguish anomalies from concept drift in data streams helps reduce the false positive rate in anomaly detection.

Streaming Autoencoder (SA) has several advantages over other popular models. Firstly, only one pass of the data is required for the algorithm to learning. Secondly, it uses fixed amounts of memory no matter how large are the datasets. Thirdly, SA only requires data from the normal class, which could work when a data stream contains very few, if any, anomalies. Fourthly, SA has the ability to deal with concept drift as it learns the data continuously with incremental updates. Lastly, the model separates streaming data into multiple threads to speed up the training process. An ensemble of autoencoders from those threads also improves the detection accuracy. Empirical comparisons to the state-of-the-art anomaly detectors in data streams demonstrate that SA detects anomalies efficiently with low false alarm rate.

The following are the main contributions of this paper:

- We introduce a fast and accurate neural networks based anomaly detector for data streams, Streaming Autoencoders, for data stream mining.
- We employ an additional checking step on the proposed algorithm to reduce the false positive rate in anomaly detection.
- Empirical experiments on multiple benchmark datasets demonstrate our method performs favorably in terms of AUC and runtime when compared to other state-of-the-art algorithms.

The rest of this paper is organized as follows. In Section 2, we discuss the related works of stream learning algorithms and their shortcomings. In Section 3, we propose our method for mining streaming data and discuss the advantages of our model over existing models. In Section 4, we describe the datasets and the experimental design. In Section 5, we present the results and compare them with the state-of-the-art stream mining algorithms. In Section 6, we conclude with remarks and discussions.

2. RELATED WORKS

In this section, we survey previous works in anomaly detection and data stream mining. Since Domingos proposed VFDT in 2000 (Domingos and Hulten, 2000), decision tree based algorithms have been mainstream in data stream mining. We therefore mainly focus on works based on decision trees and point out their deficiencies. In view of these deficiencies, we then look at works using neural networks (NN). Finally, we turn to various methods that have been proposed to improve NN-based algorithms in data stream mining including the use of mini-batches and two-step verifications.

2.1. Anomaly Detection

There have been many works proposed in the past for anomaly detection, but most of them don't work on streaming data. The most popular batch anomaly detectors include the algorithms based on classification (Abe et al., 2006), clustering (Heller et al., 2003), and distance (Bay and Schwabacher, 2003) as the traditional machine learning algorithms. A few one-class learners were also proposed to solve class imbalance problems in anomaly detection. Examples are One-Class Support Vector Machine (SVM) (Schölkopf et al., 1999) and Isolation Forest (Liu et al., 2008). These batch learning methods require loading the entire dataset for training (Tan et al., 2011). Therefore they are not suitable for processing streaming data.

Recent works on anomaly detection are more focused on stream learning. Such examples include half space trees algorithm (Tan et al., 2011) , RS-Forest (Wu et al., 2014), ensemble of random cut trees algorithm (Guha and Schrijvers, 2016) and subspace embedding-based methods (Huang and Kasiviswanathan, 2015; Pham et al., 2014). Most of these algorithms are inspired by half space trees algorithm (HS-Trees) where random decision trees were built in advance without data. Once an ensemble of trees with the same lengths are built, the data points from the last window in the data stream are passed to the leaves of each tree, then anomalies are those instances which lie in the leaves without other data instances [1]. These algorithms' performances highly depend on the random trees in the ensemble and the last subsample in data streams. Moreover, these algorithms have high false positive rates in anomaly detection because only small subsamples of training data are used. Using a small sample for monitoring will cause more leaves to be empty or near empty and the data points that fall into these leaves will be identified as anomalies.

2.2. Dynamic Tree-based Data Stream Mining Algorithms

Supervised learning algorithms are also employed for streaming anomaly detection if the labels from both classes are available during training. For example, Very Fast Decision Trees (VFDT) (Domingos and Hulten, 2000) based on Hoeffding trees or its variations (Hulten et al., 2001; Rutkowski et al., 2013, 2014) are still widely used these days.

These algorithms are based on dynamic trees whose leaves are built from different chunks of stream data. However, it is not natural to use dynamic decision trees when learning has to be continuous, which it frequently does due to concept drift. When concept drift is present, the decision tree structure is unstable in the setting of stream mining and it is costly to adjust the trees' shape with new data.

We study VFDT in this paper as an illustration for the disadvantages of dynamic tree based models because it is representative for dynamic tree based models. The model is a supervised learning model which requires the labels from both normal and abnormal

[1] Anomalies can be also identified if the number of data instances in these leaves fall below a threshold

classes. This is sometimes infeasible when abnormal instances are rare. However, the biggest problem for VFDT is its slow training time and inability to adjust to concept drift.

VFDT builds each node of the decision tree based on a small amount of data instances from a data stream. According to Domingos (Domingos and Hulten, 2000), VFDT works as follows: “given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively.”

To decide how many data instances are necessary for splitting a node, the Hoeffding bound is used. It works as follows: consider r as a real valued random variable, n as the number of independent observations of r, and R as the range of r, then the mean of r is at least $r_{avg} - \epsilon$, with probability $1 - \delta$:

$$P(\bar{r} \geqslant r_{avg} - \epsilon) = 1 - \delta$$

where

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

With the Hoeffding bound, VFDT can guarantee that the performance of Hoeffding trees converges to that of a batch learning decision tree (Domingos and Hulten, 2000). However, this data structure is not “stable” because this convergence only works when there is no concept drift. Subtle changes of the data distribution can cause Hoeffding trees to fail as demonstrated in the HTTP+SMTP dataset from our experiment.

To mitigate the issue of concept drift, the same authors introduced Concept-adapting Very Fast Decision Trees (CVFDT). CVFDT adapts better to concept drift by monitoring changes of information gain for attributes and generating alternate subtrees when needed. However, it still can’t solve the issue completely because the algorithm only monitors the leaves. Due to this disadvantage of decision tree based models, other researchers tried to stay away from having to regrow decision trees at all.

2.3. Neural Networks-based Data Stream Mining Algorithms

Unlike algorithms that alter decision trees’ structure dynamically as streaming data arrives such as VFDT, neural networks (NN) naturally handle well the task of incremental learning with the ability of dealing with concept drift. When continuous, high-volume, open-ended data streams come, neural networks learn by passing the data in smaller groups (mini-batch learning) or one at a time (online learning) before updating the weights. In this manner, they can learn by seeing each example only once and therefore do not require examples to be stored.

Many researchers have proposed algorithms based on incremental neural networks, such as the papers (Polikar et al., 2001; Carpenter et al., 1992; Furao et al., 2007; Shen and Hasegawa, 2010). These algorithms pass the data points one by one and update the weights every time after scanning a data point. The shortcoming of these algorithms is that the results are highly dependent on the random initial weights. Moreover, these incremental learning algorithms with stochastic gradient descent (SGD) might never converge if inappropriate learning rates are used (Bottou, 2012).

To avoid the convergence issue of SGD and to have a better minimization of the empirical risk E_n, mini-batch gradient descent algorithm (MB-GD) in neural networks with random sampling (Wilson and Martinez, 2003; Ruder, 2016) became a popular practice. However, due to the random sampling, this approach is only used for static datasets where all the data are accessible. In addition, although MB-GD is faster than gradient descent (GD), it is still significantly slower than stochastic gradient descent (SGD).

Streaming Autoencoders uses both SGD and MB-GD for updating the weights. Therefore, it combines the advantages of both gradient descent algorithms. Each thread of SA is equipped with two buffers. The first batch of data from one of the buffers are trained with mini-batch learning in multiple epochs. By doing so, the model will give a reliable prediction at the beginning of the streams since its initial weights will not be as random. The autoencoders are then trained with stochastic gradient descent for faster processing of the streaming data. By dividing the data into multiple threads for parallel processing, implementing buffers for receiving and sending data, and learning with MB-GD followed by SGD, the speed of SA is comparable to tree-based algorithms.

Similar to random trees based algorithms in 2.1, an issue with streaming autoencoders is the high false positive rate in detection (Maselli et al., 2003). A few papers tackled this problem for NN-based methods by using a two step verification scheme (Hayes and Capretz, 2015; Brzeziński, 2010). These methods maintain some profiles across multiple sensors to distinguish point anomalies from contextual anomalies. These techniques are quite different from the ideas presented in this paper where a second step verification is added to distinguish point anomalies from concept drift.

In summary, the model proposed in this paper overcomes the disadvantages of existing NN-based models by the following means: (i) using mini-batch learning on chunks of stream data collected in the buffers to minimize the effect of random initial weights at the beginning of stream learning; (ii) dividing the data into multiple threads and processing them in parallel to speed up the training; (iii) updating weights with SGD for fast processing when more data arrive in the buffers; (iv) forming ensembles of neural networks (autoencoders) to improve the prediction accuracy; (v) adopting a two steps verification scheme to reduce the false positive rate in anomaly detection.

3. THE PROPOSED METHODS

This section provides detailed introduction of the proposed algorithm - Streaming Autoencoders (SA). We first introduce the basics of autoencoders. Then we present the implementation details of SA. Next, we discuss the techniques SA employed to deal with concept drift during model training and testing. The major notations used in this section are summarized in Table I.

3.1. Autoencoders

An autoencoder is a feedforward neural network with an input layer, an output layer and one or more hidden layers. What distinguishes autoencoders from other types of neural networks is that the input layer and the output layer of an autoencoder have the same size. Autoencoders have the goal of reconstructing their own inputs X.

We usually call the connections between the input layer to hidden layers of autoencoders as “encoders”. Similarly, we call the connections between hidden layers to the output layer as “decoders”. Then an autoencoder can be defined as a composition of two maps ϕ and ψ, such that the goal is to minimize the reconstruction errors of inputs X:

$$\phi : \mathcal{X} \rightarrow \mathcal{F}$$

$$\psi : \mathcal{F} \rightarrow \mathcal{X}$$

$$\arg\min_{\phi,\psi} \|X - (\psi \circ \phi)X\|^2$$

In our study, we consider autoencoders with only one hidden layer. An autoencoder with

Table 1: Table of Major Notations

Notation	Description
$\mathcal{X} = R^d$	Feature space with d dimensions
X	A random variable take values from the feature space $\mathcal{X}$
$\mathbf{x} \in \mathcal{X}$	An instance in feature space $\mathcal{X}$
$\mathbf{x}'$	A reconstruction instance of $\mathbf{x}$
ϕ	A shaping function for autoencoders
ψ	An activation function for autoencoders
$\mathbf{h}$	The latent representation in autoencoders of an instance
$\mathbf{w}$	Weight matrix in an autoencoder
$\mathbf{b}$	A vector of bias in autoencoders
$L(\mathbf{x}, \mathbf{w})$	Loss function of an instance $\mathbf{x}$
$E_n(\mathbf{w})$	The empirical risk for n samples
η	Learning rate for updating autoencoders
$\mathbf{z} = (\mathbf{x}, y)$ A pair of data instance and its corresponding label	

one hidden layer takes the input $\mathbf{x} \in \mathbb{R}^d$ and maps it to $\mathbf{h} \in \mathbb{R}^p$ (p is the number of hidden units in the hidden layer) with a non-linearity activation function ψ_1:

$$\mathbf{h} = \psi_1(\mathbf{w}\mathbf{x} + \mathbf{b}).$$

Then $\mathbf{h}$ is mapped back to $\mathbf{x}'$ through the decoding process, where $\mathbf{x}'$ is called a reconstruction of $\mathbf{x}$:

$$\mathbf{x}' = \psi_2(\mathbf{w}'\mathbf{h} + \mathbf{b}').$$

The reconstruction error can be defined in terms of many ways, the most common ones are the squared error and the cross entropy. Autoencoders are trained to minimize reconstruction errors usually defined as the follows:

$$L(\mathbf{x}, \mathbf{w}) = \mathcal{L}(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|^2 = \|\mathbf{x} - \sigma_2(\mathbf{w}'(\sigma_1(\mathbf{w}\mathbf{x} + \mathbf{b})) + \mathbf{b}')\|^2.$$

The empirical risk for n samples is the loss $L(\mathbf{x}, \mathbf{w})$ averaged over n samples :

$$E_n(\mathbf{w}) = \frac{1}{n}\sum_{i-1}^{n} L(\mathbf{x_i}, \mathbf{w}).$$

To minimize the empirical risk, autoencoders need to be trained through weights adjusting. There are different error propagation mechanisms and gradient descent optimizers for this task. As mentioned earlier in section 2.3, two kinds of gradient descent optimization procedures are used with backpropagation in this paper:

(1) Mini-batch gradient descent: parameter updates are carried out after accumulating a fix size n of training data. The gradient is then computed as the average of losses over this n data points:

$$\mathbf{w}(k) = \mathbf{w}(k-1) - \eta\frac{\partial E_n}{\partial \mathbf{w}}(\mathbf{w}(k-1))$$

$$= \mathbf{w}(k-1) - \eta \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L}{\partial \mathbf{w}}(\mathbf{x_i}, \mathbf{w}(k-1)).$$

(2) Stochastic gradient descent (online gradient): parameter updates are performed every time a single example is passed through the network $\mathbf{x_t}$:

$$\mathbf{w}(t) = \mathbf{w}(t-1) - \eta \frac{\partial L}{\partial \mathbf{w}}(\mathbf{x_t}, \mathbf{w}(t-1))$$

where $\mathbf{w}$ is the weights (including the bias), η is the learning rate, L is the loss function of a particular data point and E_n is the empirical risk of n data examples.

While constructing an autoencoder, we usually want to compress the data in order to find a good representation. This is because the autoencoder can become useless by learning an identity function. This only happens when the hidden layers are larger than the input layer. In this case, we usually use denoising autoencoders or autoencoders with dropout to avoid learning the identity function (Bengio, 2009). In our model, we use autoencoders with dropout since it gives the best results throughout our experiments.

3.2. Streaming Autoencoders

The proposed method, Streaming Autoencoders, is based on ensembles of incremental multi-threaded neural networks with buffers. Each neural network is built from a particular subspace (one thread) of the data stream. In each thread, we installed two buffers for collecting and processing data. A master thread is used to distribute the streaming data. The two buffers in each thread are used alternatively for receiving data from the stream and sending the data to neural networks for training. This guarantees that each autoencoder is continuously learning without being idle and the data which haven't been processed in time are stored.

This model is specifically designed for anomaly detection where the percentage of negative instances is very small. Since the autoencoders are only trained on normal instances in the dataset, this is an unsupervised learning model. This model has the advantage of dealing with the class imbalance problem, namely when the abnormal instances are rare or not available for training (Japkowicz et al., 1995). Moreover, this model has the ability to reduce the false positive rate in anomaly detection by using an extra anomaly checker as explained in section 3.4.

In this section, we present the implementation details and algorithmic flows of Streaming Autoencoders. We first talk about the training process of SA, then turn into the testing process with the two step verification scheme. Next, we discuss our algorithm in terms of time and space complexity.

3.3. Training a Streaming Autoencoder

SA is trained with a combination of mini-batch gradient descent algorithm (MB-GD) and stochastic gradient descent algorithm (SGD). To facilitate a better set of initial weights, the first set of data points collected in each buffer are trained in multiple epochs with MB-GD. After that, the models are trained incrementally with SGD on each evolving data thread. The system then operates with multiple threads of incremental autoencoders for anomaly detection.

The general pipeline for training the SA is shown in Figure 1 and the pseudo-code is shown in Algorithm 1. SA is roughly trained with the following steps:

(1) Initialize M threads and $2M$ buffers for parallel processing of data streams.
(2) For each data point x_i in the stream,

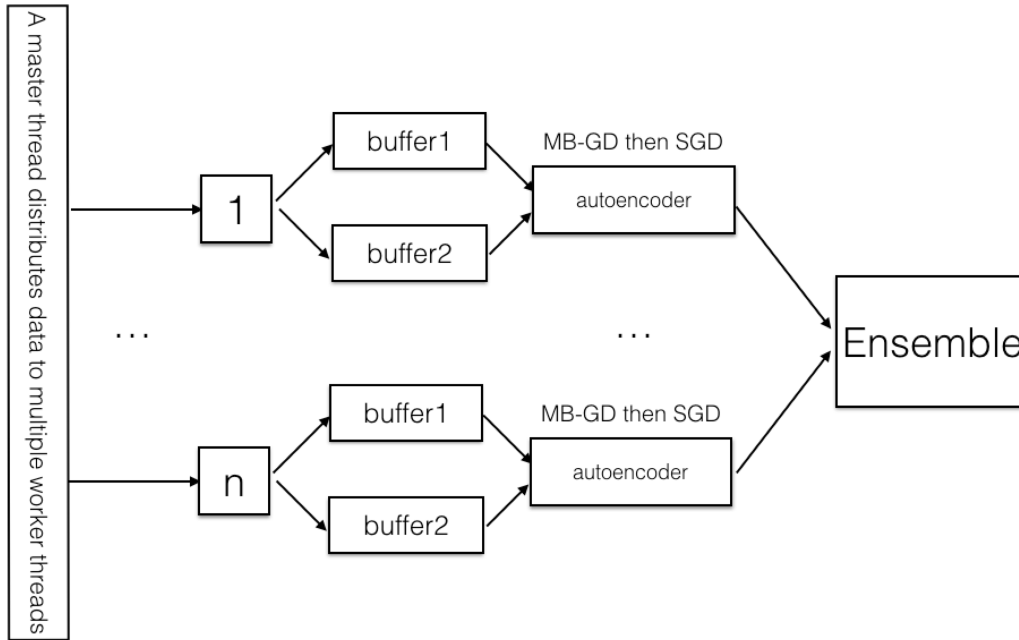

FIGURE 1: An illustration of the training pipeline for the proposed model - Streaming Autoencoders (SA).

- x_i goes to the first empty buffer it encounters
- when the buffer is full, start training the corresponding autoencoder with the data from the buffer

During training, using MB-GD followed by SGD overcomes the disadvantages of only using SGD. As mentioned in (Le Cun, 2004), "simple batch algorithms converge linearly to the optimum $\mathbf{w_n}^*$ of the empirical cost E_n. Whereas online algorithms may converge to the general area of the optimum, the optimization proceeds rather slowly during the final convergence phase". In some cases, it might not converge to the optimum weights at all if the learning rate is not chosen properly.

However, according to (Le Cun, 2004), "new examples in the data stream will follow the underlying asymptotic distribution $p(\mathbf{z})$ if the amount of such data points are nearly unlimited". In this case, the optimal weights $\mathbf{w}(t)$ are approximated directly and therefore the empirical risk is converged to the expected risk E_∞ over the underlying data distribution. Thus, using mini-batch training followed by online learning (as proposed in SA) has the advantage of finding the area of local optimal for the first k samples, then using possibly unlimited data in the stream for $\mathbf{w}$ converge to the optimal $\mathbf{w}*$ where the E_∞ is minimized.

After passing all training data, SA is ready for classification. When a testing instance arrives, the M neural networks trained from M threads will vote for the output. If the labels of testing data arrive later, we can use them to update the model.

3.4. Thresholds for Predictions and the Two Step Verifications

In order to be used for anomaly detection, the autoencoders are trained on normal (positive) instances as shown in Algorithm 1. Once trained, they have the ability to recognize abnormal instances in new data (those instances with a large reconstruction error). Therefore,

Algorithm 1 StreamingAutoencoders

Input: S is a sequence of positive (normal) Data Stream examples,
t is the number of epochs trained at the beginning of each thread,
B is the buffer size,
M is the number of threads
Output: SA is an ensemble of streaming autoencoders
Buffers is a set of buffers with latest data points in the stream stored in the buffers

1: **procedure** STREAMINGAUTOENCODERS(S,B,M)
2: *Initialize M threads*
3: *In each thread i, initialize 2 buffers B_{2i+1} and B_{2i+2} of size B*
4: *In each thread i, initialize an autoencoder i with firstBatch[i]=False*
5: $k = 0$
6: **for** each data sample $x_j \in S$ **do** InsertAndTrainBuffer(x_j, k, t, firstBatch)
7: **return** SA= $(A_1, ..., A_M)$; Buffers= $(B_1, ..., B_{2M})$
8:
9:
10: **procedure** INSERTANDTRAINBUFFER(x_j, k, t, firstBatch)
11: **while** BufferIsFull(k) == False **do**
12: Insert x_j into buffer k
13: **if** autoencoder $floor((k-1)/2)$ is idle **then**
14: **if** firstBatch=False **then**
15: ▷ *mini-bach train $autoencoder_{floor((k-1)/2)}$ with data from buffer k in t epochs*
16: miniBatchTrain($floor((k-1)/2)$, buffer k, t)
17: firstBatch=True
18: Clear buffer k
19: **else**
20: ▷ *online train $autoencoder_{floor((k-1)/2)}$ with data from buffer k one by one*
21: onlineTrain($floor((k-1)/2)$, buffer k)
22: Clear buffer k
23: k=(k+1)%n
24: **return** SA= $(A_1, ..., A_M)$; Buffers= $(B_1, ..., B_{2M})$

the reconstruction error is used as the measurement to profile the degree of anomaly for autoencoders.

When the labels of training data are available, we could pass them to the model for deciding a better threshold that discriminates normal and abnormal instances. We installed a window of flexible size h to collect all anomalies appearing in the training data streams. These anomalies are then passed to each autoencoder for deciding the threshold of reconstruction error. The threshold of reconstruction error is then set empirically as discussed in paper (Japkowicz et al., 1995). The basic idea is to find the threshold which gives the best separation of the two classes in training data as figure 2 shows.

Note that using Algorithm 2 with the empirically decided threshold for testing will predict more anomalous instances than a batch learner. This is because only the positive instances stored in the buffer are used to decide the threshold in SA. However, we implemented a checking step (Algorithm 3) where we distinguish anomalies from concept drift and therefore reduce the false positive rate in detection. Normal test instances (with or without labels) are then queued into the buffers for continuously learning. The pipeline of SA for testing is shown in figure 3.

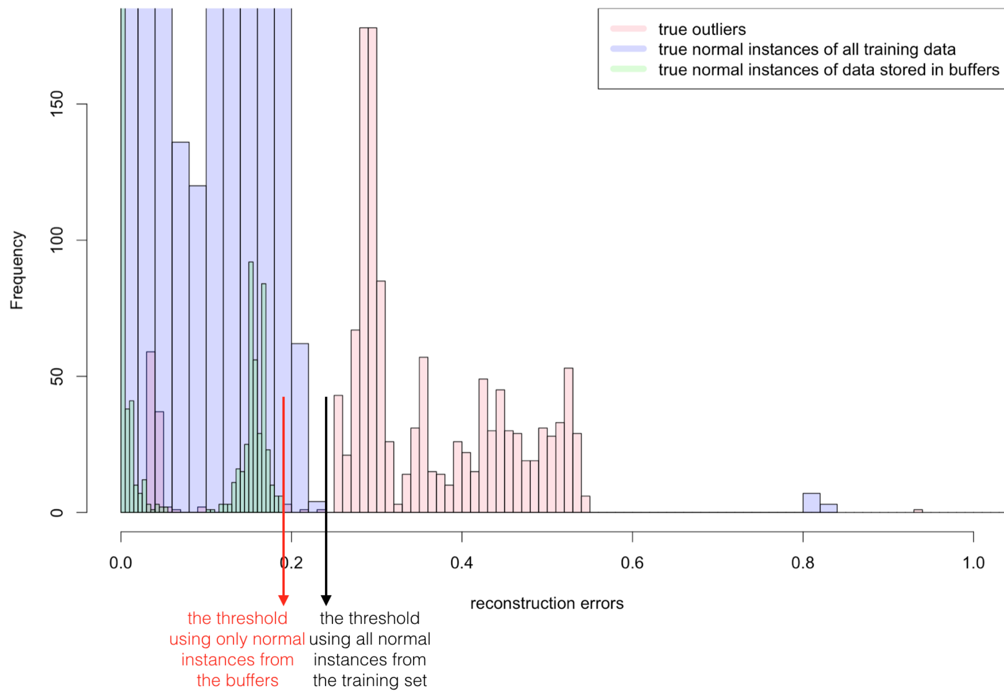

FIGURE 2: An illustration of how to empirically decide the threshold of the reconstruction error for Streaming Autoencoders

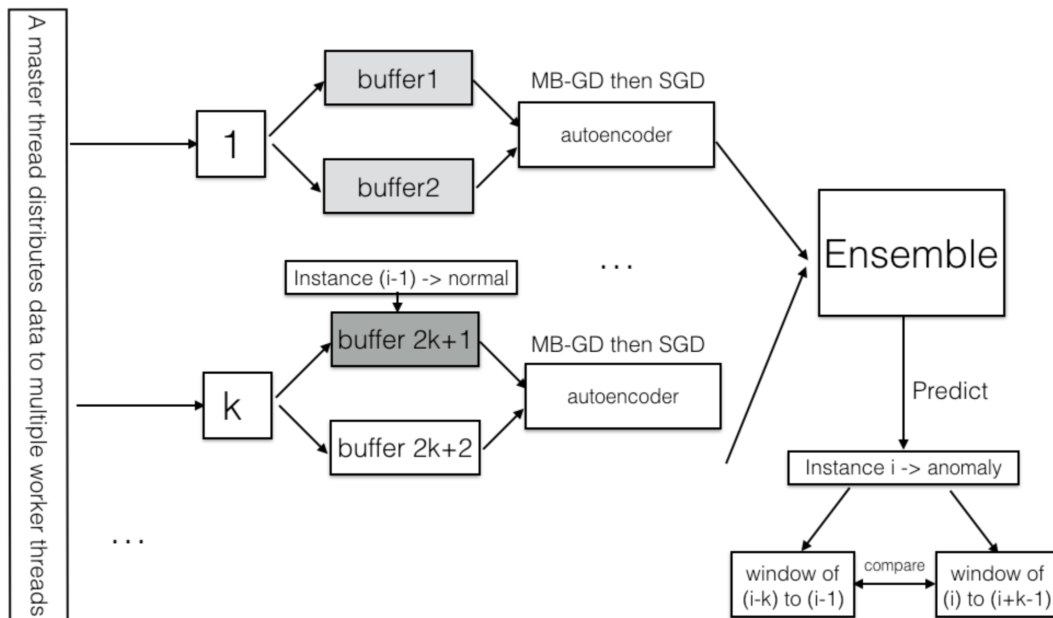

FIGURE 3: An illustration of the testing and continuously learning process for Sreaming Autoencoders.

Algorithm 2 AnomalyDetect

Input: T is a sequence of testing examples,
t is the number of epochs trained at the beginning of each thread,
SA is a set of autoencoders,
w is the window size,
Buffers is a set of buffers with latest data points in the stream stored in the buffers
Thresholds is a vector of thresholds for autoencoders in the ensemble
Output: Results is a list of predicted labels,
SA; Buffers a set of autoencoders and buffers updated by predicted normal testing data

```
1:  procedure ANOMALYDETECT(T, SA, w, t, Buffers)
2:      for each testing data sample x_i ∈ T do
3:          if pred(x_i) == False then
4:              Results[i]="normal"
5:              InsertAndTrainBuffer(x_i, k, t, firstBatch=False)
6:          else if IndeedAnomaly(x_i, w, Buffers) == False then
7:              Results[i]="normal"
8:              InsertAndTrainBuffer(x_i, k, t, firstBatch=False)
9:          else
10:             Results[i]="abnormal"
        return SA=(A_1,...,A_M); Buffers; Results
11:
12:
13: procedure PRED(x,Thresholds)
14:     pass x through each trained autoencoder in the ensemble
15:     for each autoencoder A_i do
16:         if L_i(x, w) > Thresholds[i] then
17:             pred[i] == "abnormal"
18:         else
19:             pred[i]== "normal"
        return mode(pred)
```

Algorithm 3 AnomalyCheck

Input: x_i is a testing instance which is detected as anomaly in pred(x_i)
Output: Anomaly = True or False

```
1: procedure INDEEDANOMALOUS(x_i, w, Buffers)
2:     extract w data points (w_1 = i − w, ..., i − 1) from buffers before i
3:     train an autoencoder A_before on these w normal instances
4:     accumulate w instance after data i
5:     train an autoencoder A_after on these w unknown label instances
6:     if pred(x_i, A_before) == "abnormal" and pred(i, A_after) =="abnormal" then
7:         return "abnormal"
8:     else
9:         return "normal"
```

3.5. Time and Space Complexities

The key operation in SA is updating the weights each time from a subset of data collected in a buffer. The first subset in the data stream are trained with MB-GD and the rest are trained with SGD. The updating that uses backpropagation with Stochastic gradient descent is very fast compared to MB-GD. Training a neural network with backpropagation can

be slow. However, with multi-threaded processing and online learning, the speed improves dramatically.

Because only the first subset of data in the stream are trained with MB-GD in multiple epochs, SA can be trained very fast. Training k data samples from a buffer with MB-GD or SGD uses constant time [2].

An ensemble of autoencoders has training time complexity $O(t+\frac{n-BM}{BM})$ which is linear for an ensemble with fixed buffers size B, t epochs for the first training window and ensemble size M (same as thread size). Adding a checking step in SA makes the testing time longer. However, for each anomaly detected, the checking time is constant. Therefore, the testing time for SA is also linear with respect to the number of testing instances.

With this fast processing, we can safely discard the normal data which have been processed. In order to choose the thresholds for the reconstruction errors, we store all the negative instances (size h) in SA. Other than that, only the data points in the buffers and the weights of the M neural networks are being stored in memory. The space needed to store $2M$ data points is constant and the space needed to store M neural networks is also constant. The space complexity for SA is $O(1+h)$ which is almost a constant since h is usually small.

In terms of accuracy, SA optimizes incremental neural networks since the initial weights are adjusted by mini-batch training from the first B data instances. This minimizes the problem brought by the random initial weights. Therefore, the performance of SA is less random than that of incremental neural networks. Moreover, having ensembles of autoencoders from different segments of the data stream improves the performance (Wang et al., 2003). Ensembles of classifiers usually have better performances than a single classifier as long as individual classifiers are diverse enough. In SA, each autoencoder in the ensemble is trained on a unique subset of the data stream. Therefore, the individual autoencoders in the ensemble are diverse enough to give a better performance than a single classifier.

3.6. Dealing with Concept Drift

One requirement for stream learning algorithms is the ability to deal with concept drift. In data streams, the underlying data distribution can change over time. This phenomenon is called *concept drift*. Concept drift can cause predictions to become less accurate over time (Widmer and Kubat, 1996).

Kelly et al. (Kelly et al., 1999) presented three ways in which concept drift may occur: the chance of prior probabilities of classes, the change of class-conditional probability distributions, and the change of posterior probabilities. Our study is restricted to the second type of concept drift where the conditional distribution of the output changes.

Dealing with Concept Drift in Training Phase: when the conditional distribution changes in the output, there are two solutions to prevent deterioration of prediction accuracy (Kelly et al., 1999). Active solutions (called concept drift detectors) explicitly detect concept drift in the data generating process. Usually such models have no ability to learn the stream other than detecting distribution changes. On the contrary, passive solutions build models for learning data streams which are continuously updated. This can be achieved by keeping a sliding window which stores the most recent data points in the stream (Widmer and Kubat, 1996) or employing an ensemble of classifiers (Elwell and Polikar, 2011).

The model proposed in this paper is a passive predictive model with adaptive learning. It uses blind adaptation strategies (Elwell and Polikar, 2011) without any explicit detection of concept drift. As (Elwell and Polikar, 2011) explained, “the blind approaches forget old concepts at a constant rate whether changes are happening or not. As time goes on, the newly arrived data tend to erase away the prior patterns”.

[2] Although the constant time of training k examples with MB-GD and SGD differs.

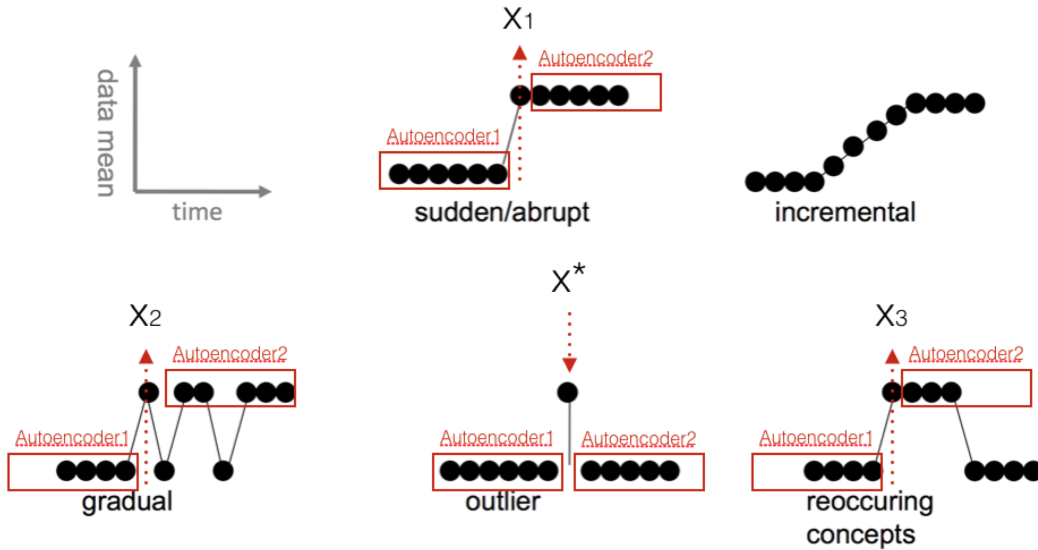

FIGURE 4: Configuration of changes over time. Only x^* is the outlier we want to detect, and the rest x_1, x_2, x_3 will usually be detected in the first phase of SA, but eliminated in the second phase.

In neural networks, learning is always associated with forgetting. The newly arrived data update the weights of neural networks, which makes the effects of old data in the stream become less and less important. Since streaming data arrive at a high speed, the weights in the models get updated rapidly and the model adjusts to the concept drift quickly.

Moreover, the use of threaded ensemble learning makes our model less sensitive to noises and more stable in adapting to concept drift. The ability of continuously learning from data streams while keeping previously learned knowledge is known as the stability–plasticity dilemma (Carpenter et al., 1991). This dilemma happens because "there is a trade-off between handling noise in a stable way and learning new patterns" (Carpenter et al., 1991). Since we split the data into multiple threads, the chance of noises affecting all threads is small compared to a single neural network. If a concept drift is truly happening, it will show up in multiple threads and be detected and adapted by the majority vote of the neural networks in the ensemble.

Dealing with Concept Drift in Testing Phase: with the additional checking step, SA has the ability to distinguish the real outliers from concept drift during the testing phase. Figure 4 shows five different types of changes over time that may occur in a single variable data stream, the detailed description of these changes can be found in (Gama et al., 2014) and (Kuncheva, 2004).

As shown in figure 4, only x^* is the real anomaly we want to detect. The rest of x_1, x_2, x_3 will be detected by most of the outliers detectors, as well as by the first detection phase of SA. However, x_1, x_2, x_3 are points that really represent concept drift rather than real outliers. In the checking phase of SA, these points will likely be labeled as normal because the autoencoders built from adjacent windows before and after $x_i, i = 1, 2, 3$ will disagree which indicate concept drift are detected instead of outliers.

4. EXPERIMENTAL SETUP

In this section, we first describe the datasets that are used in the experiments. We then discuss the experimental settings for our algorithm and the baseline algorithms.

4.1. Datasets

The datasets used in our experiments are summarized in Columns 2 to 4 of Table 3. We used typical benchmark datasets for comparisons. SMTP, HTTP and SMTP+HTTP are streaming anomaly data published from the KDD Cup. Note that SMTPHTTP is constructed by using data from SMTP followed by data from HTTP. This dataset involves concept drift as more details can be found in (Tan et al., 2011).

The fourth dataset COVERTYPE is from the UCI repository. It has some distribution changes since there are multiple different cover-types in the normal class (Tan et al., 2011). We used the smallest class Cottonwood/Willow with 2747 instances as the anomaly class. The next dataset we used is the SHUTTLE dataset from UCI repository. This dataset has no distribution change.

Weather dataset is a sensor dataset we collected from the Meteorological Assimilation Data Ingest System[3] . It consists of meteorological data collected from Road Weather Information System (RWIS). We use sensor readings such as temperature, humidity, dew points etc. to predict the road surface conditions. We extracted the RWIS data from December 1st, 2014 to December 1st, 2015 in the area of Iowa restricted in the square defined from (43.492237, -96.573944) to (40.415721, -91.404758) [4]. We use classes 1,2,4 and 9 as normal instances, and all the other classes (Snow/Ice Warning, Black Ice Warning, etc.) as anomalies.

Table 2 summarizes the datasets used in the experiments.

Table 2: Benchmark datasets

Dataset	Data Size	Dimensionality	Anomaly
HTTP	623091	41	6.49%
SMTP	96554	41	12.25%
SMTP+HTTP	719645	41	7.26%
COVERTYPE	581012	54	0.47%
SHUTTLE	14500	10	5.97%
Weather	1,719,781	32	3.53%

4.2. Experimental Settings

We compared the performance of SA with two popular tree based anomaly detectors for stream learning: Streaming Half-Space-Trees (HSTa) (Tan et al., 2011), and Very Fast Decision Trees (VFDT) (Domingos and Hulten, 2000). We also compared our model with SMLP (Dong and Japkowicz, 2016), a previous model of SA proposed by the same authors of this paper.

HSTa was proposed recently as a one-class anomaly detector for streaming data[5]. It is an algorithms based on random trees. We used the default parameters settings as described in the original paper (i.e., window size = 512, ensemble size = 30, and tree depth = 15). VFDT is a state-of-the-art classification algorithm for data streams. RMOA package in R is used for

[3] https://madis-data.noaa.gov/madisPublic1/
[4] (43.492237, -96.573944) is the upper left corner of the square and (40.415721, -91.404758) is the lower right corner.
[5] https://sites.google.com/site/analyticsofthings/recent-work-fast-anomalydetection-for-streaming-data

the comparison of VFDT (the underlying program is written in Java). The default parameters are used with a chunk size of 512.

We implemented SMLP and SA in Python with h2o deep learning model. The parameter settings for SMLP, SA and SA2 (SA with an additional checking step) are 8 threads with chunk sizes of 512. For Streaming Autoencoders, we use input dropout of 0.1 and Tanh functions with 50% dropout as the activation functions.

The area under the curve (AUC) based on anomaly scores is used as the evaluation metric. In HSTa, SMLP and VFDT, the anomaly scores are calculated as the probability that a point is predicted as normal. Thus, a true anomaly instance generally has a low anomaly score, while a normal point has a high probability and therefore has high anomaly scores. For SA and SA2, we use the inverse of the reconstruction error as the anomaly score, because a testing data point with a smaller reconstruction error is more likely to be a normal instance.

Based on the anomaly scores and the true labels, we then computed the AUC scores for all anomaly detectors. AUC is the main indicator reported in this paper as in Table 3, which ranges from 0 (the worse) to 1 (the best).

All experiments were conducted on a 2.7 GHz Intel Core i5 CPU with 8GB RAM. The order of the data instances is preserved in each data stream. We split the data stream into 80% for training and 20% for testing. During the testing stage, all algorithms perform test-and-train process each time a new segment (chunk size 512) of data arrives. The results reported are averaged over ten runs; each run is obtained using a different random seed for all non-deterministic algorithms. Pairwise t-tests with 5% significance level were used for comparisons.

5. EXPERIMENTAL RESULTS

Comparative Results. Table 3 presents the AUC scores of the algorithms considered in this experiments. First, VFDT and SMLP are both multi-class classifiers. They are expected to perform better than other one-class learners because VFDT and SMLP use data from both classes. In contrast, HSTa and Streaming Autoencoders use only "normal" data for training, by which we mean data from the positive class.

Based on Table 3, we can conclude SA2 outperforms other algorithms on the majority benchmark datasets. We could calculate the pairwise win-loss-tie statistics after running the t-tests. The results between SA2 and SMLP, VFDT, HSTa, are 3-1-2, 4-2-0, 4-2-0, respectively. SA2 is indeed a competitive algorithm based on the win-loss-tie scores and the averaged AUC over all datasets. Moreover, when there is a strong concept shift as in SMTP+HTTP, SA and SA2 perform significantly better than the other approaches. SA and SA2 also adapt better in Weather dataset where concept drift happens due to the seasonal changes.

Runtime Comparison. Table 4 summarizes the average training and testing time over 10 independent runs for each method. Compared to the other one-class learner, SA and SA2 perform faster than HSTa on all datasets except COVERTYPE. These one-class learners (SA,SA2, and HSTa) perform significantly faster than the other two multi-classes learners, namely VFDT and SMLP.

VFDT is the slowest method among all because it needs to modify the tree structure constantly. VFDT's runtime is on average about 10 times slower than the other algorithms considered, as shown in Table 4. More interestingly, we found that streaming autoencoders (SA and SA2) are always much faster than SMLP in the training stage. This is because it is faster to update the autoencoders when the normal data instances are very similar to each other.

Unlike VFDT, our neural networks based models use incremental learning by adaptively

Dataset	AUC				
	SA	SA2	SMLP	VFDT	HSTa
HTTP	0.977	0.990	0.990	0.996*	0.995*
SMTP	0.997	0.999	0.998v	0.987v	0.880v
SMTP+HTTP	0.989	0.987	0.899v	0.867v	0.946v
COVERTYPE	0.973	0.982	0.994*	0.991*	0.980v
SHUTTLE	0.992	0.994	0.988v	0.990v	0.999*
Weather	0.923	0.904	0.909	0.877v	0.886v
Average	0.975	0.976	0.963	0.951	0.948

Table 3: This table presents the AUC scores of the algorithms considered in this experiments. The last row calculates the average AUC over all datasets. v/* indicates that SA2 (SA with an addtional checker) performs significantly better/worse than the corresponding method (SMLP, VFDT, or HSTa) based on the paired t-tests with a significant leavel of 5%.

Dataset	Training Time (seconds)			Testing Time (seconds)				
	SA and SA2	SMLP	VFDT	SA	SA2	SMLP	VFDT	HSTa
HTTP	20	41	349	12	22	13	399	49
SMTP	6	27	53	1	2	2	53	10
SMTP+HTTP	25	35	363	12	23	13	494	57
COVERTYPE	33	102	358	10	22	15	460	30
SHUTTLE	2	7	10	10	20	13	16	6
Weather	73	112	1260	21	31	27	951	203

Table 4: Training time and testing time comparisions. All algorithms perform a test-and-train process continuously while the testing data arrives. Note that HSTa only use the data in the last window from training data, so the training time is almost zero. Thus, only the testing time (run-time) is recorded.

changing the weights. The backpropagation in multiple threads is very efficient. During the testing stage, all algorithms perform a test-and-train process each time a new segment (chunk size 512) of data arrives. The testing time of SA2 is longer than SA because of the additional verification step. However, the test-and-train process of SA2 is still significantly faster than that of VFDT, and the overall runtime of SA2 is still less than that of HSTa.

We also recorded the runtime on different sizes of training data. Figure 5 shows the runtime comparison between SMLP, SA, SA2 and VFDT using the Weather dataset.

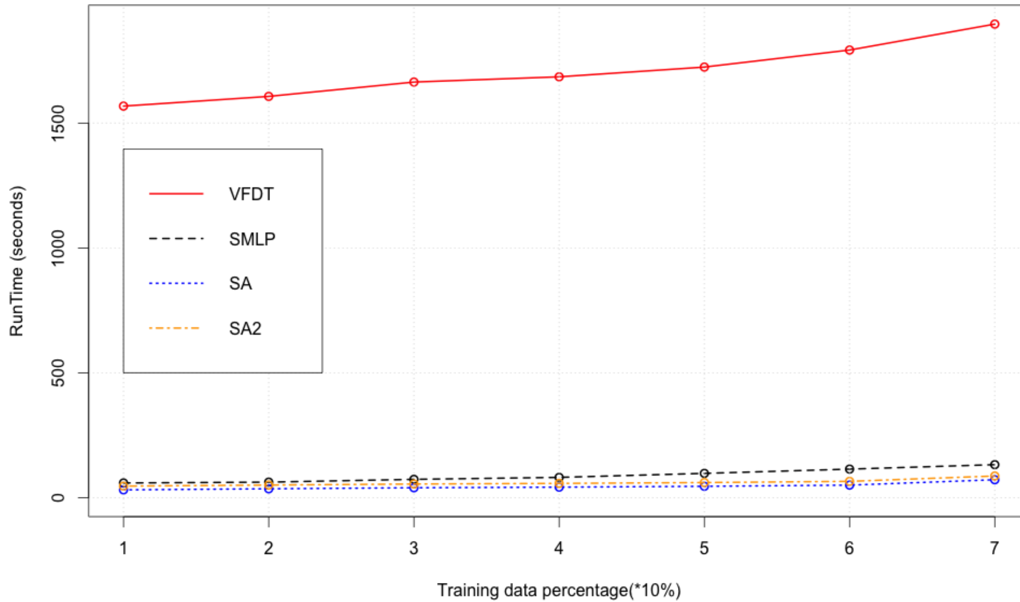

FIGURE 5: Run time (training + testing) comparison between SMLP, SA and VFDT in Weather dataset with increasing training data sizes. Note that HSTa has a constant runtime (203s) when the size of testing data is fixed. This is because HSTa only uses the last window of training data for learning. Therfore, the total runtime is constant no matter how large the training dataset is.

In this experiment, the running time of SMLP, SA and SA2 are linear of the data size n because each instance only requires a constant process time. We can see from figure 5 that when the data size increases from about 200k to 1.2 million, the running time increases from 59 to 132 seconds for SMLP, 31 to 71 seconds for SA, 46 to 86 seconds for SA2.

False Postive Rate Study. As we explained earlier, SA employed a two step verification process to check if an instance is indeed an anomaly. This reduces the false positive rate as shown in figure 6. Compare the AUC of SA and SA2 from table 3, we can see that SA2 is not always performing better than SA in terms of AUC. However, SA2 always outperforms SA in terms of low false alarm rate. This is due to some anomalous instances detected in SA are eliminated in the verification stage of SA2.

In our experiments, we found that the performance of our model is not too sensitive to the number of hidden units we choose in the hidden layers. Suppose the dimensionality (the number of attributes) of the datasets is n. Varying the number of hidden units from $2/3 * n$ to $2 * n$ gives similar results. The effect of randomly picking parameter values reduces as the ensemble size goes up. We can conclude that the insensitivity of our model to the parameters is due to ensemble learning. The random parameter settings in each learner might result in a weak learner, but ensemble several weak learners can lead to a strong learning algorithm.

FIGURE 6: We generate false positive rate (false alarm) curves for SA and SA2 as follows. The x axis represents our six datasets, and the y axis indicates the false positive rate (fpr) in anomaly detections. This process is repeated 10 times for each training set and the averaged fpr is reported.

6. CONCLUSIONS

The proposed stream learning algorithm for anomaly detection, Streaming Autoencoders (without or with the verification step), has the following features for stream learning: (i) it processes endless data streams in one pass without the need to store the entire dataset from streams; (ii) it has the ability to deal with concept drift by learning continuously and adaptively. Moreover, it has the ability to learn different anomalies even if they have not been observed in the training data.

We have identified the key weaknesses for the commonly used tree based algorithms in this paper. The empirical studies show that our neural network based model performs favorably than tree based algorithms in evolving data streams, especially when concept drift is present. Compared to other tree based algorithms, SA2 performs favorably in terms of the win-tie-loss scores and the averaged AUC scores. Our algorithm is comparable to VFDT in terms of AUC scores and outperforms VFDT significantly in terms of the runtime. Compared to HSTa, a state-of-the-art algorithm for one-class learning, our model outperforms it in terms of AUC scores and has the comparable speed as HSTa. Compared to our previous multi-classes model SMLP, SA2 performs better in terms of win-tie-loss in the six datasets tested. Moreover, we are able to improve the speed of our model to 20-40% faster than the previous models. The time and space complexities of our new model are also significantly better with linear training time and constant memory space requirement. Moreover, Streaming Autoencoders can be used with a verification checker to reduce the false alarm rate in anomaly detection. It can also be trained using only data instances from the positive class, which is an advantage when the abnormal instances are rare or not even available for training.

REFERENCES

ABE, NAOKI, BIANCA ZADROZNY, and JOHN LANGFORD. 2006. Outlier detection by active learning. *In* Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 504–509.

BAY, STEPHEN D, and MARK SCHWABACHER. 2003. Mining distance-based outliers in near linear time with randomization and a simple pruning rule. *In* Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 29–38.

BENGIO, YOSHUA. 2009. Learning deep architectures for ai. Foundations and trends® in Machine Learning, **2**(1):1–127.

BOTTOU, LÉON. 2012. Stochastic gradient descent tricks. *In* Neural Networks: Tricks of the Trade. Springer, pp. 421–436.

BRZEZIŃSKI, DARIUSZ. 2010. Mining data streams with concept drift. Ph. D. thesis, Poznan University of Technology.

CARPENTER, GAIL A, STEPHEN GROSSBERG, NATALYA MARKUZON, JOHN H REYNOLDS, and DAVID B ROSEN. 1992. Fuzzy artmap: A neural network architecture for incremental supervised learning of analog multidimensional maps. IEEE Transactions on neural networks, **3**(5):698–713.

CARPENTER, GAIL A, STEPHEN GROSSBERG, and JOHN H REYNOLDS. 1991. Artmap: Supervised real-time learning and classification of nonstationary data by a self-organizing neural network. Neural networks, **4**(5):565–588.

DOMINGOS, PEDRO, and GEOFF HULTEN. 2000. Mining high-speed data streams. *In* Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 71–80.

DONG, YUE, and NATHALIE JAPKOWICZ. 2016. Threaded ensembles of supervised and unsupervised neural networks for stream learning. *In* Canadian Conference on Artificial Intelligence, Springer, pp. 304–315.

ELWELL, RYAN, and ROBI POLIKAR. 2011. Incremental learning of concept drift in nonstationary environments. IEEE Transactions on Neural Networks, **22**(10):1517–1531.

FURAO, SHEN, TOMOTAKA OGURA, and OSAMU HASEGAWA. 2007. An enhanced self-organizing incremental neural network for online unsupervised learning. Neural Networks, **20**(8):893–903.

GAMA, JOÃO, INDRĖ ŽLIOBAITĖ, ALBERT BIFET, MYKOLA PECHENIZKIY, and ABDELHAMID BOUCHACHIA. 2014. A survey on concept drift adaptation. ACM Computing Surveys (CSUR), **46**(4):44.

GUHA, SUDIPTO, and OKKE SCHRIJVERS. 2016. Robust random cut forest based anomaly detection on streams. *In* Proceedings of The 33rd International Conference on Machine Learning, pp. 2712–2721.

HAYES, MICHAEL A, and MIRIAM AM CAPRETZ. 2015. Contextual anomaly detection framework for big sensor data. Journal of Big Data, **2**(1):1.

HELLER, KATHERINE A, KRYSTA M SVORE, ANGELOS D KEROMYTIS, and SALVATORE J STOLFO. 2003. One class support vector machines for detecting anomalous windows registry accesses. *In* Proc. of the workshop on Data Mining for Computer Security.

HUANG, HAO, and SHIVA PRASAD KASIVISWANATHAN. 2015. Streaming anomaly detection using randomized matrix sketching. Proceedings of the VLDB Endowment, **9**(3):192–203.

HULTEN, GEOFF, LAURIE SPENCER, and PEDRO DOMINGOS. 2001. Mining time-changing data streams. *In* Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 97–106.

JAPKOWICZ, NATHALIE, CATHERINE MYERS, and MARK GLUCK. 1995. A novelty detection approach to classification. *In* Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 1, IJCAI'95, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 518–523.

KELLY, MARK G, DAVID J HAND, and NIALL M ADAMS. 1999. The impact of changing populations on classifier performance. *In* Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 367–371.

KUNCHEVA, LUDMILA I. 2004. Combining pattern classifiers: methods and algorithms. John Wiley & Sons.

LE CUN, LEON BOTTOU YANN. 2004. Large scale online learning. *In* Advances in neural information processing systems.

LIU, FEI TONY, KAI MING TING, and ZHI-HUA ZHOU. 2008. Isolation forest. *In* 2008 Eighth IEEE International Conference on Data Mining, IEEE, pp. 413–422.

MASELLI, GAIA, LUCA DERI, and S. SUIN. 2003. Design and implementation of an anomaly detection system: an empirical approach. *In* Proceedings of Terena Networking Conference (TNC 03), Zagreb, Croatia.

PHAM, DUC-SON, SVETHA VENKATESH, MIHAI LAZARESCU, and SAHA BUDHADITYA. 2014. Anomaly

detection in large-scale data stream networks. Data Mining and Knowledge Discovery, **28**(1):145–189.

POLIKAR, ROBI, LALITA UPDA, SATISH S UPDA, and VASANT HONAVAR. 2001. Learn++: An incremental learning algorithm for supervised neural networks. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), **31**(4):497–508.

RUDER, SEBASTIAN. 2016. An overview of gradient descent optimization algorithms. *In* arXiv preprint arXiv:1609.04747.

RUTKOWSKI, LESZEK, MACIEJ JAWORSKI, LENA PIETRUCZUK, and PIOTR DUDA. 2014. Decision trees for mining data streams based on the gaussian approximation. IEEE Transactions on Knowledge and Data Engineering, **26**(1):108–119.

RUTKOWSKI, LESZEK, LENA PIETRUCZUK, PIOTR DUDA, and MACIEJ JAWORSKI. 2013. Decision trees for mining data streams based on the mcdiarmid's bound. IEEE Transactions on Knowledge and Data Engineering, **25**(6):1272–1279.

SCHÖLKOPF, B, RC WILLIAMSON, AJ SMOLA, J SHAWE-TAYLOR, and JC PLATT. 1999. Support vector method for novelty detection. *In* NIPS, The MIT Press, pp. 582–588.

SHEN, FURAO, and OSAMU HASEGAWA. 2010. Self-organizing incremental neural network and its application. *In* International Conference on Artificial Neural Networks, Springer, pp. 535–540.

TAN, SWEE CHUAN, KAI MING TING, and TONY FEI LIU. 2011. Fast anomaly detection for streaming data. *In* Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Two, IJCAI'11, AAAI Press, pp. 1511–1516.

WANG, HAIXUN, WEI FAN, PHILIP S YU, and JIAWEI HAN. 2003. Mining concept-drifting data streams using ensemble classifiers. *In* Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp. 226–235.

WIDMER, GERHARD, and MIROSLAV KUBAT. 1996. Learning in the presence of concept drift and hidden contexts. Machine learning, **23**(1):69–101.

WILSON, D RANDALL, and TONY R MARTINEZ. 2003. The general inefficiency of batch training for gradient descent learning. Neural Networks, **16**(10):1429–1451.

WU, KE, KUN ZHANG, WEI FAN, ANDREA EDWARDS, and S YU PHILIP. 2014. Rs-forest: A rapid density estimator for streaming anomaly detection. *In* 2014 IEEE International Conference on Data Mining, IEEE, pp. 600–609.

Chapter 4

Conclusions and Discussions

In short, as the first main objective, this thesis has explained in details that a polynomial perceptron has finite VC dimension, as well as a neural network with polynomial perceptrons. We have shown that higher order neural networks have larger capacities than first order neural networks with the same structure. Moreover, we have demonstrated empirically that the XOR problem, the famous example that cannot be solved by first order single layer neural networks, could be solved by a single polynomial perceptron.

The second part of the thesis is composed of two papers which have proposed three methods for anomaly detection in data streams. All three models satisfy the requirements for stream learning: (i) they process endless data streams in one pass without the need to store the entire dataset from streams; (ii) they have the ability to deal with concept drift by learning continuously and adaptively. Moreover, two of the three models (SA and SA2) have the ability to learn different anomalies even if they have not been observed in the training data.

We have identified the key weaknesses for the commonly used tree based algorithms in the papers. The empirical studies show that our neural network based models perform favorably than tree based algorithms in evolving data streams, especially when concept drift is present. The time and space complexities of our models are also better with linear training time and constant memory space requirement. Moreover, Streaming Autoencoders can be used with a verification checker to reduce the false alarm rate in anomaly detection. It can also be trained using only data instances from the positive class, which is an advantage when the abnormal instances are rare or not even available for training.

Bibliography

[1] Larry Page. Google 2013 founders‘ letter. `http://investor.google.com/corporate/2013/founders-letter.html`. Accessed: 2016-12-31.

[2] Vladimir Pestov. Is the k-nn classifier in high dimensions affected by the curse of dimensionality? *Computers & Mathematics with Applications*, 65(10):1427–1437, 2013.

[3] Hubert Haoyang Duan. Applying supervised learning algorithms and a new feature selection method to predict coronary artery disease. *arXiv preprint arXiv:1402.0459*, 2014.

[4] Marvin Minsky and Seymour Papert. *Perceptrons.* Oxford, England: M.I.T. Press, 1969.

[5] Yue Dong and Nathalie Japkowicz. *Threaded Ensembles of Supervised and Unsupervised Neural Networks for Stream Learning*, pages 304–315. Springer International Publishing, Cham, 2016.

[6] David S. Moore. *The Basic Practice of Statistics.* W. H. Freeman & Co., New York, NY, USA, 3rd edition, 2007.

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

[8] Geoffrey Hinton. Neural networks lecture 8: Two simple learning algorithms. Coursera, `https://www.coursera.org/learn/neural-networks`, 2016. Accessed: 2016-12-01.

[9] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[10] Martin Anthony and Peter L. Bartlett. *Neural Network Learning: Theoretical Foundations.* Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[11] James McClelland. Training hidden units: The generalized delta rule. https://web.stanford.edu/group/pdplab/originalpdphandbook/Chapter%205.pdf, 2015. Accessed: 2016-12-31.

[12] Bernard Widrow, Marcian E Hoff, et al. Adaptive switching circuits. In *Neurocomputing: Foundations of Research*. IRE WESCON Convention Record, New York IRE, 1960.

[13] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[14] Michael Nielsen. Why are deep neural networks hard to train? http://neuralnetworksanddeeplearning.com/chap5.html, 2016. Accessed: 2016-12-01.

[15] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[16] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In Geoffrey J. Gordon and David B. Dunson, editors, *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)*, volume 15, pages 315–323. Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011.

[17] Ian J. Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C. Courville, and Yoshua Bengio. Maxout networks. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, pages 1319–1327, 2013.

[18] K. Ming Leung. Backpropagation in multilayer perceptrons. http://cis.poly.edu/~mleung/CS6673/s09/BackPropagation.pdf. Accessed: 2016-12-31.

[19] Andrew Ng. Stanford machine learning lecture notes: (17) large scale machine learning. http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html. Accessed: 2016-09-30.

[20] D Randall Wilson and Tony R Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003.

[21] Vera Kurkova. Kolmogorov's theorem and multilayer neural networks. *Neural networks*, 5(3):501–506, 1992.

[22] George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

[23] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[24] Ming Zhang. *Applied Artificial Higher Order Neural Networks for Control and Recognition*. IGI Global, 2016.

[25] C Lee Giles and Tom Maxwell. Learning, invariance, and generalization in high-order neural networks. *Applied optics*, 26(23):4972–4978, 1987.

[26] Mats Bengtsson. *Higher Order Artificial Neural Networks*. Diane Publishing Company, 1990.

[27] J. Zurada. *Introduction to Artificial Neural Systems*. West Publishing Co., St. Paul, MN, USA, 1992.

[28] Luc Devroye, László Györfi, and Gábor Lugosi. *A probabilistic theory of pattern recognition*, volume 31. Springer Science & Business Media, 2013.

[29] Vladimir Pestov. Foundations of statistical machine learning and neural networks. The Vapnik-Chervonenkis theory. Lecture notes, `http://aix1.uottawa.ca/~vpest283/5313/l6.pdf`, 2012.

[30] Vladimir Pestov. Foundations of statistical machine learning and neural networks. Perceptron learning algorithm. Lecture notes, `http://aix1.uottawa.ca/~vpest283/5313/l18.pdf`, 2012.

[31] Martin Anthony and Peter L Bartlett. *Neural network learning: Theoretical foundations*. Cambridge University Press, 2009.

[32] Brendan Murphy. Counting monomials. `https://murphmath.wordpress.com/2012/08/22/counting-monomials/`, 2012. Accessed: 2016-09-30.

[33] JF Feng and B Tirozzi. Learning in a higher-order simple perceptron. *Mathematical and computer modelling*, 30(9):217–223, 1999.

[34] Leon Bottou and Yann LeCun. Large scale online learning. In *Advances in Neural Information Processing Systems (NIPS 2003)*, volume 16. MIT Press, 2004.